**Notes**

Before using this information and the product it supports, be sure to read the general information under "Notices" at the end of this information.

Subsequent editions of this PDF will not be delivered in IBM Publications Center. Always download the latest edition from **IBM Documentation**.
About this information................................................................................................................... xi
Who should read this information.................................................................................................... xii
Db2 Utilities Suite for z/OS.................................................................................................................... xii
Terminology and citations.................................................................................................................. xii
Accessibility features for Db2 for z/OS............................................................................................ xiii
Terminology and citations.................................................................................................................. xii
How to read syntax diagrams.......................................................................................................... xiv

Chapter 1. Planning for and designing Db2 applications................................................................. 1
Application and SQL release incompatibilities.................................................................................. 1
  SUBSTR built-in function always returns an error message for invalid input................................. 2
  CREATE TABLESPACE and CREATE INDEX statements with no space-level USING clause fail if
  the storage group specified when the containing database was created does not exist............... 2
  New maximum number of parameter markers or host variables in a single SQL statement......... 3
  Result change for SQL statement EXPLAIN PACKAGE............................................................... 3
  Result change for SQL statement EXPLAIN STABILIZED DYNAMIC QUERY......................... 3
  SYSCOPY catalog table DSVOLSER column changes................................................................. 3
  Application compatibility levels apply to data definition and data control statements................ 4
  Automatic rebind of plans and packages created before DB2 10.................................................... 4
  KEEPDYNAMIC(YES) bind option support for ROLLBACK........................................................... 5
  Alterations to index compression are pending changes for universal table spaces.......................5
  Data types of output arguments from a stored procedure call in a Java application...................... 6
  SELECT INTO statements with UNION or UNION ALL................................................................. 6
  Change in SQLCODE when the POWER built-in function result is out of range........................... 7
  CHAR9 and VARCHAR9 functions for compatibility with pre-DB2 10 string formatting of
  decimal data .................................................................................................................................... 8
  Subsystem parameter BIF_COMPATIBILITY and SQL schemas for compatibility with pre-DB2
  10 string formatting of decimal data ............................................................................................ 8
  EBCDIC mixed string input to the RTRIM, TRIM, LTRIM, and STRIP built-in functions must be
  valid................................................................................................................................................. 9
  SQL reserved words....................................................................................................................... 10
  Built-in function and existing user-defined functions................................................................... 10
  SQLCODE changes....................................................................................................................... 13
  Determining the value of any SQL processing options that affect the design of your program...... 13
  Changes that invalidate packages.................................................................................................. 14
  Identifying invalidated packages.................................................................................................... 17
  Determining the value of any bind options that affect the design of your program...................... 17
  Programming applications for performance................................................................................ 17
Designing your application for recovery.......................................................................................... 19
  Unit of work in TSO..................................................................................................................... 20
  Unit of work in CICS................................................................................................................... 20
  Planning for program recovery in IMS programs......................................................................... 21
  Undoing selected changes within a unit of work by using savepoints........................................... 27
  Planning for recovery of table spaces that are not logged............................................................ 28
  Designing your application to access distributed data................................................................. 29
  Remote servers and distributed data............................................................................................. 30
  Preparing for coordinated updates to two or more data sources............................................... 31
  Forcing restricted system rules in your program......................................................................... 31

Chapter 2. Connecting to Db2 from your application program...................................................... 33
Chapter 4. Embedded SQL programming ............................................................ 461

Overview of programming applications that access Db2 for z/OS data .................. 461

Declaring table and view definitions ................................................................. 463

DCLGEN (declarations generator) ...................................................................... 464

Generating table and view declarations by using DCLGEN ............................... 464

Including declarations from DCLGEN in your program .................................... 471

Example: Adding DCLGEN declarations to a library ....................................... 472

Defining the items that your program can use to check whether an SQL statement executed successfully ................................................................. 475

Defining SQL descriptor areas (SQLDA). ............................................................. 476

Declaring host variables and indicator variables ................................................. 476

Host variables .................................................................................................. 477

Host-variable arrays ......................................................................................... 478

Host structures ................................................................................................. 479

Indicator variables, arrays, and structures ........................................................ 479

Setting the CCSID for host variables ............................................................... 481

Determining what caused an error when retrieving data into a host variable .......... 482

Accessing an application defaults module ....................................................... 483

Compatibility of SQL and language data types ................................................. 484

Using host variables in SQL statements .......................................................... 487

Retrieving a single row of data into host variables ............................................ 487

Determining whether a retrieved value in a host variable is null or truncated ....... 489

Updating data by using host variables ............................................................. 491

Inserting a single row by using a host variable .................................................. 492

Using host-variable arrays in SQL statements .................................................. 492

Retrieving multiple rows of data into host-variable arrays ............................... 493

Inserting multiple rows of data from host-variable arrays ............................... 493

Inserting null values into columns by using indicator variables or arrays .......... 494

Retrieving a single row of data into a host structure .......................................... 495

Including dynamic SQL in your program ....................................................... 496

Differences between static and dynamic SQL .................................................. 497

Possible host languages for dynamic SQL applications .................................... 500

Including dynamic SQL for non-SELECT statements in your program ............. 501

Including dynamic SQL for fixed-list SELECT statements in your program ...... 502

Including dynamic SQL for varying-list SELECT statements in your program ... 504

Dynamically executing an SQL statement by using EXECUTE IMMEDIATE ......... 519

Dynamically executing an SQL statement by using PREPARE and EXECUTE ....... 520

Dynamically executing a data change statement .............................................. 523

Dynamically executing a statement with parameter markers by using the SQLDA .. 525
Checking the execution of SQL statements ................................................................. 527
Checking the execution of SQL statements by using the SQLCA ............................. 528
Checking the execution of SQL statements by using SQLCODE and SQLSTATE ... 532
Checking the execution of SQL statements by using the WHENEVER statement ... 533
Checking the execution of SQL statements by using the GET DIAGNOSTICS statement 534
Handling SQL error codes ....................................................................................... 539
Arithmetic and conversion errors ............................................................................ 540
Writing applications that enable users to create and modify tables ....................... 540
Saving SQL statements that are translated from user requests ............................. 541
XML data in embedded SQL applications .............................................................. 541
Host variable data types for XML data in embedded SQL applications ................. 542
XML column updates in embedded SQL applications .......................................... 547
XML data retrieval in embedded SQL applications .............................................. 549
Example programs that call stored procedures ..................................................... 552
Assembler applications that issue SQL statements ................................................ 552
Assembler programming examples ........................................................................ 555
Defining the SQL communications area, SQLSTATE, and SQLCODE in assembler 555
Defining SQL descriptor areas (SQLDA) in assembler ........................................... 556
Declaring host variables and indicator variables in assembler .............................. 557
Equivalent SQL and assembler data types ......................................................... 564
Macros for assembler applications ..................................................................... 570
Handling SQL error codes in assembler applications ......................................... 570
C and C++ applications that issue SQL statements ............................................. 571
C and C++ programming examples ....................................................................... 573
Defining the SQL communications area, SQLSTATE, and SQLCODE in C and C++ 582
Defining SQL descriptor areas (SQLDA) in C and C++ ....................................... 583
Declaring host variables and indicator variables in C and C++ ............................. 583
Equivalent SQL and C data types ........................................................................ 611
Handling SQL error codes in C and C++ applications ........................................ 617
COBOL applications that issue SQL statements ................................................ 619
COBOL programming examples ......................................................................... 623
Defining the SQL communications area, SQLSTATE, and SQLCODE in COBOL 648
Defining SQL descriptor areas (SQLDA) in COBOL ............................................. 649
Declaring host variables and indicator variables in COBOL ............................... 650
Equivalent SQL and COBOL data types .............................................................. 677
Object-oriented extensions in COBOL ................................................................. 682
Handling SQL error codes in Cobol applications .............................................. 683
Fortran applications that issue SQL statements .................................................. 684
Defining the SQL communications area, SQLSTATE, and SQLCODE in Fortran 686
Defining SQL descriptor areas in (SQLDA) Fortran ........................................... 687
Declaring host variables and indicator variables in Fortran ............................... 688
Equivalent SQL and Fortran data types ............................................................... 693
PL/I applications that issue SQL statements ....................................................... 695
PL/I programming examples ................................................................................ 699
Defining the SQL communications area, SQLSTATE, and SQLCODE in PL/I 704
Defining SQL descriptor areas (SQLDA) in PL/I ................................................. 705
Declaring host variables and indicator variables in PL/I .................................... 705
Equivalent SQL and PL/I data types ................................................................. 720
REXX applications that issue SQL statements .................................................... 725
REXX programming examples ............................................................................ 727
Defining the SQL communications area, SQLSTATE, and SQLCODE in REXX 742
Defining SQL descriptor areas (SQLDA) in REXX ............................................. 743
Equivalent SQL and REXX data types ............................................................... 743
Accessing the Db2 REXX language support application programming interfaces 745
Ensuring that Db2 correctly interprets character input data in REXX programs .... 747
Passing the data type of an input data type to Db2 for REXX programs ............... 747
Setting the isolation level of SQL statements in a REXX program .................... 748
Retrieving data from Db2 tables in REXX programs .......................................... 749
Chapter 5. Calling a stored procedure from your application........................................ 753
  Passing large output parameters to stored procedures by using indicator variables........................ 758
  Data types for calling stored procedures................................................................................. 758
  How Db2 determines which stored procedure to run............................................................ 763
  Writing a program to receive the result sets from a stored procedure..................................... 768

Chapter 6. Coding methods for distributed data.............................................................. 773
  Accessing distributed data by using three-part table names.................................................. 773
  Accessing declared temporary tables by using three-part table names.................................. 775
  Identifying the server at run time............................................................................................ 778
  SQL limitations at dissimilar servers...................................................................................... 778
  Distributed queries against ASCII or Unicode tables............................................................ 779
  Restrictions when using scrollable cursors to access distributed data..................................... 779
  Restrictions when using rowset-positioned cursors to access distributed data............................ 780
  IBM MQ with Db2.................................................................................................................. 780
    IBM MQ messages................................................................................................................ 780
    Db2 MQ functions and Db2 MQ XML stored procedures....................................................... 782
    Generating XML documents from existing tables and sending them to an MQ message queue... 785
    Shredding XML documents from an MQ message queue.................................................... 785
    Basic messaging with IBM MQ............................................................................................. 794
    Sending messages with IBM MQ............................................................................................ 795
    Asynchronous messaging in Db2 for z/OS............................................................................. 798

Chapter 7. Db2 as a web services consumer and provider................................................. 811
  Deprecated: The SOAPHTTPV and SOAPHTTPC user-defined functions............................. 811
  The SOAPHTTPNV and SOAPHTTPNC user-defined functions............................................ 813
  SQLSTATEs for Db2 as a web services consumer....................................................................... 813

Chapter 8. Application compatibility levels in Db2......................................................... 817
  V12R1Mnn application compatibility levels.............................................................................. 819
  V12R1Mnn application compatibility levels for data server clients and drivers......................... 819
  DSNTIJLC............................................................................................................................... 821
  DSNTIJLR............................................................................................................................... 824
  Using profile tables to control which Db2 for z/OS application compatibility levels to use for specific data server client applications.............................................................. 826
  V11R1 application compatibility level....................................................................................... 828
  V10R1 application compatibility level....................................................................................... 828
  Managing application incompatibilities.................................................................................... 831
Chapter 9. Preparing an application to run on Db2 for z/OS................................. 835

Setting the DB2I defaults................................................................................................. 836
Processing SQL statements.................................................................................................. 839
  Processing SQL statements by using the Db2 precompiler.............................................. 840
  Processing SQL statements by using the Db2 coprocessor.............................................. 846
  Translating command-level statements in a CICS program.............................................. 849
  Differences between the Db2 precompiler and the Db2 coprocessor............................... 850
  Options for SQL statement processing............................................................................ 851
Compiling and link-editing an application........................................................................... 863
Binding application packages and plans............................................................................ 864
  Creating a package version............................................................................................... 865
  Binding a DBRM that is in an HFS file to a package or collection..................................... 866
  Binding an application plan............................................................................................. 869
  Bind process for remote access....................................................................................... 873
  Binding a batch program................................................................................................ 877
  Conversion of DBRMs that are bound to a plan to DBRMs that are bound to a package....... 877
  Converting an existing plan into packages to run remotely............................................. 878
  Setting the program level............................................................................................... 879
  Dynamic rules options for dynamic SQL statements....................................................... 879
  Dynamic plan selection.................................................................................................. 881
Rebinding applications....................................................................................................... 883
  Rebinding a package....................................................................................................... 883
  Rebinding a plan............................................................................................................ 885
  Rebinding lists of plans and packages............................................................................. 886
  Generating lists of REBIND commands.......................................................................... 886
  Automatic rebinds........................................................................................................... 891
Specifying the rules that apply to SQL behavior at run time.............................................. 892
Db2 program preparation overview.................................................................................... 893
Input and output data sets for DL/I batch jobs..................................................................... 895
Db2-specified JCL procedures for preparing an application............................................... 897
  JCL to include the appropriate interface code when using the Db2-specified JCL procedures.. 897
  Tailoring Db2-specified JCL procedures for preparing CICS programs................................ 898
DB2I panels that are used for program preparation............................................................. 900
  Db2 Program Preparation panel..................................................................................... 901
  DB2I Defaults Panel 1.................................................................................................. 905
  DB2I Defaults Panel 2.................................................................................................. 907
Precompile panel................................................................................................................ 908
  Bind Package panel....................................................................................................... 910
  Bind Plan panel............................................................................................................ 912
  Defaults for Bind Package and Defaults for Rebind Package panels............................... 915
  Defaults for Bind Plan and Defaults for Rebind Plan panels........................................ 917
  System Connection Types panel.................................................................................... 919
  Panels for entering lists of values................................................................................ 921
  Program Preparation: Compile, Link, and Run panel....................................................... 922
DB2I panels that are used to rebind and free plans and packages...................................... 923
  Bind/Rebind/Free Selection panel................................................................................ 924
  Rebind Package panel.................................................................................................... 925
  Rebind Trigger Package panel....................................................................................... 927
  Rebind Plan panel......................................................................................................... 928
  Free Package panel....................................................................................................... 930
  Free Plan panel............................................................................................................ 931

Chapter 10. Running an application on Db2 for z/OS.................................................... 933

  DSN command processor............................................................................................... 933
  DB2I Run panel............................................................................................................. 934
Chapter 11. Testing and debugging an application program on Db2 for z/OS........ 947
Designing a test data structure................................................................. 947
Analyzing application data needs.......................................................... 947
Authorization for test tables and applications.......................................... 949
Example SQL statements to create a comprehensive test structure............ 949
Populating the test tables with data......................................................... 950
Methods for testing SQL statements...................................................... 950
Executing SQL by using SPUFI............................................................... 951
Content of a SPUFI input data set............................................................ 954
The SPUFI panel...................................................................................... 955
Changing SPUFI defaults........................................................................ 956
Setting the SQL terminator character in a SPUFI input data set................. 961
Controlling toleration of warnings in SPUFI............................................ 962
Output from SPUFI.................................................................................. 962
Testing an external user-defined function................................................ 964
Testing a user-defined function by using the Debug Tool for z/OS.............. 964
Testing a user-defined function by routing the debugging messages to SYSPRINT 966
Testing a user-defined function by using driver applications..................... 966
Testing a user-defined function by using SQL INSERT statements............ 966
Debugging stored procedures................................................................. 967
Debugging stored procedures with the Debug Tool and IBM VisualAge COBOL 968
Debugging a C language stored procedure with the Debug Tool and C/C++ Productivity Tools for z/OS 969
Debugging stored procedures by using the Unified Debugger.................. 970
Debugging stored procedures with the Debug Tool for z/OS.................... 971
Recording stored procedure debugging messages in a file........................ 972
Driver applications for debugging procedures....................................... 973
Db2 tables that contain debugging information..................................... 973
Debugging an application program......................................................... 973
Locating the problem in an application................................................. 973
Techniques for debugging programs in TSO......................................... 978
Techniques for debugging programs in IMS......................................... 978
Techniques for debugging programs in CICS....................................... 979
Finding a violated referential or check constraint................................... 983

Chapter 12. Sample data and applications supplied with Db2 for z/OS......... 985
Db2 sample tables.................................................................................. 985
Activity table (DSN8C10.ACT).............................................................. 985
Department table (DSN8C10.DEPT)....................................................... 986
Employee table (DSN8C10.EMP)........................................................... 988
Employee photo and resume table (DSN8C10.EMP_PHOTO_RESUME)...... 992
Project table (DSN8C10.PROJ)............................................................. 993
Project activity table (DSN8C10.PROJACT)........................................... 994
About this information

This information discusses how to design and write application programs that access Db2 for z/OS (Db2), a highly flexible relational database management system (DBMS).

Throughout this information, “Db2” means “Db2 12 for z/OS”. References to other Db2 products use complete names or specific abbreviations.

Important: To find the most up to date content for Db2 12 for z/OS, always use IBM® Documentation or download the latest PDF file from PDF format manuals for Db2 12 for z/OS (Db2 for z/OS in IBM Documentation).

Most documentation topics for Db2 12 for z/OS assume that the highest available function level is activated and that your applications are running with the highest available application compatibility level, with the following exceptions:

• The following documentation sections describe the Db2 12 migration process and how to activate new capabilities in function levels:
  – Migrating to Db2 12 (Db2 Installation and Migration)
  – What’s new in Db2 12 (Db2 for z/OS What’s New?)
  – Adopting new capabilities in Db2 12 continuous delivery (Db2 for z/OS What’s New?)
• FL 501 A label like this one usually marks documentation changed for function level 500 or higher, with a link to the description of the function level that introduces the change in Db2 12. For more information, see How Db2 function levels are documented (Db2 for z/OS What’s New?).

The availability of new function depends on the type of enhancement, the activated function level, and the application compatibility levels of applications. In the initial Db2 12 release, most new capabilities are enabled only after the activation of function level 500 or higher.

Virtual storage enhancements
Virtual storage enhancements become available at the activation of the function level that introduces them or higher. Activation of function level 100 introduces all virtual storage enhancements in the initial Db2 12 release. That is, activation of function level 500 introduces no virtual storage enhancements.

Subsystem parameters
New subsystem parameter settings are in effect only when the function level that introduced them or a higher function level is activated. Many subsystem parameter changes in the initial Db2 12 release take effect in function level 500. For more information about subsystem parameter changes in Db2 12, see Subsystem parameter changes in Db2 12 (Db2 for z/OS What’s New?).

Optimization enhancements
Optimization enhancements become available after the activation of the function level that introduces them or higher, and full prepare of the SQL statements. When a full prepare occurs depends on the statement type:

• For static SQL statements, after bind or rebind of the package
• For non-stabilized dynamic SQL statements, immediately, unless the statement is in the dynamic statement cache
• For stabilized dynamic SQL statements, after invalidation, free, or changed application compatibility level

Activation of function level 100 introduces all optimization enhancements in the initial Db2 12 release. That is, function level 500 introduces no optimization enhancements.

SQL capabilities
New SQL capabilities become available after the activation of the function level that introduces them or higher, for applications that run at the equivalent application compatibility level or higher. New SQL capabilities in the initial Db2 12 release become available in function level 500 for applications that
run at the equivalent application compatibility level or higher. You can continue to run SQL statements compatibly with lower function levels, or previous Db2 releases, including Db2 11 and DB2® 10. For details, see Chapter 8, “Application compatibility levels in Db2,” on page 817

Who should read this information

This information is for Db2 application developers who are familiar with Structured Query Language (SQL) and who know one or more programming languages that Db2 supports.

Db2 Utilities Suite for z/OS

Important: Db2 Utilities Suite for z/OS is available as an optional product. You must separately order and purchase a license to such utilities, and discussion of those utility functions in this publication is not intended to otherwise imply that you have a license to them.

Db2 12 utilities can use the DFSORT program regardless of whether you purchased a license for DFSORT on your system. For more information about DFSORT, see https://www.ibm.com/support/pages/dfsort.

Db2 utilities can use IBM Db2 Sort for z/OS as an alternative to DFSORT for utility SORT and MERGE functions. Use of Db2 Sort for z/OS requires the purchase of a Db2 Sort for z/OS license. For more information about Db2 Sort for z/OS, see Db2 Sort for z/OS documentation.

Related concepts
Db2 utilities packaging (Db2 Utilities)

Terminology and citations

When referring to a Db2 product other than Db2 for z/OS, this information uses the product’s full name to avoid ambiguity.

The following terms are used as indicated:

Db2
Represents either the Db2 licensed program or a particular Db2 subsystem.

IBM rebranded DB2 to Db2, and Db2 for z/OS is the new name of the offering that was previously known as “DB2 for z/OS”. For more information, see Revised naming for IBM Db2 family products on IBM z/OS platform. As a result, you might sometimes still see references to the original names, such as “DB2 for z/OS” and “DB2”, in different IBM web pages and documents. If the PID, Entitlement Entity, version, modification, and release information match, assume that they refer to the same product.

IBM OMEGAMON® for Db2 Performance Expert on z/OS
Refers to any of the following products:
• IBM IBM OMEGAMON for Db2 Performance Expert on z/OS
• IBM Db2 Performance Monitor on z/OS
• IBM Db2 Performance Expert for Multiplatforms and Workgroups
• IBM Db2 Buffer Pool Analyzer for z/OS

C, C++, and C language
Represent the C or C++ programming language.

CICS®
Represents CICS Transaction Server for z/OS.

IMS
Represents the IMS Database Manager or IMS Transaction Manager.

MVS™
Represents the MVS element of the z/OS operating system, which is equivalent to the Base Control Program (BCP) component of the z/OS operating system.
RACF®
Represents the functions that are provided by the RACF component of the z/OS Security Server.

Accessibility features for Db2 for z/OS

Accessibility features help a user who has a physical disability, such as restricted mobility or limited vision, to use information technology products successfully.

Accessibility features

The following list includes the major accessibility features in z/OS products, including Db2 for z/OS. These features support:

• Keyboard-only operation.
• Interfaces that are commonly used by screen readers and screen magnifiers.
• Customization of display attributes such as color, contrast, and font size

Tip: IBM Documentation (which includes information for Db2 for z/OS) and its related publications are accessibility-enabled for the IBM Home Page Reader. You can operate all features using the keyboard instead of the mouse.

Keyboard navigation

For information about navigating the Db2 for z/OS ISPF panels using TSO/E or ISPF, refer to the z/OS TSO/E Primer, the z/OS TSO/E User's Guide, and the z/OS ISPF User's Guide. These guides describe how to navigate each interface, including the use of keyboard shortcuts or function keys (PF keys). Each guide includes the default settings for the PF keys and explains how to modify their functions.

Related accessibility information

IBM and accessibility

See the IBM Accessibility Center at http://www.ibm.com/able for more information about the commitment that IBM has to accessibility.

How to send your comments about Db2 for z/OS documentation

Your feedback helps IBM to provide quality documentation.

Send any comments about Db2 for z/OS and related product documentation by email to db2zinfo@us.ibm.com.

To help us respond to your comment, include the following information in your email:

• The product name and version
• The address (URL) of the page, for comments about online documentation
• The book name and publication date, for comments about PDF manuals
• The topic or section title
• The specific text that you are commenting about and your comment

Related concepts
About this information (Db2 for z/OS in IBM Documentation)

Related reference
PDF format manuals for Db2 12 for z/OS (Db2 for z/OS in IBM Documentation)
How to read syntax diagrams

Certain conventions apply to the syntax diagrams that are used in IBM documentation. Apply the following rules when reading the syntax diagrams that are used in Db2 for z/OS documentation:

• Read the syntax diagrams from left to right, from top to bottom, following the path of the line.
  
  The ►►─── symbol indicates the beginning of a statement.
  
  The ───► symbol indicates that the statement syntax is continued on the next line.
  
  The ►─── symbol indicates that a statement is continued from the previous line.
  
  The ───◄ symbol indicates the end of a statement.

• Required items appear on the horizontal line (the main path).

  ►► required_item ──►

• Optional items appear below the main path.

  ►► required_item ───►

  If an optional item appears above the main path, that item has no effect on the execution of the statement and is used only for readability.

  ►► required_item ───►

• If you can choose from two or more items, they appear vertically, in a stack.

  If you must choose one of the items, one item of the stack appears on the main path.

  ►► required_item ───►

  If choosing one of the items is optional, the entire stack appears below the main path.

  ►► required_item ───►

  If one of the items is the default, it appears above the main path and the remaining choices are shown below.

  ►► required_item ───►

• An arrow returning to the left, above the main line, indicates an item that can be repeated.

  ►► required_item ───►

  If the repeat arrow contains a comma, you must separate repeated items with a comma.
A repeat arrow above a stack indicates that you can repeat the items in the stack.

- Sometimes a diagram must be split into fragments. The syntax fragment is shown separately from the main syntax diagram, but the contents of the fragment should be read as if they are on the main path of the diagram.

- For some references in syntax diagrams, you must follow any rules described in the description for that diagram, and also rules that are described in other syntax diagrams. For example:
  - For expression, you must also follow the rules described in Expressions (Db2 SQL).
  - For references to fullselect, you must also follow the rules described in fullselect (Db2 SQL).
  - For references to search-condition, you must also follow the rules described in Search conditions (Db2 SQL).

- With the exception of XPath keywords, keywords appear in uppercase (for example, FROM). Keywords must be spelled exactly as shown. XPath keywords are defined as lowercase names, and must be spelled exactly as shown. Variables appear in all lowercase letters (for example, column-name). They represent user-supplied names or values.

- If punctuation marks, parentheses, arithmetic operators, or other such symbols are shown, you must enter them as part of the syntax.

Related concepts
Commands in Db2 (Db2 Commands)
Db2 online utilities (Db2 Utilities)
Db2 stand-alone utilities (Db2 Utilities)
Chapter 1. Planning for and designing Db2 applications

Before you write or run your program, you need to make some planning and design decisions. These decisions need to be made whether you are writing a new Db2 application or migrating an existing application from a previous release of Db2.

About this task

If you are migrating an existing application from a previous release of Db2, read the application and SQL release incompatibilities and make any necessary changes in the application.

If you are writing a new Db2 application, first determine the following items:

• the value of some of the SQL processing options
• the binding method
• the value of some of the bind options

Then make sure that your program implements the appropriate recommendations so that it promotes concurrency, can handle recovery and restart situations, and can efficiently access distributed data.

Related concepts
Tools and IDEs for developing Db2 applications (Introduction to Db2 for z/OS)

Related tasks
Programming applications for performance (Db2 Performance)
Programming for concurrency (Db2 Performance)
Writing efficient SQL queries (Db2 Performance)
Improving performance for applications that access distributed data (Db2 Performance)

Related reference
BIND and REBIND options for packages, plans, and services (Db2 Commands)

Application and SQL release incompatibilities

When you migrate to or apply maintenance in Db2 12, be aware of and plan for application and SQL release incompatibilities that might affect your Db2 environment.

The following incompatible changes apply at any Db2 12 function level, including when you first migrate to Db2 12. For incompatible changes that might impact your Db2 12 environment when you activate function levels 501 and higher, see Incompatible changes summary for function levels 501 and higher (Db2 for z/OS What's New?).

SQL capabilities
New SQL capabilities become available after the activation of the function level that introduces them or higher, for applications that run at the equivalent application compatibility level or higher. New SQL capabilities in the initial Db2 12 release become available in function level 500 for applications that run at the equivalent application compatibility level or higher. You can continue to run SQL statements compatibly with lower function levels, or previous Db2 releases, including Db2 11 and DB2 10. For details, see Chapter 8, “Application compatibility levels in Db2,” on page 817.

Release incompatibilities that were changed or added since the first edition of this Db2 12 publication are indicated by a vertical bar in the left margin. In other areas of this publication, a vertical bar in the margin indicates a change or addition that has occurred since the Db2 11 release of this publication.
SUBSTR built-in function always returns an error message for invalid input

Previously, during execution of the SUBSTR built-in function, Db2 sometimes incorrectly returned a result for invalid input instead of issuing an appropriate error message. After the PTF for APAR PH36071 is applied and Db2 12 function level 500 or higher is activated, the SUBSTR_COMPATIBILITY subsystem parameter is set to PREVIOUS by default and Db2 continues to behave as before the PTF was applied. If the SUBSTR_COMPATIBILITY subsystem parameter is set to CURRENT, Db2 always enforces the rules for the SUBSTR built-in function that are documented in the SQL Reference and returns an SQL error code if the rules are not met.

For example, if the SUBSTR_COMPATIBILITY subsystem parameter is set to CURRENT, the following query returns an SQL error code:

```
SELECT SUBSTR('ABCD', 2+1, 3) FROM SYSIBM.SYSDUMMY1;
```

Previously, this query incorrectly returned the result 'CD '.

Before you set the SUBSTR_COMPATIBILITY subsystem parameter to CURRENT, you might need to modify some of your applications to handle this change.

Actions to take

In Db2 12, before you set the SUBSTR_COMPATIBILITY subsystem parameter to CURRENT, identify applications that are incompatible with this change by starting a trace for IFCID 0376 and then running the applications. Review the trace output for incompatible changes with the identifier 14. Correct affected applications so that they will be compatible if the SUBSTR_COMPATIBILITY subsystem parameter is set to CURRENT in the future.

Related reference

SUBSTR COMPATIBILITY field (SUBSTR_COMPATIBILITY subsystem parameter) (Db2 Installation and Migration)

CREATE TABLESPACE and CREATE INDEX statements with no space-level USING clause fail if the storage group specified when the containing database was created does not exist

Starting in Db2 12 at function level 500, Db2 records the default storage group for a table space or index in the Db2 catalog. However, Db2 also does not validate the existence of a storage group specified in the STOGROUP clause of a CREATE DATABASE statement. As a result, a CREATE TABLESPACE or CREATE INDEX statement that omits the USING clause at the table space or index level now fails with SQLCODE -204, if the storage group specified when the containing database was created does not exist.

Actions to take

If the storage group specified in a CREATE DATABASE statement does not exist, take one of the following actions:

- Specify a USING clause at the table space or index level in any CREATE TABLESPACE or CREATE INDEX statement that creates a table space or index in that database.
- Issue an ALTER DATABASE statement and specify STOGROUP clause that identifies a storage group that exists.

Related reference

CREATE TABLESPACE (Db2 SQL)
CREATE INDEX (Db2 SQL)
CREATE DATABASE (Db2 SQL)
ALTER DATABASE (Db2 SQL)
New maximum number of parameter markers or host variables in a single SQL statement

Db2 12 enforces the maximum number of parameter markers or host variables in a single SQL statement. Starting in function level 100, Db2 12 issues SQLCODE -101 for any SQL statement that contains more than 16,000 parameter markers or host variables.

Actions to take

Identify and modify any existing SQL statement that contains more than 16,000 parameter markers or host variables.

Related reference

Limits in Db2 for z/OS (Db2 SQL)

Result change for SQL statement EXPLAIN PACKAGE

When Db2 processes the SQL statement EXPLAIN PACKAGE, the HINT_USED column in the PLAN_TABLE is populated with EXPLAIN PACKAGE: copy. The copy field in the HINT_USED column will be one of the following values:

- "CURRENT" - the current copy
- "PREVIOUS" - the previous copy
- "ORIGINAL" - the original copy

This change supports the new rebind phase-in capability that is introduced by function level 505. However, the change takes effect immediately when you migrate to Db2 12.

Actions to take

Change the expected output for queries that reference this column.

Result change for SQL statement EXPLAIN STABILIZED DYNAMIC QUERY

When Db2 processes the SQL statement EXPLAIN STABILIZED DYNAMIC QUERY, the HINT_USED column in the PLAN_TABLE is populated with EXPLAIN PACKAGE: copy. The copy field in the HINT_USED column will be one of the following values:

- "CURRENT" - the current copy
- "INVALID" - the invalid copy

This change supports the new rebind phase-in capability that is introduced by function level 505. However, the change takes effect immediately when you migrate to Db2 12.

Actions to take

Change the expected output for queries that reference this column.

SYSCOPY catalog table DSVOLSER column changes

Db2 12 introduces a new capability to delete only FlashCopy image copies if equivalent sequential image copies exist, for an efficient backup procedure that uses minimal disk space. In support of this capability, the possible values for the DSVOLSER column in the SYSIBM.SYSCOPY catalog table have changed. Previously, the DSVOLSER column value was an empty string for cataloged, sequential, full image copies. Some applications might assume that if the length attribute of the DSVOLSER value is zero, the image copy is cataloged. In Db2 12, that assumption is no longer correct. For cataloged, sequential, full image copies that are created from a FlashCopy image copy with consistency, and also had uncommitted units of work backed out, the DSVOLSER column now contains Db2 checkpoint information.
Actions to take

Modify any applications that use the DSVOLSER column in the SYSCOPY catalog table to tolerate the checkpoint information for cataloged, sequential, full image copies. For details, see the description of DSVOLSER in SYSCOPY catalog table.

Application compatibility levels apply to data definition and data control statements

After the activation of function level 500 or higher in Db2 12, application compatibility levels also control syntax, semantics, default values, and option validation for most data definition statements and data control statements. Data definition statements (sometimes abbreviated as DDL) include various CREATE and ALTER statements. Data control statements (sometimes abbreviated as DCL) include various GRANT and REVOKE statements. Only application compatibility levels V12R1M509 and higher control the behavior of any data definition or data control statements.

The APPLCOMPAT bind option for a package applies to most static SQL data definition and data control statements. The CURRENT APPLICATION COMPATIBILITY special register applies to most dynamic SQL data definition and data control statements.

For implicit regeneration of an object, the application compatibility level that was in effect for the previous CREATE or ALTER statement for that object is used.

For materialization of pending data definition changes, the application compatibility level of the pending ALTER statement is used.

You can specify the USING APPLICATION COMPATIBILITY clause of certain ALTER statements to regenerate an object with a specific application compatibility level.

Related concepts
Function levels and related levels in Db2 12 (Db2 for z/OS What's New?)
Application compatibility levels in Db2

The application compatibility level of your applications controls the adoption and use of new capabilities and enhancements, and the impact of incompatible changes. The advantage is that you can complete the Db2 12 migration process without the need to update your applications immediately.

Related reference
APPLCOMPAT bind option (Db2 Commands)
CURRENT APPLICATION COMPATIBILITY (Db2 SQL)

Automatic rebinding of plans and packages created before DB2 10

Migration-related automatic binds (also called "autobinds") occur in Db2 12 because it cannot use runtime structures from a plan or package that was last bound in a release earlier than DB2 10. Plans and packages that were bound in Db2 11 can run in Db2 12, without the risk of migration-related autobinds. However, plans and packages that are bound in Db2 12 cannot run on Db2 11 members without an autobind in Db2 11.

If you specify YES or COEXIST for the ABIND subsystem parameter, Db2 12 automatically rebinds plans and packages that were bound before DB2 10 when Db2 executes the packages. The result of the automatic bind creates a new package and discards the current copy. Db2 does not move the current copy to the previous or original copy because Db2 12 cannot use it. If a regression occurs, REBIND SWITCH PREVIOUS and REBIND SWITCH ORIGINAL are not available.

If you specify NO for the ABIND subsystem parameter, negative SQLCODEs are returned for each attempt to run a package or plan that was bound before DB2 10. SQLCODE -908, SQLSTATE 23510 is returned for packages, and SQLCODE -923, SQLSTATE 57015 is returned for plans until they are rebound in Db2 12.
Actions to take

By preparing for migration to Db2 12 in Db2 11, you can reduce the change and risk for packages that are subject to automatic binds in Db2 12. To do that, you rebind all packages that were last bound before DB2 10 in Db2 11, before you migrate to Db2 12. For more information about the impacts that migration-related automatic rebinds can have in your Db2 environment and actions that you can take to avoid them, see Rebind old plans and packages in Db2 11 to avoid disruptive autobinds in Db2 12 (Db2 Installation and Migration).

Related reference
AUTO BIND field (ABIND subsystem parameter) (Db2 Installation and Migration)

Related information
-908 (Db2 Codes)
-923 (Db2 Codes)

KEEPDYNAMIC(YES) bind option support for ROLLBACK

In Db2 12, when the APPLCOMPAT value is V12R1M500, the KEEP DYNAMIC(YES) bind option affects both COMMIT and ROLLBACK statements. With KEEP DYNAMIC(YES), the dynamic SQL statements in the package are retained after COMMIT or ROLLBACK, and those statements can run again without another PREPARE.

Prior to Db2 12, the KEEP DYNAMIC(YES) bind option applied only to COMMIT statements. After a ROLLBACK statement, another PREPARE was required so that the dynamic SQL statements could run. This situation is also true in Db2 12 if application compatibility is set to V11R1 or earlier.

In Db2 12, when the APPLCOMPAT value is V12R1M500 or higher, after a ROLLBACK statement is issued, the behavior is different than in prior versions:

- An OPEN statement without a preceding PREPARE statement does not receive an SQLCODE -514.
- An EXECUTE statement without a preceding PREPARE statement does not receive an SQLCODE -518.

An application that was written in Db2 11 and that was bound with KEEP DYNAMIC(YES) was required to prepare dynamic SQL statement again after a ROLLBACK was issued. In Db2 12 when application compatibility is set to V12R1M500 or higher, those extra PREPARE statements are unnecessary.

Actions to take

As you migrate to Db2 12, review packages that use the KEEP DYNAMIC(YES) bind option. You can make dynamic SQL programs that are bound with KEEP DYNAMIC(YES) run more efficiently by removing PREPARE statements that prepare SQL statements again, following execution of ROLLBACK statements. Do not take this action until you are sure that you no longer need to run the programs in Db2 11 or earlier. After migrating to Db2 12, if you take this action (to remove PREPARE statements after ROLLBACK), programs will not work properly if you subsequently set application compatibility to V11R1 or earlier.

Related reference
KEEP DYNAMIC bind option (Db2 Commands)

Alterations to index compression are pending changes for universal table spaces

When the application compatibility level is V12R1M500 or higher, altering to use index compression for indexes in universal table spaces is a pending change that places the index in advisory REORG-pending (AREOR) status. The LOAD REPLACE and REBUILD INDEX utilities no longer materialize the change. You must use an online REORG to materialize the new value for the COMPRESS attribute in the ALTER INDEX statement.

In releases before Db2 12, any alteration to use index compression placed the index in REBUILD-pending (RBBDP) status. You needed to use the REBUILD INDEX utility to rebuild the index, or use the REORG utility to reorganize the table space that corresponds to the index.
**Actions to take**

For indexes in universal table spaces, use an online REORG to materialize the new value for the COMPRESS attribute in the ALTER INDEX statement.

**Related tasks**

Compressing indexes (Db2 Performance)

**Related reference**

ALTER INDEX (Db2 SQL)

---

**Data types of output arguments from a stored procedure call in a Java application**

In function level 500 or higher with application compatibility set to V11R1, when a Java™ application that uses the IBM Data Server Driver for JDBC and SQLJ calls a stored procedure, the data types of stored procedure output arguments match the data types of the parameters in the stored procedure definition.

**Explanation**

Before DB2 10, if a Java client called a Db2 for z/OS stored procedure, the data types of output arguments matched the data types of the corresponding CALL statement arguments. Starting in DB2 10, the data types of the output arguments match the data types of the parameters in the stored procedure definition.

In Db2 12, when application compatibility is set to V10R1, you can set the DDF_COMPATIBILITY subsystem parameter to SP_PARMS_JV to keep the behavior that existed before DB2 10. However, when application compatibility is set to V11R1 or V12R1M100, or to V12R1M500 or higher, SP_PARMS_JV is no longer supported.

In Db2 12 with application compatibility set to V11R1 or V12R1M100, or to V12R1M500 or higher, if the version of the IBM Data Server Driver for JDBC and SQLJ is lower than 3.63 or 4.13, a java.lang.ClassCastException might be thrown when an output argument value is retrieved.

**Actions to take**

Take one of the following actions:

- Upgrade the IBM Data Server Driver for JDBC and SQLJ to version 3.63 or 4.13, or later.
- Modify the data types in CallableStatement.registerOutParameter method calls to match the parameter data types in the stored procedure definitions. You can set application compatibility to V10R1 and run a trace for IFCID 0376 to identify affected applications. Trace records for those applications have a QW0376FN field value of 8.

**Related concepts**

Application compatibility levels in Db2

The application compatibility level of your applications controls the adoption and use of new capabilities and enhancements, and the impact of incompatible changes. The advantage is that you can complete the Db2 12 migration process without the need to update your applications immediately.

---

**SELECT INTO statements with UNION or UNION ALL**

A UNION or UNION ALL is not allowed in the outermost from-clause of a SELECT INTO statement. However, releases before Db2 12 inadvertently tolerate SQL statements that contain this invalid syntax.

The behavior is controlled by the DISALLOW_SEL_INTO_UNION subsystem parameter. In Db2 11, the default setting tolerates the invalid syntax. In Db2 12 the default setting disallows the invalid syntax.

An application that uses the invalid SQL syntax fails at BIND or REBIND with SQLCODE -109.
**Actions to take**

Identify any packages that use UNION or UNION ALL in the from-clause of a SELECT INTO statement and correct them as necessary. You can temporarily specify that Db2 continues to tolerate the invalid syntax NO for the DISALLOW_SEL_INTO_UNION subsystem parameter. However, this subsystem parameter is deprecated and expected to be removed in the future.

You can identify affected packages while DISALLOW_SEL_INTO_UNION is set to NO by binding suspected packages into a dummy collection ID with EXPLAIN(ONLY) and monitoring IFCID 0376 records. Trace records for the affected applications have a QW0376FN field value of 11.

Use the following procedure:

1. Issue the following SQL statement to generate a list of BIND commands.

   ```sql
   SELECT 'BIND PACKAGE(DUMMYCOL) COPY(' ||
   COLLID || ' . ' || NAME || ') ' ||
   CASE WHEN(VERSION <> '')
   THEN 'COPYVER(' || VERSION || ') ' ||
   ELSE '' END ||
   'EXPLAIN(ONLY)' ' FROM SYSIBM.SYSPACKSTMT
   WHERE STATEMENT LIKE '%SELECT%INTO%UNION%' OR STATEMENT LIKE '%SELECT%UNION%INTO';
   ```

   The statement generates output similar to the following BIND subcommand:

   ```sql
   BIND PACKAGE(DUMMYCOL) COPY(DSN_DEFAULT_COLLID_PLAY01.PLAY01) EXPLAIN(ONLY)
   ```

2. Copy the results of the SELECT statement into a bind job. If any BIND subcommands are longer than 72 bytes, formatting is required.
3. Start and collect a trace for IFCID 0376.
4. Run the bind job that you created.
5. Stop the IFCID 0376 trace and analyze the output.

**Related reference**

- **DISALLOW_SEL_INTO_UNION** in macro DSN6SPRM (Db2 Installation and Migration)

**Related information**

- `-109 (Db2 Codes)`

---

**Change in SQLCODE when the POWER built-in function result is out of range**

After the activation of function level 500 or higher in Db2 12, the SQLCODE that is returned when the result of the POWER® built-in function is out of range is changed in some cases.

Previously, when Db2 executed the POWER built-in function, and the result was a DOUBLE data type that was out of range, Db2 returned SQLCODE -802. In Db2 12 with function level 500 or higher activated, SQLCODE +802 is returned.

For example, the following query returns SQLCODE +802:

```sql
SELECT POWER(DOUBLE(2.0E38), DOUBLE(2.0))
FROM SYSIBM.SYSDUMMY1;
```

Invocations of the POWER function that have DOUBLE arguments and return out-of-range results return SQLCODE +802 instead of SQLCODE -802.

**Actions to take**

In Db2 12, before function level 500 or higher is activated, identify applications with this incompatibility by starting a trace for IFCID 0376, and then running the applications. Review the trace output for incompatible changes with the identifier 1201. Adjust error processing to account for the change in the returned SQLCODE from an error to a warning.
Related tasks
Managing application incompatibilities
Before you move an application to a new application compatibility level, you need to find application incompatibilities, adjust your applications for those incompatibilities, and verify that the incompatibilities no longer exist.

Related reference
POWER or POW (Db2 SQL)

CHAR9 and VARCHAR9 functions for compatibility with pre-DB2 10 string formatting of decimal data

DB2 10 changed the formatting of decimal data by the CHAR and VARCHAR built-in functions and CAST specifications with a CHAR or VARCHAR result type. In Db2 12 you can use alternative functions for compatibility with applications that require decimal to string output in the pre-DB2 10 formats:

- CHAR9 (Db2 SQL)
- VARCHAR9 (Db2 SQL)

Important: For portable applications that might run on platforms other than Db2 for z/OS, do not use the CHAR9 and VARCHAR9 functions. Other Db2 family products do not support these functions.

Actions to take

Review your setting for the BIF _COMPATIBILITY subsystem parameter. If the value is not CURRENT, and you have applications that require decimal to string output in the pre-DB2 10 format, you can rewrite SQL statements to use the CHAR9 and VARCHAR9 functions instead. This approach enables the development of new applications that can accept the current string formatting of decimal data.

To modify your applications to take advantage of the CHAR9, VARCHAR9 functions:

1. Use an IFCID 0376 trace to identify applications that depend on the pre-DB2 10 formats.
2. Rewrite the SQL statements in the identified applications to use the CHAR9 and VARCHAR9 functions instead of the CHAR and VARCHAR functions.
3. Set the BIF_COMPATIBILITY value to CURRENT.

Related reference

BIF COMPATIBILITY field (BIF _COMPATIBILITY subsystem parameter) (Db2 Installation and Migration)

Subsystem parameter BIF_COMPATIBILITY and SQL schemas for compatibility with pre-DB2 10 string formatting of decimal data

DB2 10 changed the formatting of decimal data by the CHAR and VARCHAR built-in functions and CAST specifications with a CHAR or VARCHAR result type. You can temporarily override these changes on a subsystem level by setting the BIF_COMPATIBILITY subsystem parameter to one of the pre-DB2 10 settings. You can also temporarily override these changes on an application level by adding schema SYSCOMPAT_V9 to the front of the PATH bind option or CURRENT PATH special register. The latter approach works for CHAR and VARCHAR functions and does not affect CAST specifications.

The recommended approach is to modify your applications to handle the DB2 10 and later behavior for these functions, as described in the following steps.

Actions to take

To modify your applications to handle the DB2 10 and later behavior for CHAR, VARCHAR, and CAST:

1. Identify applications that need to be modified to handle this change. Run a trace for IFCID 0376 to identify affected applications.
2. Ensure that the BIF_COMPATIBILITY subsystem parameter is set to V9_DECIMAL_VARCHAR.
To handle the change for the CHAR function only, you can set `BIF_COMPATIBILITY` to V9, and complete the following steps for the CHAR function.

3. Change any affected applications to handle the DB2 10 and later CHAR and VARCHAR behavior, including stored procedures, non-inline user-defined functions, and trigger packages. Rewrite affected CAST specifications with the appropriate CHAR or VARCHAR function and a CAST to the correct length if needed.

4. Rebind and prepare packages with the PATH(SYSCURRENT,SYSIBM) rebind option. Putting the SYSCURRENT schema at the beginning of the SQL path causes the DB2 10 and later behavior to be used for the CHAR and VARCHAR built-in functions.

   Repeat this step for native stored procedures (SQLPL) and non-inline SQL scalar functions.

5. For views that reference these casts or built-in functions, determine whether the view needs to be changed to have the expected output. Drop and re-create the views with the PATH(SYSCURRENT,SYSIBM) rebind option only if necessary. Rebind any applications that reference the views with the PATH(SYSCURRENT,SYSIBM) option to use the DB2 10 and later CHAR and VARCHAR built-in functions. Repeat this step for inline SQL scalar functions.

6. For materialized query tables or indexes on expressions that reference these casts or built-in functions, drop and re-create the materialized query tables or indexes on expressions with the PATH(SYSCURRENT,SYSIBM) rebind option. Issue the REFRESH TABLE statement for materialized query tables. Rebind any applications that reference the materialized query tables or indexes on expressions with the PATH(SYSCURRENT,SYSIBM) option to use the DB2 10 and later CHAR and VARCHAR built-in functions.

7. Change the value of the `BIF_COMPATIBILITY` subsystem parameter to CURRENT. When the subsystem parameter value is CURRENT, new applications, rebinds, and CREATE statements use the DB2 10 and later CHAR, VARCHAR, and CAST behavior.

   Materialized query tables and expression-based indexes use the CHAR, VARCHAR, and CAST behavior that is specified during its creation. If a reference statement has a different behavior that is specified by the `BIF_COMPATIBILITY` parameter or a different path, the materialized query table or expression-based index is not used.

   **Related reference**
   
   BIF COMPATIBILITY field (`BIF_COMPATIBILITY` subsystem parameter) (Db2 Installation and Migration)

### EBCDIC mixed string input to the RTRIM, TRIM, LTRIM, and STRIP built-in functions must be valid

Starting at application compatibility level V12R1M500 or higher, Db2 12 applies more validation checking for EBCDIC mixed-string input to the RTRIM, TRIM, LTRIM, and STRIP built-in functions.

Generally, Db2 has required valid EBCDIC mixed-string data for input to these functions since version 10, but Db2 12 now detects more cases than earlier releases.

With the new validation checking, when Db2 12 performs a trim operation, and the `string-expression` argument of the RTRIM, TRIM, LTRIM, or STRIP built-in function contains invalid EBCDIC mixed data, Db2 issues SQLCODE -171.

**Actions to take**

Check whether EBCDIC mixed data that is specified for the `string-expression` argument of an RTRIM, TRIM, LTRIM, or STRIP built-in function is valid, and resolve the invalid data.

In valid mixed data, the double-byte portions of the input strings begin with X'0E' (shift-out character), end with X'0F' (shift-in character), and have an even number of bytes between the X'0E' and X'0F' characters. Data that does not meet these criteria is invalid mixed data. When invalid mixed data is specified for the `string-expression` argument of an RTRIM, TRIM, LTRIM, or STRIP built-in function, SQLCODE -171 is returned in some cases. If Db2 trims the specified characters before it reaches an invalid portion of a mixed string, the trim operation is successful.
Setting an application compatibility of V12R1M100 or lower (if PTF for APAR PH25783 is applied) might avoid the error while you resolve the invalid data. At the lower application compatibility levels, Db2 12 attempts to tolerate the invalid mixed data identified by the new validation checking, and it allows the trim operation to complete if possible. However, if Db2 is still unable to perform the trim operation, SQLCODE -171 is issued.

Related information
-171 (Db2 Codes)

SQL reserved words

Db2 12 introduces several new SQL reserved words, which are listed in Reserved words (Db2 SQL).

In some cases, the use of these reserved words might cause an incompatibility before new function is activated in Db2 12, regardless of the setting of the APPLCOMPAT flag.

Actions to take

Collect IFCID 0376 trace records in Db2 11. Values 4, 5, and 6 for the QW0376FN field indicate instances of reserved words in applications that will cause an incompatibility in Db2 12. Adjust these applications by changing the reserved word to a delimited identifier or by using a word that is not reserved in Db2 12.

Built-in function and existing user-defined functions

For built-in and user-defined functions the combination of the function name and the parameter list form the signature that Db2 uses to identify the function. If the signatures of new or changed built-in functions in Db2 12 match the signatures existing user-defined functions, applications with unqualified references to the existing user-defined functions might start invoking the new or changed built-in functions instead of the user-defined functions. Db2 12 introduces the following built-in function changes:

Db2 12 introduces or changes the following built-in functions.

Important information about existing user-defined functions: When a new application compatibility level introduces a new or changed built-in function that has the same name and signature as an existing user-defined function, unqualified references to the user-defined function might resolve incorrectly. Applications that have unqualified references to the user-defined function might fail. To avoid this situation, modify applications to explicitly qualify references to user-defined functions with the same name and signature as the new or changed built-in functions.

<table>
<thead>
<tr>
<th>APPLCOMPAT level</th>
<th>Function name</th>
<th>Change introduced</th>
<th>Incompatible change?</th>
</tr>
</thead>
<tbody>
<tr>
<td>V12R1M507</td>
<td>Various</td>
<td>The following functions are newly supported in Db2 for z/OS as passthrough-only expressions, which are passed through to IBM Db2 Analytics Accelerator for z/OS.</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- ADD_DAYS (Db2 SQL)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>- BTRIM (Db2 SQL)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>- DAYS_BETWEEN (Db2 SQL)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>- NEXT_MONTH (Db2 SQL)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>- REGR_AVGX</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>- REGR_AVGY</td>
<td></td>
</tr>
<tr>
<td>APPLCOMP AT level</td>
<td>Function name</td>
<td>Change introduced</td>
<td>Incompatibl e change?</td>
</tr>
<tr>
<td>------------------</td>
<td>---------------</td>
<td>------------------</td>
<td>---------------------</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• REGR_COUNT</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>• REGR_INTERCEPT or REGR_ICPT</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>• REGR_R2</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>• REGR_SLOPE</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>• REGR_SXX</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>• REGR_SXY</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>• REGR_SYY</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>• ROUND_TIMESTAMP (Db2 SQL) if invoked with a date expression</td>
<td></td>
</tr>
<tr>
<td>V12R1M506</td>
<td>HASH (Db2 SQL)</td>
<td>New built-in function.</td>
<td>No</td>
</tr>
<tr>
<td>V12R1M506</td>
<td>CHARACTER_LENGTH or CHAR_LENGTH (Db2 SQL)</td>
<td>CHAR_LENGTH is now supported as an alternative function name.</td>
<td>No</td>
</tr>
<tr>
<td>V12R1M506</td>
<td>CLOB (Db2 SQL)</td>
<td>TO_CLOB is now supported as an alternative function name.</td>
<td>No</td>
</tr>
<tr>
<td>V12R1M506</td>
<td>COVAR_POP or COVARIANCE or COVAR (Db2 SQL)</td>
<td>COVAR_POP is now supported as an alternative function name.</td>
<td>No</td>
</tr>
<tr>
<td>V12R1M506</td>
<td>LEFT (Db2 SQL)</td>
<td>STRLEFT is now supported as an alternative function name.</td>
<td>No</td>
</tr>
<tr>
<td>V12R1M506</td>
<td>POWER or POW (Db2 SQL)</td>
<td>POW is now supported as an alternative function name.</td>
<td>No</td>
</tr>
<tr>
<td>V12R1M506</td>
<td>POSSTR (Db2 SQL)</td>
<td>STRPOS is now supported as an alternative function name.</td>
<td>No</td>
</tr>
<tr>
<td>V12R1M506</td>
<td>RANDOM or RAND (Db2 SQL)</td>
<td>RANDOM is now supported as an alternative function name.</td>
<td>No</td>
</tr>
<tr>
<td>V12R1M506</td>
<td>RIGHT (Db2 SQL)</td>
<td>STRRIGHT is now supported as an alternative function name.</td>
<td>No</td>
</tr>
<tr>
<td>V12R1M506</td>
<td>TIMESTAMP_FORMAT (Db2 SQL)</td>
<td>TO_TIMESTAMP is now supported as an alternative function name.</td>
<td>No</td>
</tr>
<tr>
<td>V12R1M505</td>
<td>DECRYPT_DATAKEY_INTEGER, DECRYPT_DATAKEY_BIGINT, DECRYPT_DATAKEY_DECIMAL, DECRYPT_DATAKEY_VARCHAR, DECRYPT_DATAKEY_CLOB, DECRYPT_DATAKEY_VARGRAPHIC, DECRYPT_DATAKEY_DBCLOB, and DECRYPT_DATAKEY_BIT</td>
<td>New built-in functions.</td>
<td>No</td>
</tr>
<tr>
<td>V12R1M505</td>
<td>ENCRYPT_DATAKEY</td>
<td>New built-in function.</td>
<td>No</td>
</tr>
<tr>
<td>APPLCOMPAT level</td>
<td>Function name</td>
<td>Change introduced</td>
<td>Incompatibility change?</td>
</tr>
<tr>
<td>------------------</td>
<td>---------------</td>
<td>-------------------</td>
<td>------------------------</td>
</tr>
<tr>
<td>V12R1M504</td>
<td>Various</td>
<td>The following functions are newly supported in Db2 for z/OS as passthrough-only expressions, which are passed through to IBM Db2 Analytics Accelerator for z/OS.</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• CUME_DIST</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>• CUME_DIST (aggregate) (Db2 SQL)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>• FIRST_VALUE</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>• LAG</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>• LAST_VALUE</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>• LEAD</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>• NTH_VALUE</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>• NTILE</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>• PERCENT_RANK</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>• PERCENT_RANK (aggregate) (Db2 SQL)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>• RATIO_TO_REPORT</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>• REGEXP_COUNT (Db2 SQL)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>• REGEXP_INSTR (Db2 SQL)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>• REGEXP_LIKE (Db2 SQL)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>• REGEXP_REPLACE (Db2 SQL)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>• REGEXP_SUBSTR (Db2 SQL)</td>
<td></td>
</tr>
<tr>
<td>V12R1M502</td>
<td>GRAPHIC (Db2 SQL)</td>
<td>The first argument now accepts numeric data types, including SMALLINT, INTEGER, BIGINT, DECIMAL, REAL, DOUBLE, FLOAT, and DECFLOAT.</td>
<td>No</td>
</tr>
<tr>
<td>V12R1M502</td>
<td>VARGRAPHIC (Db2 SQL)</td>
<td>The first argument accepts numeric data types, including SMALLINT, INTEGER, BIGINT, DECIMAL, REAL, DOUBLE, FLOAT, and DECFLOAT.</td>
<td>No</td>
</tr>
<tr>
<td>V12R1M501</td>
<td>LISTAGG (Db2 SQL)</td>
<td>New built-in function.</td>
<td>No</td>
</tr>
<tr>
<td>V12R1M500</td>
<td>ARRAY_AGG (Db2 SQL)</td>
<td>Newly supported built-in function when used for associative array aggregation.</td>
<td>No</td>
</tr>
<tr>
<td>V12R1M500</td>
<td>GENERATE_UNIQUE_BINARY</td>
<td>New built-in function.</td>
<td>No</td>
</tr>
<tr>
<td>V12R1M500</td>
<td>HASH_CRC32, HASH_MD5, HASH_SHA1, and HASH_SHA256</td>
<td>New built-in functions.</td>
<td>No</td>
</tr>
<tr>
<td>V12R1M500</td>
<td>PERCENTILE_CONT</td>
<td>New built-in function.</td>
<td>No</td>
</tr>
<tr>
<td>V12R1M500</td>
<td>PERCENTILE_DISC</td>
<td>New built-in function.</td>
<td>No</td>
</tr>
<tr>
<td>V12R1M500</td>
<td>TRANSLATE (Db2 SQL)</td>
<td>UNI_60 can new be specified as a locale.</td>
<td>No</td>
</tr>
<tr>
<td>V12R1M500</td>
<td>UPPER (Db2 SQL)</td>
<td>UNI_60 can new be specified as a locale.</td>
<td>No</td>
</tr>
<tr>
<td>V12R1M500</td>
<td>WRAP (Db2 SQL)</td>
<td>New built-in function.</td>
<td>No</td>
</tr>
</tbody>
</table>
### Actions to take

To continue to execute a user-defined function with the same name or signature as a new built-in function or signature, qualify the name of the existing user-defined function in your application. For more information about Db2 resolves qualified and unqualified references to functions, see [Function resolution (Db2 SQL)].

### SQLCODE changes

Some SQLCODE numbers and message text might have changed in Db2 12. Also, the conditions under which some SQLCODEs are issued might have changed. For more information, see [New, changed, and deleted codes (Db2 Codes)].

### Determining the value of any SQL processing options that affect the design of your program

When you process SQL statements in an application program, you can specify options that describe the basic characteristics of the program. You can also indicate how you want the output listings to look. Although most of these options do not affect how you design or code the program, a few options do.

#### About this task

SQL processing options specify program characteristics such as the following items:

- The host language in which the program is written
- The maximum precision of decimal numbers in the program
- How many lines are on a page of the precompiler listing

In many cases, you may want to accept the default value provided.

#### Procedure

Review the list of SQL processing options and decide the values for any options that affect the way that you write your program.

For example, you need to know if you are using NOFOR or STDSQL(YES) before you begin coding.

#### Related concepts

- [Db2 program preparation overview](#)
- Before you can run an application program on Db2 for z/OS, you need to prepare it. To prepare the program, create a load module, possibly one or more packages, and an application plan.

#### Related reference

- [Descriptions of SQL processing options](#)
You can specify any SQL processing options regardless of whether you use the Db2 precompiler or the Db2 coprocessor. However, the Db2 coprocessor might ignore certain options because host language compiler options exist that provide the same information.

## Changes that invalidate packages

Changes to your program or database objects can invalidate packages. A change to your program probably invalidates one or more of your packages. For some changes, you must bind a new object. For others, rebinding is sufficient. A package can also become invalid for reasons that do not depend on operations in your program. For example, when an index is dropped that is used in an access path by one of your queries, a package can become invalid.

Db2 might rebind invalid packages automatically the next time that the package is run. For details, see “Automatic rebinds” on page 891.

Db2 marks packages invalid when they depend on the target object, and sometimes on related objects that are affected by cascading effects, of the following actions:

- **Altering tables:**
  - Adding a TIME, TIMESTAMP, or DATE column when the default value for added rows is CURRENT DATE, CURRENT TIME, CURRENT TIMESTAMP (p) WITHOUT TIME ZONE, or CURRENT TIMESTAMP (p) WITH TIME ZONE respectively
  - Adding a constraint with a delete rule of SET NULL or CASCADE. Packages that depend on tables that cascade deletes to the altered parent table are also invalidated.
  - Adding a security label
  - Altering a column
  - Renaming a column (Cascading effects apply. See “Cascading effects on packages of renaming a column” on page 15.)
  - Altering a column such that a view cannot regenerate
  - Altering the AUDIT attribute
  - Dropping a column. For pending definition changes, the package invalidation occurs when the pending definition change is applied to the table.
  - Altering for hash organization, or dropping hash organization
  - Adding or removing a BUSINESS_TIME period for temporal versioning
  - Enabling or disabling transparent archiving
  - Adding, altering, or dropping a materialized query table (MQT) definition
  - Dropping a clone table
  - Activating or deactivating row-level access control
  - Activating column-level access control if the table has an enabled column, or deactivating column-level access control
  - For created temporary tables, adding a column

- **Altering views:**
  - Altering a view to regenerate it

- **Altering table spaces:**
  - Changing the SBCS CCSID attribute
  - Increasing the MAXPARTITIONS attribute
  - Changing the SEGSIZE attribute to convert the table space to a partition-by-range (UTS) table space
  - Changing the DSSIZE attribute of a partitioned table space
  - Changing the buffer pool page size

- **Altering partitions of partition-by-range (PBR) or partitioned (non-UTS) table spaces:**
– Adding partitions
– Altering limit keys
– Rotating partitions

• Materializing pending definition changes to table spaces with the REORG TABLESPACE utility. For more information, see Pending data definition changes (Db2 Administration Guide).

• Altering indexes:
  – Adding a column
  – Altering an index to regenerate it
  – Altering the PADDED or NOT PADDED attribute
  – Altering a limit key value of a partitioning index
  – Specifying NOT CLUSTER for the partitioning index of a table that uses index-controlled partitioning, to convert the table to use table controlled partitioning

• Materializing pending definition changes to indexes with the REORG INDEX utility. For more information, see Pending data definition changes (Db2 Administration Guide).

• Regenerating procedures. See the information about invalidation of packages in ALTER PROCEDURE (SQL - native) (Db2 SQL) for more information.

• Altering functions:
  – Altering an external function
  – Altering an inlined SQL scalar function
  – Altering a version of a compiled SQL scalar function to change certain options that are specified for the active version. See the information about invalidation of packages in ALTER FUNCTION (compiled SQL scalar) (Db2 SQL) for more information.
  – Altering SQL table functions:
    - Altering the SECURED or NOT SECTURED attribute
    - Altering the DETERMINISTIC or NOT DETERMINISTIC attribute, regardless of whether RESTRICT is specified
    - Regenerating a table function

• Enabling or disabling masks if column access control is in effect
• Dropping the package
• Dropping a package that provides the execute privilege for a plan
• Dropping objects such as aliases, global variables, indexes, materialized query tables, roles, sequences, tables, table spaces, triggers, views
• If column access control is enforced for a table, dropping row permissions or column masks
• Revoking authorization from the package owner to access a table, index, or view
• Revoking authorization from the package owner execute a stored procedure, if the package uses the CALL procedure-name form of the CALL statement to call the stored procedure
• Running the REORG utility with the REBALANCE keyword
• Running the REPAIR utility on a database with the DBD REBUILD option

Cascading effects on packages of renaming a column

ALTER TABLE RENAME COLUMN will invalidates any package that depends on the table in which the column is renamed. Any attempt to execute the invalidated package triggers an automatic rebind of the package.

The automatic rebind fails if the column is referenced in the package because the referenced column no longer exists in the table. In this case, applications that reference the package need to be modified, recompiled, and rebound to return the expected result.
The automatic rebind succeeds in either of the following cases:

- The package does not reference the column. In this case, the renaming of the column does not affect the query results that are returned by the package. The application does not need to be modified as a result of renaming the column.

- The package does reference the column, but after the column is renamed, another column with the name of the original column is added to the table. In this case, any query that references the name of the original column might return a different result set. In order to restore the expected results, the application would need to be modified to specify the new column name.

**Example**

The following scenario shows how renaming a column can cause a package to return unexpected results:

```sql
CREATE TABLE MYTABLE (MYCOL1 INT);
INSERT INTO TABLE MYTABLE
VALUES (1);
SELECT MYCOL1 FROM MYTABLE -- this is the statement in
-- the package MYPACKAGE,
-- a value of 1
ALTER TABLE MYTABLE
RENAME COLUMN
MYCOL1 TO MYCOL2;      -- MYPACKAGE is invalidated
-- and automatic rebind
-- of MYPACKAGE will fail
-- at this point
ALTER TABLE MYTABLE
ADD COLUMN MYCOL1 VARCHAR(10);  -- automatic rebind
-- of MYPACKAGE
-- will be successful
INSERT INTO TABLE MYTABLE (MYCOL1)
VALUES ('ABCD');
```

At this point an application executes MYPACKAGE, which results in a successful automatic rebind. However, the statement in the package will return 'ABCD' instead of the expected '1'.

**Related concepts**

Automatic rebinds

*Automatic rebinds* (sometimes called "autobinds") occur when an authorized user runs a package or plan and the runtime structures in the plan or package cannot be used. This situation usually results from changes to the attributes of the data on which the package or plan depends, or changes to the environment in which the package or plan runs.

“Trigger packages” on page 160

A *trigger package* is a special type of package that is created only when you execute a CREATE TRIGGER statement. A trigger package executes only when the associated trigger is activated.

**Related tasks**

Identifying packages with characteristics that affect performance, concurrency, or the ability to run (Db2 Performance)

“Rebinding applications” on page 883

You must rebind applications to change bind options. You also need to rebind applications when you make changes that affect the plan or package, such as creating an index, but you have not changed the SQL statements.

**Related reference**

Invalid and inoperative packages (Managing Security)

**Related information**

00E30305 (Db2 Codes)
Identifying invalidated packages

You can identify packages that will become invalidated when certain changes are made to objects.

About this task

Certain changes to objects invalidate packages. By identifying these invalidated packages before you make the changes, you can prepare necessary rebind operations accordingly.

Procedure

To identify all packages that will be invalidated by a change to a specific object, run the following query:

```sql
SELECT DISTINCT DCOLLID, DNAME, DTYPE
FROM SYSIBM.SYSPACKDEP
WHERE BQUALIFIER = object_qualifier
  AND BNAME = object_name
  AND BTYPE = object_type
ORDER BY DCOLLID, DNAME;
```

- **object_qualifier**
  The qualifier of the object

- **object_name**
  The name of the object

- **object_type**
  The type of object

Results

The query returns a table that contains package information based on the selected values in the query. For details about the selected values, see SYSPACKDEP catalog table (Db2 SQL).

Determining the value of any bind options that affect the design of your program

Several options of the BIND PACKAGE and BIND PLAN commands can affect your program design. For example, you can use a bind option to ensure that a package or plan can run only from a particular CICS connection or IMS region. Your code does not need to enforce this situation.

Procedure

Review the list of bind options and decide the values for any options that affect the way that you write your program.

For example, you should decide the values of the ACQUIRE and RELEASE options before you write your program. These options determine when your application acquires and releases locks on the objects it uses.

Related reference

BIND and REBIND options for packages, plans, and services (Db2 Commands)

Programming applications for performance

You can achieve better Db2 performance by considering performance as you program and deploy your applications.

Procedure

To improve the performance of application programs that access data in Db2, use the following approaches when writing and preparing your programs:
• Program your applications for concurrency.
  The goal is to program and prepare applications in a way that:
  – Protects the integrity of the data that is being read or updated from being changed by other
    applications.
  – Minimizes the length of time that other access to the data is prevented.
For more information about data concurrency in Db2 and recommendations for improving concurrency
in your application programs, see the following topics:
  – Designing databases for concurrency (Db2 Performance)
  – Concurrency and locks (Db2 Performance)
  – Improving concurrency (Db2 Performance)
  – Improving concurrency in data sharing environments (Db2 Data Sharing Planning and
    Administration)
• Write SQL statements that access data efficiently.
  The predicates, subqueries, and other structures in SQL statements affect the access paths that Db2
  uses to access the data.
For information about how to write SQL statements that access data efficiently, see the following
topics:
  – Ways to improve query performance (Introduction to Db2 for z/OS)
  – Writing efficient SQL queries (Db2 Performance)
• Use EXPLAIN or SQL optimization tools to analyze the access paths that Db2 chooses to process your
  SQL statements.
  By analyzing the access path that Db2 uses to access the data for an SQL statement, you can discover
  potential problems. You can use this information to modify your statement to perform better.
For information about how you can use EXPLAIN tables to analyze the access paths for your SQL
statements, see the following topics:
  – Investigating access path problems (Db2 Performance)
  – 00C200A4 (Db2 Codes)
  – Investigating SQL performance by using EXPLAIN (Db2 Performance)
  – Interpreting data access by using EXPLAIN (Db2 Performance)
  – EXPLAIN tables (Db2 Performance)
  – EXPLAIN (Db2 SQL)
• Consider performance in the design of applications that access distributed data.
  The goal is to reduce the amount of network traffic that is required to access the distributed data, and
to manage the use of system resources such as distributed database access threads and connections.
For information about improving the performance of applications that access distributed data, see the
following topics:
  – Ways to reduce network traffic (Introduction to Db2 for z/OS)
  – Managing Db2 threads (Db2 Performance)
  – Improving performance for applications that access distributed data (Db2 Performance)
  – Improving performance for SQL statements in distributed applications (Db2 Performance)
• Use stored procedures to improve performance, and consider performance when creating stored
  procedures.
For information about stored procedures and Db2 performance, see the following topics:
  – Implementing Db2 stored procedures (Stored procedures provided by Db2)
  – Improving the performance of stored procedures and user-defined functions (Db2 Performance)
Related concepts
Query and application performance analysis (Introduction to Db2 for z/OS)
Programming for the instrumentation facility interface (IFI) (Db2 Performance)

Related tasks
Overview of programming applications that access Db2 for z/OS data
Applications that interact with Db2 must first connect to Db2. They can then read, add, or modify data or manipulate Db2 objects.

Setting limits for system resource usage by using the resource limit facility (Db2 Performance)
Planning for and designing Db2 applications
Before you write or run your program, you need to make some planning and design decisions. These decisions need to be made whether you are writing a new Db2 application or migrating an existing application from a previous release of Db2.

Designing your application for recovery
If your application fails or Db2 terminates abnormally, you need to ensure the integrity of any data that was manipulated in your application. You should consider possible recovery situations when you design your application.

Procedure
To design your application for recovery:

1. Put any changes that logically need to be made at the same time in the same unit of work. This action ensures that in case Db2 terminates abnormally or your application fails, the data is left in a consistent state.

   A unit of work is a logically distinct procedure that contains steps that change the data. If all the steps complete successfully, you want the data changes to become permanent. But, if any of the steps fail, you want all modified data to return to the original value before the procedure began. For example, suppose two employees in the sample table DSN8C10.EMP exchange offices. You need to exchange their office phone numbers in the PHONENO column. You need to use two UPDATE statements to make each phone number current. Both statements, taken together, are a unit of work. You want both statements to complete successfully. For example, if only one statement is successful, you want both phone numbers rolled back to their original values before attempting another update.

2. Consider how often you should commit any changes to the data.

   If your program abends or the system fails, Db2 backs out all uncommitted data changes. Changed data returns to its original condition without interfering with other system activities.

   For IMS and CICS applications, if the system fails, Db2 data does not always return to a consistent state immediately. Db2 does not process indoubt data (data that is neither uncommitted nor committed) until you restart IMS or the CICS attachment facility. To ensure that Db2 and IMS are synchronized, restart both Db2 and IMS. To ensure that Db2 and CICS are synchronized, restart both Db2 and the CICS attachment facility.

3. Consider whether your application should intercept abends.

   If your application intercepts abends, Db2 commits work, because it is unaware that an abend has occurred. If you want Db2 to roll back work automatically when an abend occurs in your program, do not let the program or run time environment intercept the abend. If your program uses Language Environment®, and you want Db2 to roll back work automatically when an abend occurs in the program, specify the run time options ABTERMENC(ABEND) and TRAP(ON).

4. For TSO applications only: Issue COMMIT statements before you connect to another DBMS.

   If the system fails at this point, Db2 cannot know whether your transaction is complete. In this case, as in the case of a failure during a one-phase commit operation for a single subsystem, you must make your own provision for maintaining data integrity.

5. For TSO applications only: Determine if you want to provide an abend exit routine in your program.
If you provide this routine, it must use tracking indicators to determine if an abend occurs during Db2 processing. If an abend does occur when Db2 has control, you must allow task termination to complete. Db2 detects task termination and terminates the thread with the ABRT parameter. Do not re-run the program.

Allowing task termination to complete is the only action that you can take for abends that are caused by the CANCEL command or by DETACH. You cannot use additional SQL statements at this point. If you attempt to execute another SQL statement from the application program or its recovery routine, unexpected errors can occur.

Related concepts
Unit of work (Introduction to Db2 for z/OS)

Unit of work in TSO
Applications that use the TSO attachment facility can explicitly define units of work by using the SQL COMMIT and ROLLBACK statements.

In TSO applications, a unit of work starts when the first updates of a Db2 object occur. A unit of work ends when one of the following conditions occurs:

• The program issues a subsequent COMMIT statement. At this point in the processing, your program has determined that the data is consistent; all data changes that were made since the previous commit point were made correctly.
• The program issues a subsequent ROLLBACK statement. At this point in the processing, your program has determined that the data changes were not made correctly and, therefore, should not be permanent. A ROLLBACK statement causes any data changes that were made since the last commit point to be backed out.
• The program terminates and returns to the DSN command processor, which returns to the TSO Terminal Monitor Program (TMP).

The first and third conditions in the preceding list are called a commit point. A commit point occurs when you issue a COMMIT statement or your program terminates normally.

Related reference
COMMIT (Db2 SQL)
ROLLBACK (Db2 SQL)

Unit of work in CICS
CICS applications can explicitly define units of work by using the CICS SYNCPOINT command. Alternatively, units of work are defined implicitly by several logic-breaking points.

All the processing that occurs in your program between two commit points is known as a logical unit of work (LUW) or unit of work. In CICS applications, a unit of work is marked as complete by a commit or synchronization (sync) point, which is defined in one of following ways:

• Implicitly at the end of a transaction, which is signaled by a CICS RETURN command at the highest logical level.
• Explicitly by CICS SYNCPOINT commands that the program issues at logically appropriate points in the transaction.
• Implicitly through a DL/I PSB termination (TERM) call or command.
• Implicitly when a batch DL/I program issues a DL/I checkpoint call. This call can occur when the batch DL/I program shares a database with CICS applications through the database sharing facility.

For example, consider a program that subtracts the quantity of items sold from an inventory file and then adds that quantity to a reorder file. When both transactions complete (and not before) and the data in the two files is consistent, the program can then issue a DL/I TERM call or a SYNCPOINT command. If one of the steps fails, you want the data to return to the value it had before the unit of work began. That is, you
want it rolled back to a previous point of consistency. You can achieve this state by using the SYNCPOINT command with the ROLLBACK option.

By using a SYNCPOINT command with the ROLLBACK option, you can back out uncommitted data changes. For example, a program that updates a set of related rows sometimes encounters an error after updating several of them. The program can use the SYNCPOINT command with the ROLLBACK option to undo all of the updates without giving up control.

The SQL COMMIT and ROLLBACK statements are not valid in a CICS environment. You can coordinate Db2 with CICS functions that are used in programs, so that Db2 and non-Db2 data are consistent.

**Planning for program recovery in IMS programs**

To be prepared for recovery situations for IMS programs that access Db2 data, you need to make several design decisions that are specific to IMS programs. These decisions are in addition to the general recommendations that you should follow when designing your application for recovery.

**About this task**

Both IMS and Db2 handle recovery in an IMS application program that accesses Db2 data. IMS coordinates the process, and Db2 handles recovery for Db2 data.

**Procedure**

To plan for program recovery in IMS programs:

1. For a program that processes messages as its input, decide whether to specify single-mode or multiple-mode transactions on the TRANSACT statement of the APPLCTN macro for the program.

   **Single-mode**
   
   Indicates that a commit point in Db2 occurs each time the program issues a call to retrieve a new message. Specifying single-mode can simplify recovery; if the program abends, you can restart the program from the most recent call for a new message. When IMS restarts the program, the program starts by processing the next message.

   **Multiple-mode**
   
   Indicates that a commit point occurs when the program issues a checkpoint call or when it terminates normally. Those two events are the only times during the program that IMS sends the program's output messages to their destinations. Because fewer commit points are processed in multiple-mode programs than in single-mode programs, multiple-mode programs could perform slightly better than single-mode programs. When a multiple-mode program abends, IMS can restart it only from a checkpoint call. Instead of having only the most recent message to reprocess, a program might have several messages to reprocess. The number of messages to process depends on when the program issued the last checkpoint call.

   Db2 does some processing with single- and multiple-mode programs. When a multiple-mode program issues a call to retrieve a new message, Db2 performs an authorization check and closes all open cursors in the program.

2. Decide whether to issue checkpoint calls (CHKP) and if so, how often to issue them.

   Each call indicates to IMS that the program has reached a sync point and establishes a place in the program from which you can restart the program.

   Consider the following factors when deciding when to use checkpoint calls:

   - How long it takes to back out and recover that unit of work. The program must issue checkpoints frequently enough to make the program easy to back out and recover.
   - How long database resources are locked in Db2 and IMS.
   - For multiple-mode programs: How you want the output messages grouped. Checkpoint calls establish how a multiple-mode program groups its output messages. Programs must issue checkpoints frequently enough to avoid building up too many output messages.
**Restriction:** You cannot use SQL COMMIT and ROLLBACK statements in the Db2 DL/I batch support environment, because IMS coordinates the unit of work.

3. Issue CLOSE CURSOR statements before any checkpoint calls or GU calls to the message queue, not after.

4. After any checkpoint calls, set the value of any special registers that were reset if their values are needed after the checkpoint:

   A CHKP call causes IMS to sign on to Db2 again, which resets the special registers that are shown in the following table.

   **Table 1. Special registers that are reset by a checkpoint call.**

<table>
<thead>
<tr>
<th>Special register</th>
<th>Value to which it is reset after a checkpoint call</th>
</tr>
</thead>
<tbody>
<tr>
<td>CURRENT PACKAGESET</td>
<td>blanks</td>
</tr>
<tr>
<td>CURRENT SERVER</td>
<td>blanks</td>
</tr>
<tr>
<td>CURRENT SQLID</td>
<td>blanks</td>
</tr>
<tr>
<td>CURRENT DEGREE</td>
<td>1</td>
</tr>
</tbody>
</table>

5. After any commit points, reopen the cursors that you want and re-establish positioning

6. Decide whether to specify the WITH HOLD option for any cursors.

   This option determines whether the program retains the position of the cursor in the Db2 database after you issue IMS CHKP calls. You always lose the program database positioning in DL/I after an IMS CHKP call.

   The program database positioning in Db2 is affected according to the following criteria:

   - If you do not specify the WITH HOLD option for a cursor, you lose the position of that cursor.
   - If you specify the WITH HOLD option for a cursor and the application is message-driven, you lose the position of that cursor.
   - If you specify the WITH HOLD option for a cursor and the application is operating in DL/I batch or DL/I BMP, you retain the position of that cursor.

7. Use IMS rollback calls, ROLL and ROLB, to back out Db2 and DL/I changes to the last commit point.

   These options have the following differences:

   **ROLL**
   Specifies that all changes since the last commit point are to be backed out and the program is to be terminated. IMS terminates the program with user abend code U0778 and without a storage dump.

   When you issue a ROLL call, the only option you supply is the call function, ROLL.

   **ROLB**
   Specifies that all changes since the last commit point are to be backed out and control is to be returned to the program so that it can continue processing.

   A ROLB call has the following options:

   - The call function, ROLB
   - The name of the I/O PCB

   How ROLL and ROLB calls effect DL/I changes in a batch environment depends on the IMS system log and back out options that are specified, as shown in the following table.
### Table 2. Effects of ROLL and ROLLB calls on DL/I changes in a batch environment

<table>
<thead>
<tr>
<th>Rollback call</th>
<th>System log option</th>
<th>Backout option</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>ROLL</td>
<td>tape</td>
<td>any</td>
<td>DL/I does not back out updates, and abend U0778 occurs. Db2 backs out updates to the previous checkpoint.</td>
</tr>
<tr>
<td></td>
<td>disk</td>
<td>BKO=NO</td>
<td></td>
</tr>
<tr>
<td></td>
<td>disk</td>
<td>BKO=YES</td>
<td>DL/I backs out updates, and abend U0778 occurs. Db2 backs out updates to the previous checkpoint.</td>
</tr>
<tr>
<td>ROLB</td>
<td>tape</td>
<td>any</td>
<td>DL/I does not back out updates, and an AL status code is returned in the PCB. Db2 backs out updates to the previous checkpoint. The Db2 DL/I support causes the application program to abend when ROLB fails.</td>
</tr>
<tr>
<td></td>
<td>disk</td>
<td>BKO=NO</td>
<td></td>
</tr>
<tr>
<td></td>
<td>disk</td>
<td>BKO=YES</td>
<td>DL/I backs out database updates, and control is passed back to the application program. Db2 backs out updates to the previous checkpoint.</td>
</tr>
</tbody>
</table>

**Restriction:** You cannot specify the address of an I/O area as one of the options on the call; if you do, your program receives an AD status code. However, you must have an I/O PCB for your program. Specify CMPAT=YES on the CMPAT keyword in the PSBGEN statement for your program's PSB.

**Related concepts**
- Checkpoints in IMS programs
Issuing checkpoint calls releases locked resources and establishes a place in the program from which you can restart the program. The decision about whether your program should issue checkpoints (and if so, how often) depends on your program.

**Unit of work in IMS online programs**

IMS applications can explicitly define units of work by using a CHKP, SYNC, ROLL, or ROLB call, or, for single-mode transactions, a GU call.

In IMS, a unit of work starts when one of the following events occurs:

- When the program starts
- After a CHKP, SYNC, ROLL, or ROLB call has completed
- For single-mode transactions, when a GU call is issued to the I/O PCB

A unit of work ends when one of the following events occurs:

- The program issues either a subsequent CHKP or SYNC call, or, for single-mode transactions, a GU call to the I/O PCB. At this point in the processing, the data is consistent. All data changes that were made since the previous commit point are made correctly.
- The program issues a subsequent ROLB or ROLL call. At this point in the processing, your program has determined that the data changes are not correct and, therefore, that the data changes should not become permanent.
- The program terminates.

**Restriction:** The SQL COMMIT and ROLLBACK statements are not valid in an IMS environment.

A commit point occurs in a program as the result of any one of the following events:

- The program terminates normally. Normal program termination is always a commit point.
- The program issues a checkpoint call. Checkpoint calls are a program's means of explicitly indicating to IMS that it has reached a commit point in its processing.
- The program issues a SYNC call. A SYNC call is a Fast Path system service call to request commit-point processing. You can use a SYNC call only in a non-message-driven Fast Path program.
- For a program that processes messages as its input, a commit point can occur when the program retrieves a new message. This behavior depends on the mode that you specify in the APPLCTN macro for the program:
  - If you specify single-mode transactions, a commit point in Db2 occurs each time the program issues a call to retrieve a new message.
  - If you specify multiple-mode transactions or you do not specify a mode, a commit point occurs when the program issues a checkpoint call or when it terminates normally.

At the time of a commit point, the following actions occur:

- IMS and Db2 can release locks that the program has held since the last commit point. Releasing these locks makes the data available to other application programs and users.
- Db2 closes any open cursors that the program has been using.
- IMS and Db2 make the program's changes to the database permanent.
- If the program processes messages, IMS sends the output messages that the application program produces to their final destinations. Until the program reaches a commit point, IMS holds the program's output messages at a temporary destination.

If the program abends before reaching the commit point, the following actions occur:

- Both IMS and Db2 back out all the changes the program has made to the database since the last commit point.
- IMS deletes any output messages that the program has produced since the last commit point (for nonexpress PCBs).
If the program processes messages, people at terminals and other application programs receive information from the terminating application program.

If the system fails, a unit of work resolves automatically when Db2 and IMS batch programs reconnect. Any indoubt units of work are resolved at reconnect time.

**Specifying checkpoint frequency in IMS programs**

A checkpoint indicates a commit point in IMS programs. You should specify checkpoint frequency in your program in a way that allows it to easily be changed, in case the frequency that you initially specify is not appropriate.

**Procedure**

To specify checkpoint frequency in IMS programs:

1. Use a counter in your program to keep track of one of the following items:
   - Elapsed time
   - The number of root segments that your program accesses
   - The number of updates that your program performs
2. Issue a checkpoint call after a certain time interval, number of root segments, or number of updates.

**Checkpoints in IMS programs**

Issuing checkpoint calls releases locked resources and establishes a place in the program from which you can restart the program. The decision about whether your program should issue checkpoints (and if so, how often) depends on your program.

Generally, the following types of programs should issue checkpoint calls:

- Multiple-mode programs
- Batch-oriented BMPs
- Nonmessage-driven Fast Path programs. (These programs can use a special Fast Path call, but they can also use symbolic checkpoint calls.)
- Most batch programs
- Programs that run in a data sharing environment. (Data sharing makes it possible for online and batch application programs in separate IMS systems, in the same or separate processors, to access databases concurrently. Issuing checkpoint calls frequently in programs that run in a data sharing environment is important, because programs in several IMS systems access the database.)

You do not need to issue checkpoints in the following types of programs:

- Single-mode programs
- Database load programs
- Programs that access the database in read-only mode (defined with the processing option GO during a PSBGEN) and are short enough to restart from the beginning
- Programs that, by their nature, must have exclusive use of the database

A CHKP call causes IMS to perform the following actions:

- Inform Db2 that the changes that your program made to the database can become permanent. Db2 makes the changes to Db2 data permanent, and IMS makes the changes to IMS data permanent.
- Send a message that contains the checkpoint identification that is given in the call to the system console operator and to the IMS master terminal operator (MTO).
- Return the next input message to the program's I/O area if the program processes input messages. In MPPs and transaction-oriented BMPs, a checkpoint call acts like a call for a new message.
- Sign on to Db2 again.
Programs that issue symbolic checkpoint calls can specify as many as seven data areas in the program that is to be restored at restart. Db2 always recovers to the last checkpoint. You must restart the program from that point.

If you use symbolic checkpoint calls, you can use a restart call (XRST) to restart a program after an abend. This call restores the program's data areas to the way they were when the program terminated abnormally, and it restarts the program from the last checkpoint call that the program issued before terminating abnormally.

**Restriction:** For BMP programs that process Db2 databases, you can restart the program only from the latest checkpoint and not from any checkpoint, as in IMS.

### Checkpoints in MPPs and transaction-oriented BMPs

In single-mode programs, checkpoint calls and message retrieval calls (called get-unique calls) both establish commit points. The checkpoint calls retrieve input messages and take the place of get-unique calls. BMPs that access non-DL/I databases and MPPs can issue both get unique calls and checkpoint calls to establish commit points. However, message-driven BMPs must issue checkpoint calls rather than get-unique calls to establish commit points, because they can restart from a checkpoint only. If a program abends after issuing a get-unique call, IMS backs out the database updates to the most recent commit point, which is the get-unique call.

In multiple-mode BMPs and MPPs, the only commit points are the checkpoint calls that the program issues and normal program termination. If the program abends and it has not issued checkpoint calls, IMS backs out the program's database updates and cancels the messages that it has created since the beginning of the program. If the program has issued checkpoint calls, IMS backs out the program's changes and cancels the output messages it has created since the most recent checkpoint call.

### Checkpoints in batch-oriented BMPs

If a batch-oriented BMP does not issue checkpoints frequently enough, IMS can abend that BMP or another application program for one of the following reasons:

- **Other programs cannot get to the data that they need within a specified amount of time.**
  
  If a BMP retrieves and updates many database records between checkpoint calls, it can monopolize large portions of the databases and cause long waits for other programs that need those segments. (The exception to this situation is a BMP with a processing option of GO; IMS does not enqueue segments for programs with this processing option.) Issuing checkpoint calls releases the segments that the BMP has enqueued and makes them available to other programs.

- **Not enough storage is available for the segments that the program has read and updated.**
  
  If IMS is using program isolation enqueuing, the space that is needed to enqueue information about the segments that the program has read and updated must not exceed the amount of storage that is defined for the IMS system. (The amount of storage available is specified during IMS system definition.) If a BMP enqueues too many segments, the amount of storage that is needed for the enqueued segments can exceed the amount of available storage. In that case, IMS terminates the program abnormally. You then need to increase the program's checkpoint frequency before rerunning the program.

When you issue a DL/I CHKP call from an application program that uses Db2 databases, IMS processes the CHKP call for all DL/I databases, and Db2 commits all the Db2 database resources. No checkpoint information is recorded for Db2 databases in the IMS log or the Db2 log. The application program must record relevant information about Db2 databases for a checkpoint, if necessary. One way to record such information is to put it in a data area that is included in the DL/I CHKP call.

Performance might be slowed by the commit processing that Db2 does during a DL/I CHKP call, because the program needs to re-establish position within a Db2 database. The fastest way to re-establish a position in a Db2 database is to use an index on the target table, with a key that matches one-to-one with every column in the SQL predicate.
Recovering data in IMS programs

Online IMS systems handle recovery and restart. For a batch region, the operational procedures control recovery and restart for your location.

Procedure

Take one or more of the following actions depending on the type of program:

<table>
<thead>
<tr>
<th>Program type</th>
<th>Recommended action</th>
</tr>
</thead>
<tbody>
<tr>
<td>DL/I batch applications</td>
<td>Use the DL/I batch backout utility to back out DL/I changes. Db2 automatically backs out changes whenever the application program abends.</td>
</tr>
<tr>
<td>Applications that use symbolic checkpoints</td>
<td>Use a restart call (XRST) to restart a program after an abend. This call restores the program's data areas to the way they were when the program terminated abnormally, and it restarts the program from the last checkpoint call that the program issued before terminating abnormally.</td>
</tr>
<tr>
<td>BMP programs that access Db2 databases</td>
<td>Restart the program from the latest checkpoint.</td>
</tr>
<tr>
<td>Restriction: You can restart the program only from the latest checkpoint and not from any checkpoint, as in IMS.</td>
<td></td>
</tr>
<tr>
<td>Applications that use online IMS systems</td>
<td>No action needed. Recovery and restart are part of the IMS system</td>
</tr>
<tr>
<td>Applications that reside in the batch region</td>
<td>Follow your location's operational procedures to control recovery and restart.</td>
</tr>
</tbody>
</table>

Undoing selected changes within a unit of work by using savepoints

Savepoints enable you to undo selected changes within a unit of work. Your application can set any number of savepoints and then specify a specific savepoint to indicate which changes to undo within the unit of work.

Procedure

To undo selected changes within a unit of work by using savepoints:

1. Set any savepoints by using SQL SAVEPOINT statements.
   
   Savepoints set a point to which you can undo changes within a unit of work.

   Consider the following abilities and restrictions when setting savepoints:

   • You can set a savepoint with the same name multiple times within a unit of work. Each time that you set the savepoint, the new value of the savepoint replaces the old value.
   • If you do not want a savepoint to have different values within a unit of work, use the UNIQUE option in the SAVEPOINT statement. If an application executes a SAVEPOINT statement with the same name as a savepoint that was previously defined as unique, an SQL error occurs.
   • If you set a savepoint before you execute a CONNECT statement, the scope of that savepoint is the local site. If you set a savepoint after you execute the CONNECT statement, the scope of that savepoint is the site to which you are connected.
   • When savepoints are active, which they are until the unit of work completes, you cannot access remote sites by using three-part names or aliases for three-part names. You can, however, use DRDA access with explicit CONNECT statements.
   • You cannot use savepoints in global transactions, triggers, user-defined functions, or stored procedures that are nested within triggers or user-defined functions.

2. Specify the changes that you want to undo within a unit of work by using the SQL ROLLBACK TO SAVEPOINT statement.
Db2 undoes all changes since the specified savepoint. If you do not specify a savepoint name, Db2 rolls back work to the most recently created savepoint.

3. Optional: If you no longer need a savepoint, delete it by using the SQL RELEASE SAVEPOINT statement.

**Recommendation:** If you no longer need a savepoint before the end of a transaction, release it. Otherwise, savepoints are automatically released at the end of a unit of work. Releasing savepoints is essential if you need to use three-part names to access remote locations, because you cannot perform this action while savepoints are active.

**Examples**

**Example: Rolling back to the most recently created savepoint**

When the ROLLBACK TO SAVEPOINT statement is executed in the following code, Db2 rolls back work to savepoint B.

```sql
EXEC SQL SAVEPOINT A;

EXEC SQL SAVEPOINT B;

EXEC SQL ROLLBACK TO SAVEPOINT;
```

**Example: Setting savepoints during distributed processing**

An application performs the following tasks:

1. Sets savepoint C1.
2. Does some local processing.
3. Executes a CONNECT statement to connect to a remote site.

Because savepoint C1 is set before the application connects to a remote site, savepoint C1 is known only at the local site. However, because savepoint C2 is set after the application connects to the remote site, savepoint C2 is known only at the remote site.

**Setting multiple savepoints with the same name**

Suppose that the following actions occur within a unit of work:

1. Application A sets savepoint S.
2. Application A calls stored procedure P.
3. Stored procedure P sets savepoint S.
4. Stored procedure P executes the following statement: `ROLLBACK TO SAVEPOINT S`

When Db2 executes the ROLLBACK statement, Db2 rolls back work to the savepoint that was set in the stored procedure, because that value is the most recent value of savepoint S.

**Related reference**

- [RELEASE SAVEPOINT (Db2 SQL)](http://www.ibm.com/support/docview.wss?uid=swg21238851)
- [ROLLBACK (Db2 SQL)](http://www.ibm.com/support/docview.wss?uid=swg21238850)
- [SAVEPOINT (Db2 SQL)](http://www.ibm.com/support/docview.wss?uid=swg21238853)

**Planning for recovery of table spaces that are not logged**

To suppress logging, you can specify the NOT LOGGED option when you create or alter a table space. However, because logs are generally used in recovery, planning for recovery of table spaces for which changes are not logged requires some additional planning.

**About this task**

Although you can plan for recovery, you still need to take some corrective actions after any system failures to recover the data and fix any affected table spaces. For example, if a table space that is not logged was open for update at the time that Db2 terminates, the subsequent restart places that table...
space in LPL and marks it with RECOVER-pending status. You need to take corrective action to clear the RECOVER-pending status.

**Procedure**

To plan for recovery of table spaces that are not logged:

1. Ensure that you can recover lost data by performing one of the following actions:
   - Ensure that you have a data recovery source that does not rely on a log record to re-create any lost data.
   - Limit modifications that are not logged to easily repeatable changes that can be quickly repeated.

2. Avoid placing a table space that is not logged in a RECOVER-pending status.

   The following actions place a table space in RECOVER-pending status:
   - Issuing a ROLLBACK statement or ROLLBACK TO SAVEPOINT statement after modifying a table in a table space that is not logged.
   - Causing duplicate keys or referential integrity violations when you modify a table space that is not logged.

   If the table space is placed in RECOVER-pending status, it is unavailable until you manually fix it.

3. For table spaces that are not logged and have associated LOB or XML table spaces, take image copies as a recovery set.

   This action ensures that the base table space and all the associated LOB or XML table spaces are copied at the same point in time. A subsequent RECOVER TO LASTCOPY operation for the entire set results in consistent data across the base table space and all of the associated LOB and XML table spaces.

**Related tasks**

- Clearing the RECOVER-pending status (Db2 Administration Guide)

**Related reference**

- RECOVER (Db2 Utilities)

---

**Designing your application to access distributed data**

You can design applications that access data on another database management system (DBMS) other than your local system. You should consider the limitations and recommendations for such programs when designing them.

**Procedure**

To design your application to access distributed data:

1. Ensure that the appropriate authorization ID has been granted authorization at the remote server to connect to that server and use resources from it.

2. If your application contains SQL statements that run at the requester, include at the requester a database request module (DBRM) that is bound directly into a package that is included in the plan's package list.

3. Copy the requester package to any remote server that is accessed by the application via a bind package copy command and include the remote packages in the application plan's package list.

   **Recommendation:** Specify an asterisk (*) instead of a specific name in the location name of any package entry of a plan so that the plan does not have to be rebound whenever a new location is accessed by the application or a different location is to be accessed.

4. For TSO and batch applications that update data at a remote server, ensure that one of the following conditions is true:
   - No other connections exist.
• All existing connections are to servers that are restricted to read-only operations.

**Restriction:** If neither of these conditions are met, the application is restricted to read-only operations.

If one of these conditions is met, and if the first connection in a logical unit of work is to a server that supports two-phase commit, that server and all servers that support two-phase commit can update data. However, if the first connection is to a server that does not support two-phase commit, only that server is allowed to update data.

5. For programs that access at least one restricted system, ensure that your program does not violate any of the limitations for accessing restricted systems.

A **restricted system** is a DBMS that does not implement two-phase commit processing.

Accessing restricted systems has the following limitations:

• For programs that access CICS or IMS, you cannot update data on restricted systems.
• Within a unit of work, you cannot update a restricted system after updating a non-restricted system.
• Within a unit of work, if you update a restricted system, you cannot update any other systems.

If you are accessing a mixture of systems, some of which might be restricted, you can perform the following actions:

• Read from any of the systems at any time.
• Update any one system many times in one unit of work.
• Update many systems, including CICS or IMS, in one unit of work, provided that none of them is a restricted system. If the first system you update in a unit of work is not restricted, any attempt to update a restricted system in that unit of work returns an error.
• Update one restricted system in a unit of work, provided that you do not try to update any other system in the same unit of work. If the first system you update in a unit of work is restricted, any attempt to update any other system in that unit of work returns an error.

**Related concepts**
Phase 6: Accessing data at a remote site (Db2 Installation and Migration)

**Related tasks**
Improving performance for applications that access distributed data (Db2 Performance)

**Remote servers and distributed data**

*Distributed data* is data that resides on a database management system (DBMS) other than your local system. Your local DBMS is the one on which you bind your application plan. All other DBMSs are remote.

If you are requesting services from a remote DBMS, that DBMS is a server, and your local system is a requester or client.

Your application can be connected to many DBMSs at one time; the one that is currently performing work is the current server. When the local system is performing work, it also is called the current server.

A remote server can be physically remote, or it can be another subsystem of the same operating system that your local DBMS runs under. A remote server might be an instance of Db2 for z/OS, or it might be an instance of one of another product.

A DBMS, whether local or remote, is known to your Db2 system by its location name. The location name of a remote DBMS is recorded in the communications database.

**Related tasks**
Choosing names for the local subsystem (Db2 Installation and Migration)
Preparing for coordinated updates to two or more data sources

Two or more updates are coordinated if they must all commit or all roll back in the same unit of work.

About this task

This situation is common in banking. Suppose that an amount is subtracted from one account and added to another. The two actions must either both commit or both roll back at the end of the unit of work.

Procedure

Ensure that all systems that your program accesses implement two-phase commit processing. This processing ensures that updates to two or more DBMSs are coordinated automatically.

For example, Db2 and IMS, and Db2 and CICS, jointly implement a two-phase commit process. You can update an IMS database and a Db2 table in the same unit of work. If a system or communication failure occurs between committing the work on IMS and on Db2, the two programs restore the two systems to a consistent point when activity resumes.

You cannot do true coordinated updates within a DBMS that does not implement two-phase commit processing, because Db2 prevents you from updating such a DBMS and any other system within the same unit of work. In this context, update includes the statements INSERT, UPDATE, MERGE, DELETE, CREATE, ALTER, DROP, GRANT, REVOKE, RENAME, COMMENT, and LABEL.

However, if you cannot implement two-phase commit processing on all systems that your program accesses, you can simulate the effect of coordinated updates by performing the following actions:

a. Update one system and commit that work.
b. Update the second system and commit its work.
c. Ensure that your program has code to undo the first update if a failure occurs after the first update is committed and before the second update is committed. No automatic provision exists for bringing the two systems back to a consistent point.

Related concepts

Two-phase commit process (Db2 Administration Guide)

Forcing restricted system rules in your program

A restricted system is a DBMS that does not implement two-phase commit processing. These systems have a number of update restrictions. You can restrict your program completely to the rules for these restricted systems, regardless of whether the program is accessing restricted systems or non-restricted systems.

About this task

Accessing restricted systems has the following limitations:

• For programs that access CICS or IMS, you cannot update data on restricted systems.
• Within a unit of work, you cannot update a restricted system after updating a non-restricted system.
• Within a unit of work, if you update a restricted system, you cannot update any other systems.

Procedure

When you prepare your program, specify the SQL processing option CONNECT(1).

This option applies type 1 CONNECT statement rules.

Restriction: Do not use packages that are precompiled with the CONNECT(1) option and packages that are precompiled with the CONNECT(2) option in the same package list. The first CONNECT statement that is executed by your program determines which rules are in effect for the entire execution: type 1 or type...
2. If your program attempts to execute a later CONNECT statement that is precompiled with the other type, Db2 returns an error.

**Related concepts**

Options for SQL statement processing
Use SQL processing options to specify how the Db2 precompiler and the Db2 coprocessor interpret and process input, and how they present output.
Chapter 2. Connecting to Db2 from your application program

Application programs communicate with Db2 through an attachment facility. You must invoke an attachment facility, either implicitly or explicitly, before your program can interact with Db2.

About this task
You can use the following attachment facilities in a z/OS environment:

**CICS attachment facility**
Use this facility to access Db2 from CICS application programs.

**IMS attachment facility**
Use this facility to access Db2 from IMS application programs.

**Time Sharing Option (TSO) attachment facility**
Use this facility to communicate to a local Db2 subsystem. This facility invokes the DSN command processor.

**Call attachment facility (CAF)**
Use this facility as an alternative to the TSO attachment facility when your application needs tight control over the session environment.

**Resource Recovery Services attachment facility (RRSAF)**
Use this facility for stored procedures that run in a WLM-established address space or as an alternative to the CAF. RRSAF provides support for z/OS RRS as the recovery coordinator and supports other capabilities not present in CAF.

For distributed applications, use the distributed data facility (DDF).

**Requirement:** Ensure that any application that requests Db2 services satisfies the following environment characteristics, regardless of the attachment facility that you use:

- The application must be running in TCB mode. SRB mode is not supported.
- An application task cannot have any Enabled Unlocked Task (EUT) functional recovery routines (FRRs) active when requesting Db2 services. If an EUT FRR is active, the Db2 functional recovery can fail, and your application can receive some unpredictable abends.
- Different attachment facilities cannot be active concurrently within the same address space. Specifically, the following requirements exist:
  - An application must not use CAF or RRSAF in a CICS or IMS address space.
  - An application that runs in an address space that has a CAF connection to Db2 cannot connect to Db2 by using RRSAF.
  - An application that runs in an address space that has an RRSAF connection to Db2 cannot connect to Db2 by using CAF.
  - An application cannot invoke the z/OS AXSET macro after executing the CAF CONNECT call and before executing the CAF DISCONNECT call.
- One attachment facility cannot start another. For example, your CAF or RRSAF application cannot use DSN, and a DSN RUN subcommand cannot call your CAF or RRSAF application.
- The language interface modules for CAF and RRSAF, DSNALI and DSNRLI, are shipped with the linkage attributes AMODE(31) and RMODE(ANY). If your applications load CAF or RRSAF below the 16-MB line, you must link-edit DSNALI or DSNRLI again.

Related concepts
Db2 attachment facilities (Introduction to Db2 for z/OS)
Distributed data facility (Introduction to Db2 for z/OS)
Invoking the call attachment facility

Invoke the call attachment facility (CAF) when you want your application program to establish and control its own connection to Db2. Applications that use CAF can explicitly control the state of their connections to Db2 by using connection functions that CAF supplies.

Before you begin
Before you can invoke CAF, perform the following actions:

- Ensure that the CAF language interface (DSNALI) is available.
- Ensure that your application satisfies the requirements for programs that access CAF.
- Ensure that your application satisfies the general environment characteristics for connecting to Db2.
- Ensure that you are familiar with the following z/OS concepts and facilities:
  - The CALL macro and standard module linkage conventions
  - Program addressing and residency options (AMODE and RMODE)
  - Creating and controlling tasks; multitasking
  - Functional recovery facilities such as ESTAE, ESTAI, and FRRs
  - Asynchronous events and TSO attention exits (STAX)
  - Synchronization techniques such as WAIT/POST.

About this task
Applications that use CAF can be written in assembler language, C, COBOL, Fortran, and PL/I. When choosing a language to code your application in, consider the following restrictions:

- If you need to use z/OS macros (ATTACH, WAIT, POST, and so on), use a programming language that supports them or embed them in modules that are written in assembler language.
- The CAF TRANSLATE function is not available in Fortran. To use this function, code it in a routine that is written in another language, and then call that routine from Fortran.

Recommendations: For IMS and DSN applications, consider the following recommendations:

- For IMS batch applications, do not use CAF. Instead use the Db2 DL/I batch support. Although it is possible for IMS batch applications to access Db2 databases through CAF, that method does not coordinate the commitment of work between the IMS and Db2 systems.
- For DSN applications, do not use CAF unless you provide an application controller to manage the DSN application and replace any needed DSN functions. You might also have to change the application to communicate connection failures to the controller correctly. Running DSN applications with CAF is not advantageous, and the loss of DSN services can affect how well your program runs.

Procedure
Perform one of the following actions:

- Explicitly invoke CAF by including in your program CALL DSNALI statements with the appropriate options.

  The first option is a CAF connection function, which describes the action that you want CAF to take. The effect of any function depends in part on what functions the program has already run.

  Requirement: For C and PL/I applications, you must also include in your program the compiler directives that are listed in the following table, because DSNALI is an assembler language program.
Table 3. Compiler directives to include in C and PL/I applications that contain CALL DSNALI statements

<table>
<thead>
<tr>
<th>Language</th>
<th>Compiler directive to include</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>#pragma linkage(dsnali, OS)</td>
</tr>
<tr>
<td>C++</td>
<td>extern &quot;OS&quot; {</td>
</tr>
<tr>
<td></td>
<td>int DSNALI(char * functn,</td>
</tr>
<tr>
<td></td>
<td>...); }</td>
</tr>
<tr>
<td>PL/I</td>
<td>DCL DSNALI ENTRY OPTIONS(ASM, INTER, RETCODE;</td>
</tr>
</tbody>
</table>

- Implicitly invoke CAF by including SQL statements or IFI calls in your program just as you would in any program. The CAF facility establishes the connections to Db2 with the default values for the subsystem name and plan name.

**Restriction:** If your program can make its first SQL call from different modules with different DBRMs, you cannot use a default plan name and thus, you cannot implicitly invoke CAF. Instead, you must explicitly invoke CAF by using the OPEN function.

**Requirement:** If your application includes both SQL and IFI calls, you must issue at least one SQL call before you issue any IFI calls. This action ensures that your application uses the correct plan.

Although doing so is not recommended, you can run existing DSN applications with CAF by allowing them to make implicit connections to Db2. For Db2 to make an implicit connection successfully, the plan name for the application must be the same as the member name of the database request module (DBRM) that Db2 produced when you precompiled the source program that contains the first SQL call. You must also substitute the DSNALI language interface module for the TSO language interface module, DSNELI.

If you do not specify the return code and reason code parameters in your CAF calls or you invoked CAF implicitly, CAF puts a return code in register 15 and a reason code in register 0.

To determine if an implicit connection was successful, the application program should examine the return and reason codes immediately after the first executable SQL statement in the application program by performing one of the following actions:

- Examining registers 0 and 15 directly.
- Examining the SQLCA, and if the SQLCODE is -991, obtain the return and reason code from the message text. The return code is the first token, and the reason code is the second token.

If the implicit connection was successful, the application can examine the SQLCODE for the first, and subsequent, SQL statements.

**Examples**

**Example of a CAF configuration**

The following figure shows an conceptual example of invoking and using CAF. The application contains statements to load DSNALI, DSNHLI2, and DSNWLI2. The application accesses Db2 by using the CAF Language Interface. It calls DSNALI to handle CAF requests, DSNUWLI to handle IFI calls, and DSNHLI to handle SQL calls.
Sample programs that use CAF
You can find a sample assembler program (DSN8CA) and a sample COBOL program (DSN8CC) that use the CAF in library prefix.SDSSAMP. A PL/I application (DSN8SPM) calls DSN8CA, and a COBOL application (DSN8SCM) calls DSN8CC.

Related concepts
Sample applications supplied with Db2 for z/OS
Db2 provides sample applications to help you with Db2 programming techniques and coding practices within each of the four environments: batch, TSO, IMS, and CICS. The sample applications contain various applications that might apply to managing a company.

Related reference
CAF connection functions
A CAF connection function specifies the action that you want CAF to take. You specify these functions when you invoke CAF through CALL DSNALI statements.

Call attachment facility
An attachment facility enables programs to communicate with Db2. The call attachment facility (CAF) provides such a connection for programs that run in z/OS batch, TSO foreground, and TSO background. The CAF needs tight control over the session environment.

A program that uses CAF can perform the following actions:
- Access Db2 from z/OS address spaces where TSO, IMS, or CICS do not exist.
- Access Db2 from multiple z/OS tasks in an address space.
- Access the Db2 IFI.
- Run when Db2 is down.

Restriction: The application cannot run SQL when Db2 is down.
- Run with or without the TSO terminal monitor program (TMP).
- Run without being a subtask of the DSN command processor or of any Db2 code.
- Run above or below the 16-MB line. (The CAF code resides below the line.)
- Establish an explicit connection to Db2, through a CALL interface, with control over the exact state of the connection.
- Establish an implicit connection to Db2, by using SQL statements or IFI calls without first calling CAF, with a default plan name and subsystem identifier.
- Verify that the application is using the correct release of Db2.
- Supply event control blocks (ECBs), for Db2 to post, that signal startup or termination.
- Intercept return codes, reason codes, and abend codes from Db2 and translate them into messages.

Any task in an address space can establish a connection to Db2 through CAF. Only one connection can exist for each task control block (TCB). A Db2 service request that is issued by a program that is running under a given task is associated with that task’s connection to Db2. The service request operates independently of any Db2 activity under any other task.

Each connected task can run a plan. Multiple tasks in a single address space can specify the same plan, but each instance of a plan runs independently from the others. A task can terminate its plan and run a different plan without fully breaking its connection to Db2.

CAF does not generate task structures.

When you design your application, consider that using multiple simultaneous connections can increase the possibility of deadlocks and Db2 resource contention.

A tracing facility provides diagnostic messages that aid in debugging programs and diagnosing errors in the CAF code. In particular, attempts to use CAF incorrectly cause error messages in the trace stream.

Restriction: CAF does not provide attention processing exits or functional recovery routines. You can provide whatever attention handling and functional recovery your application needs, but you must use ESTAE/ESTAI type recovery routines and not Enabled Unlocked Task (EUT) FRR routines.

Properties of CAF connections
Call attachment facility (CAF) enables programs to communicate with Db2.

The connection that CAF makes with Db2 has the basic properties that are listed in the following table.
### Table 4. Properties of CAF connections

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Connection name</td>
<td>DB2CALL</td>
<td>You can use the DISPLAY THREAD command to list CAF applications that have the connection name DB2CALL.</td>
</tr>
<tr>
<td>Connection type</td>
<td>BATCH</td>
<td>BATCH connections use a single phase commit process that is coordinated by Db2. Application programs can also control when statements are committed by using the SQL COMMIT and ROLLBACK statements.</td>
</tr>
<tr>
<td>Authorization IDs</td>
<td>Authorization IDs that are associated with the address space</td>
<td>Db2 establishes authorization IDs for each task's connection when it processes that connection. For the BATCH connection type, Db2 creates a list of authorization IDs based on the authorization ID that is associated with the address space. This list is the same for every task. A location can provide a Db2 connection authorization exit routine to change the list of IDs.</td>
</tr>
<tr>
<td>Scope</td>
<td>CAF processes connections as if each task is entirely isolated. When a task requests a function, the CAF passes the functions to Db2 and is unaware of the connection status of other tasks in the address space. However, the application program and the Db2 subsystem are aware of the connection status of multiple tasks in an address space.</td>
<td>none</td>
</tr>
</tbody>
</table>

If a connected task terminates normally before the CLOSE function deallocates the plan, Db2 commits any database changes that the thread made since the last commit point. If a connected task abends before the CLOSE function deallocates the plan, Db2 rolls back any database changes since the last commit point. In either case, Db2 deallocates the plan, if necessary, and terminates the task's connection before it allows the task to terminate.

If Db2 abnormally terminates while an application is running, the application is rolled back to the last commit point. If Db2 terminates while processing a commit request, Db2 either commits or rolls back any changes at the next restart. The action taken depends on the state of the commit request when Db2 terminates.

**Related concepts**
- Connection routines and sign-on routines (Managing Security)
Attention exit routines for CAF

An attention exit routine enables you to regain control from Db2 during long-running or erroneous requests. Call attachment facility (CAF) has no attention exit routines, but you can provide your own if necessary.

An attention exit routine works by detaching the TCB that is currently waiting on an SQL or IFI request to complete. After the TCB is detached, Db2 detects the resulting abend and performs termination processing for that task. The termination processing includes any necessary rollback of transactions.

You can provide your own attention exit routines. However, your routine might not get control if you request attention while Db2 code is running, because Db2 uses enabled unlocked task (EUT) functional recovery routines (FRRs).

Recovery routines for CAF

You can use abend recovery routines and functional recovery routines (FRRs) to handle unexpected errors. An abend recovery routine controls what happens when an abend occurs while Db2 has control. A functional recovery routine can obtain information about and recover from program errors.

The CAF has no abend recovery routines, but you can provide your own. Any abend recovery routines that you provide must use tracking indicators to determine if an abend occurred during Db2 processing. If an abend occurs while Db2 has control, the recovery routine can take one of the following actions:

• Allow task termination to complete. Do not try the program again. Db2 detects task termination and terminates the thread with the ABRT parameter. You lose all database changes back to the last sync point or commit point.

  This action is the only action that you can take for abends that are caused by the CANCEL command or by DETACH. You cannot use additional SQL statements. If you attempt to execute another SQL statement from the application program or its recovery routine, you receive a return code of +256 and a reason code of X'00F30083'.

• In an ESTAE routine, issue a CLOSE function call with the ABRT parameter followed by a DISCONNECT function call. The ESTAE exit routine can try again so that you do not need to reinstate the application task.

FRRs must comply with the following requirements and restrictions:

• You can use only enabled unlocked task (EUT) FRRs in your routines that call Db2. The standard z/OS functional recovery routines (FRRs) apply to only code that runs in service request block (SRB) mode, and Db2 does not support calls from SRB mode routines.

• Do not have an EUT FRR active when using CAF, processing SQL requests, or calling IFI. With z/OS, if you have an active EUT FRR, all Db2 requests fail, including the initial CONNECT or OPEN request. The requests fail because Db2 always creates an ARR-type ESTAE, and z/OS does not allow the creation of ARR-type ESTAEs when an FRR is active.

• An EUT FRR cannot retry failing Db2 requests. An EUT FRR retry bypasses ESTAE routines from Db2. The next Db2 request of any type, including a DISCONNECT request, fails with a return code of +256 and a reason code of X'00F30050'.

Making the CAF language interface (DSNALI) available

Before you can invoke the call attachment facility (CAF), you must first make DSNALI available.

About this task

Part of CAF is a Db2 load module, DSNALI, which is also known as the CAF language interface. DSNALI has the alias names DSNHLI2 and DSNWLI2. The module has five entry points: DSNALI, DSNHLI, DSNHLI2, DSNWLI, and DSNWLI2. These entry points serve the following functions:

• Entry point DSNALI handles explicit Db2 connection service requests.
• DSNHLI and DSNHLI2 handle SQL calls. Use DSNHLI if your application program link-edits DSNALI. Use DSNHLI2 if your application program loads DSNALI.

• DSNWLI and DSNWLI2 handle IFI calls. Use DSNWLI if your application program link-edits DSNALI. Use DSNWLI2 if your application program loads DSNALI.

**Procedure**

To make DSNALI available:

1. Decide which of the following methods you want to use to make DSNALI available:
   - Explicitly issuing LOAD requests when your program runs.
     By explicitly loading the DSNALI module, you beneficially isolate the maintenance of your application from future IBM maintenance to the language interface. If the language interface changes, the change will probably not affect your load module.
   - Including the DSNALI module in your load module when you link-edit your program.
     If you do not need explicit calls to DSNALI for CAF functions, link-editing DSNALI into your load module has some advantages. When you include DSNALI during the link-edit, you do not need to code a dummy DSNHLI entry point in your program or specify the precompiler option ATTACH. Module DSNALI contains an entry point for DSNHLI, which is identical to DSNHLI2, and an entry point DSNWLI, which is identical to DSNWLI2.
     A disadvantage to link-editing DSNALI into your load module is that any IBM maintenance to DSNALI requires a new link-edit of your load module.

     Alternatively, if using explicit connections via CALL DSNALI, you can link-edit your program with DSNULI, the Universal Language Interface.

2. Depending on the method that you chose in step 1, perform one of the following actions:
   - **If you want to explicitly issue LOAD requests when your program runs:**
     In your program, issue z/OS LOAD service requests for entry points DSNALI and DSNHLI2. If you use IFI services, you must also load DSNWLI2. The entry point addresses that LOAD returns are saved for later use with the CALL macro. Indicate to Db2 which entry point to use in one of the following two ways:
       - Specify the precompiler option ATTACH(CAF).
         This option causes Db2 to generate calls that specify entry point DSNHLI2.
         **Restriction:** You cannot use this option if your application is written in Fortran.
       - Code a dummy entry point named DSNHLI within your load module.
         If you do not specify the precompiler option ATTACH, the Db2 precompiler generates calls to entry point DSNHLI for each SQL request. The precompiler does not know about and is independent of the different Db2 attachment facilities. When the calls generated by the Db2 precompiler pass control to DSNHLI, your code that corresponds to the dummy entry point must preserve the option list that was passed in R1 and specify the same option list when it calls DSNHLI2.
   - **If you want to include the DSNALI module in your load module when you link-edit your program:**
     Include DSNALI in your load module during a link-edit step. The module must be in a load module library, which is included either in the SYSLIB concatenation or another INCLUDE library that is defined in the linkage editor JCL. Because all language interface modules contain an entry point declaration for DSNHLI, the linkage editor JCL must contain an INCLUDE linkage editor control statement for DSNALI; for example, INCLUDE SYSLIB(DSNALI). By coding these options, you avoid inadvertently picking up the wrong language interface module.

**Related concepts**

LOB file reference variables
In a host application, you can use a file reference variable to insert a LOB or XML value from a file into a Db2 table. You can also use a file reference variable to select a LOB or XML value from a Db2 table into a file.

Examples of invoking CAF
The call attachment facility (CAF) enables programs to communicate with Db2. If you explicitly invoke CAF in your program, you can use the CAF connection functions to control the state of the connection.

“Universal language interface (DSNULI)” on page 113
The universal language interface (DSNULI) subcomponent determines the runtime environment and dynamically loads and branches to the appropriate language interface module.

Related tasks
Link-editing an application with DSNULI
To create a single load module that can be used in more than one attachment environment, you can link-edit your program or stored procedure with the Universal Language Interface module (DSNULI) instead of with one of the environment-specific language interface modules (DSNELI, DSNALI, DSNRLI, DSNCLI, or DFSLI000).

Saving storage when manipulating LOBs by using LOB locators
LOB locators let you manipulate LOB data without retrieving the data from the Db2 table. By using locators, you avoid needing to allocate the large amounts of storage that are needed for host variables to hold LOB data.

Requirements for programs that use CAF
The call attachment facility (CAF) enables programs to communicate with Db2. Before you invoke CAF in your program, ensure that your program satisfies any requirements for using CAF.

When you write programs that use CAF, ensure that they meet the following requirements:

- The program accounts for the size of the CAF code. The CAF code requires about 16 KB of virtual storage per address space and an additional 10 KB for each TCB that uses CAF.
- If your local environment intercepts and replaces the z/OS LOAD SVC that CAF uses, you must ensure that your version of LOAD manages the load list element (LLE) and contents directory entry (CDE) chains like the standard z/OS LOAD macro. CAF uses z/OS SVC LOAD to load two modules as part of the initialization after your first service request. Both modules are loaded into fetch-protected storage that has the job-step protection key.
- If you use CAF from IMS batch, you must write data to only one system in any one unit of work. If you write to both systems within the same unit, a system failure can leave the two databases inconsistent with no possibility of automatic recovery. To end a unit of work in Db2, execute the SQL COMMIT statement. To end a unit of work in IMS, issue the SYNCPOINT command.

You can prepare application programs to run in CAF similar to how you prepare applications to run in other environments, such as CICS, IMS, and TSO. You can prepare a CAF application either in the batch environment or by using the Db2 program preparation process. You can use the program preparation system either through DB2I or through the DSNH CLIST.

Related tasks
Preparing an application to run on Db2 for z/OS
To prepare and run applications that contain embedded static SQL statements or dynamic SQL statements, you must precompile, compile, link-edit, and bind them.

How CAF modifies the content of registers
If you do not specify the return code and reason code parameters in your CAF function calls or if you invoke CAF implicitly, CAF puts a return code in register 15 and a reason code in register 0. The contents of registers 2 through 14 are preserved across calls.

The following table lists the standard calling conventions for registers R1, R13, R14, and R15.
Table 5. Standard usage of registers R1, R13, R14, and R15

<table>
<thead>
<tr>
<th>Register</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>R1</td>
<td>CALL DSNALI parameter list pointer</td>
</tr>
<tr>
<td>R13</td>
<td>Address of caller’s save area</td>
</tr>
<tr>
<td>R14</td>
<td>Caller’s return address</td>
</tr>
<tr>
<td>R15</td>
<td>CAF entry point address</td>
</tr>
</tbody>
</table>

Your CAF program should respect these register conventions.

CAF also supports high-level languages that cannot examine the contents of individual registers.

**Related concepts**

**CALL DSNALI statement parameter list**
The CALL DSNALI statement explicitly invokes CAF. When you include CALL DSNALI statements in your program, you must specify all parameters that come before the return code parameter.

**Implicit connections to CAF**

If the CAF language interface (DSNALI) is available and you do not explicitly specify CALL DSNALI statements in your application, CAF initiates implicit CONNECT and OPEN requests to Db2. These requests are subject to the same Db2 return codes and reason codes as explicitly specified requests.

Implicit connections use the following defaults:

**Subsystem name**
The default name that is specified in the module DSNHDECP. CAF uses the installation default DSNHDECP, unless your own DSNHDECP module is in a library in a STEPLIB statement of a JOBLIB concatenation or in the link list. In a data sharing group, the default subsystem name is the group attachment name.

Implicit connections to CAF always use DSNHDECP as the user-specified application defaults module.

Be certain that you know what the default name is and that it names the specific Db2 subsystem you want to use.

**Plan name**
The member name of the database request module (DBRM) that Db2 produced when you precompiled the source program that contains the first SQL call.

Different types of implicit connections exist. The simplest is for an application to call neither the CONNECT nor OPEN functions. You can also use the CONNECT function only or the OPEN function only. Each of these calls implicitly connects your application to Db2. To terminate an implicit connection, you must use the proper calls.

**Related concepts**

**Summary of CAF behavior**
The effect of any CAF function depends in part on what functions the program has already run. You should plan the CAF function calls that your program makes to avoid any errors and major structural problems in your application.

**CALL DSNALI statement parameter list**
The CALL DSNALI statement explicitly invokes CAF. When you include CALL DSNALI statements in your program, you must specify all parameters that come before the return code parameter.

For CALL DSNALI statements, use a standard z/OS CALL parameter list. Register 1 points to a list of fullword addresses that point to the actual parameters. The last address must contain a 1 in the high-order bit.
In CALL DSNALI statements, you cannot omit any of parameters that come before the return code parameter by coding zeros or blanks. No defaults exist for those parameters for explicit connection requests. Defaults are provided for only implicit connections. All parameters starting with the return code parameter are optional.

When you want to use the default value for a parameter but specify subsequent parameters, code the CALL DSNALI statement as follows:

- For C-language, when you code CALL DSNALI statements in C, you need to specify the address of every required parameter, using the "address of" operator (&), and not the parameter itself. For example, to pass the `startecb` parameter on CONNECT, specify the address of the 4-byte integer (&secb).

```c
char functn[13] = "CONNECT     ";
char ssid[5] = "DB2A";
int tecb = 0;
int secb = 0;
ptr ribptr;
int retcode;
int reascode;
ptr eibptr;
fnret = dsnali(&functn[0], &ssid[0], &tecb, &secb, &ribptr, &retcode, &reascode, NULL, &eibptr);
```

- For other languages except assembler language, code zero for that parameter in the CALL DSNALI statement. For example, suppose that you are coding a CONNECT call in a COBOL program, and you want to specify all parameters except the return code parameter. You can write a statement similar to the following statement:

```cobol
CALL 'DSNALI' USING FUNCTN SSID TECB SECB RIBPTR
BY CONTENT ZERO BY REFERENCE REASCODE SRDURA EIBPTR.
```

- For assembler language, code a comma for that parameter in the CALL DSNALI statement. For example, to specify all optional parameters except the return code parameter write a statement similar to the following statement:

```assembly
CALL DSNALI,(FUNCTN,SSID,TERMECB,STARTECB,RIBPTR,,REASCODE,SRDURA,EIBPTR,
GROUPOVERRIDE)
```

The following figure shows a sample parameter list structure for the CONNECT function.
The preceding figure illustrates how you can omit parameters for the CALL DSNALI statement to control the return code and reason code fields after a CONNECT call. You can terminate the parameter list at any of the following points. These termination points apply to all CALL DSNALI statement parameter lists.

1. Terminates the parameter list without specifying the parameters `retcode`, `reascode` and `srdura` and places the return code in register 15 and the reason code in register 0.

   Terminating the parameter list at this point ensures compatibility with CAF programs that require a return code in register 15 and a reason code in register 0.

2. Terminates the parameter list after the parameter `retcode` and places the return code in the parameter list and the reason code in register 0.

   Terminating the parameter list at this point enables the application program to take action, based on the return code, without further examination of the associated reason code.

3. Terminates the parameter list after the parameter `reascode` and places the return code and the reason code in the parameter list.

   Terminating the parameter list at this point provides support to high-level languages that are unable to examine the contents of individual registers.

   If you code your CAF application in assembler language, you can specify the reason code parameter and omit the return code parameter.

4. Terminates the parameter list after the parameter `srdura`.

   If you code your CAF application in assembler language, you can specify this parameter and omit the `retcode` and `reascode` parameters.
5. Terminates the parameter list after the parameter eibptr.

If you code your CAF application in assembler language, you can specify this parameter and omit the retnum, reascode, or srdura parameters.

6. Terminates the parameter list after the parameter groupoverride.

If you code your CAF application in assembler language, you can specify this parameter and omit the retnum, reascode, srdura, or eibptr parameters.

Even if you specify that the return code be placed in the parameter list, it is also placed in register 15 to accommodate high-level languages that support special return code processing.

**Related concepts**

**How CAF modifies the content of registers**

If you do not specify the return code and reason code parameters in your CAF function calls or if you invoke CAF implicitly, CAF puts a return code in register 15 and a reason code in register 0. The contents of registers 2 through 14 are preserved across calls.

**Summary of CAF behavior**

The effect of any CAF function depends in part on what functions the program has already run. You should plan the CAF function calls that your program makes to avoid any errors and major structural problems in your application.

The following table summarizes CAF behavior after various inputs from application programs. The top row lists the possible CAF functions that programs can call. The first column lists the task's most recent history of connection requests. For example, the value "CONNECT followed by OPEN" in the first column means that the task issued CONNECT and then OPEN with no other CAF calls in between. The intersection of a row and column shows the effect of the next call if it follows the corresponding connection history.

For example, if the call is OPEN and the connection history is CONNECT, the effect is OPEN; the OPEN function is performed. If the call is SQL and the connection history is empty (meaning that the SQL call is the first CAF function the program), the effect is that implicit CONNECT and OPEN functions are performed, followed by the SQL function.

<table>
<thead>
<tr>
<th>Previous function</th>
<th>Next function</th>
<th>CONNECT</th>
<th>OPEN</th>
<th>SQL</th>
<th>CLOSE</th>
<th>DISCONNECT</th>
<th>TRANSLATE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Empty: first call</td>
<td></td>
<td>CONNECT</td>
<td>OPEN</td>
<td>CONNECT, OPEN, followed by the SQL or IFI call</td>
<td>Error 203&lt;sup&gt;1&lt;/sup&gt;</td>
<td>Error 204&lt;sup&gt;1&lt;/sup&gt;</td>
<td>Error 205&lt;sup&gt;1&lt;/sup&gt;</td>
</tr>
<tr>
<td>CONNECT</td>
<td></td>
<td>Error 201&lt;sup&gt;1&lt;/sup&gt;</td>
<td>OPEN</td>
<td>OPEN, followed by the SQL or IFI call</td>
<td>Error 203&lt;sup&gt;1&lt;/sup&gt;</td>
<td>DISCONNECT</td>
<td>TRANSLATE</td>
</tr>
<tr>
<td>CONNECT followed by OPEN</td>
<td></td>
<td>Error 201&lt;sup&gt;1&lt;/sup&gt;</td>
<td>Error 201&lt;sup&gt;1&lt;/sup&gt;</td>
<td>The SQL or IFI call</td>
<td>CLOSE&lt;sup&gt;2&lt;/sup&gt;</td>
<td>DISCONNECT</td>
<td>TRANSLATE</td>
</tr>
<tr>
<td>CONNECT followed by SQL or IFI call</td>
<td></td>
<td>Error 201&lt;sup&gt;1&lt;/sup&gt;</td>
<td>Error 201&lt;sup&gt;1&lt;/sup&gt;</td>
<td>The SQL or IFI call</td>
<td>CLOSE&lt;sup&gt;2&lt;/sup&gt;</td>
<td>DISCONNECT</td>
<td>TRANSLATE</td>
</tr>
<tr>
<td>OPEN</td>
<td></td>
<td>Error 201&lt;sup&gt;1&lt;/sup&gt;</td>
<td>Error 201&lt;sup&gt;1&lt;/sup&gt;</td>
<td>The SQL or IFI call</td>
<td>CLOSE&lt;sup&gt;2&lt;/sup&gt;</td>
<td>Error 204&lt;sup&gt;1&lt;/sup&gt;</td>
<td>TRANSLATE</td>
</tr>
<tr>
<td>SQL or IFI call</td>
<td></td>
<td>Error 201&lt;sup&gt;1&lt;/sup&gt;</td>
<td>Error 201&lt;sup&gt;1&lt;/sup&gt;</td>
<td>The SQL or IFI call</td>
<td>CLOSE&lt;sup&gt;2&lt;/sup&gt;</td>
<td>Error 204&lt;sup&gt;1&lt;/sup&gt;</td>
<td>TRANSLATE&lt;sup&gt;3&lt;/sup&gt;</td>
</tr>
</tbody>
</table>
Notes:
1. An error is shown in this table as Error nnn. The corresponding reason code is X'00C10nnn'. The message number is DSNA{}nnn{}I or DSNA{}nnn{}E.
2. The task and address space connections remain active. If the CLOSE call fails because Db2 was down, the CAF control blocks are reset, the function produces return code 4 and reason code X'00C10824', and CAF is ready for more connection requests when Db2 is up.
3. A TRANSLATE request is accepted, but in this case it is redundant. CAF automatically issues a TRANSLATE request when an SQL or IFI request fails.

Related reference
CAF return codes and reason codes
CAF provides the return codes either to the corresponding parameters that are specified in a CAF function call or, if you choose not to use those parameters, to registers 15 and 0.

CAF connection functions
A CAF connection function specifies the action that you want CAF to take. You specify these functions when you invoke CAF through CALL DSNALI statements.

You can specify the following CAF functions in a CALL DSNALI statement:

CONNECT
Establishes the task (TCB) as a user of the named Db2 subsystem. When the first task within an address space issues a connection request, the address space is also initialized as a user of Db2.

OPEN
Allocates a Db2 plan. You must allocate a plan before Db2 can process SQL statements. If you did not request the CONNECT function, the OPEN function implicitly establishes the task, and optionally the address space, as a user of Db2.

CLOSE
Commits or abnormally terminates any database changes and deallocates the plan. If the OPEN function implicitly requests the CONNECT function, the CLOSE function removes the task, and possibly the address space, as a user of Db2.

DISCONNECT
Removes the task as a user of Db2 and, if this task is the last or only task in the address space with a Db2 connection, terminates the address space connection to Db2.

TRANSLATE
Returns an SQL code and printable text that describe a Db2 hexadecimal error reason code. This information is returned to the SQLCA.

Restriction: You cannot call the TRANSLATE function from the Fortran language.

Recommendation: Because the effect of any CAF function depends on what functions the program has already run, carefully plan the calls that your program makes to these CAF connection functions. Read about the summary of CAF behavior and make these function calls accordingly.

Related concepts
Summary of CAF behavior
The effect of any CAF function depends in part on what functions the program has already run. You should plan the CAF function calls that your program makes to avoid any errors and major structural problems in your application.

CALL DSNALI statement parameter list

Related reference
CAF return codes and reason codes
CAF provides the return codes either to the corresponding parameters that are specified in a CAF function call or, if you choose not to use those parameters, to registers 15 and 0.
The CALL DSNALI statement explicitly invokes CAF. When you include CALL DSNALI statements in your program, you must specify all parameters that come before the return code parameter.

**CONNECT function for CAF**

The CAF CONNECT function initializes a connection to Db2. This function is different than the SQL CONNECT statement that accesses a remote location within Db2.

The CONNECT function establishes the caller’s task as a user of Db2 services. If no other task in the address space currently holds a connection with the specified subsystem, the CONNECT function also initializes the address space for communication to the Db2 address spaces. The CONNECT function establishes the address space's cross memory authorization to Db2 and builds address space control blocks. You can issue a CONNECT request from any or all tasks in the address space, but the address space level is initialized only once when the first task connects.

Using the CONNECT function is optional. If you do not call the CONNECT function, make an implicit connection by calling neither the CONNECT function nor the OPEN function. In a program that does not contain the CONNECT or OPEN functions, when the first SQL statement is executed, the implicit connection is made to the default Db2 subsystem. The default Db2 subsystem name is the subsystem name that is specified by the SSID=xxxx parameter in installation job DSNTIJUA. Job DSNTIJUA assembles the Db2 data-only application defaults module.

If you do not call the CONNECT function, the first request from a task, either an OPEN request or an SQL or IFI call, causes CAF to issue an implicit CONNECT request. If a task is connected implicitly, the connection to Db2 is terminated either when you call the CLOSE function or when the task terminates.

Call the CONNECT function in all of the following situations:

- You need to specify a particular subsystem name (ssnm) other than the default subsystem name.
- You need the value of the CURRENT DEGREE special register to last as long as the connection (srdura).
- You need to monitor the Db2 startup ECB (startecb), the Db2 termination ECB (termecb), or the Db2 release level.
- You plan to have multiple tasks in the address space open and close plans or a single task in the address space open and close plans more than once.

Establishing task and address space level connections involves significant overhead. Using the CONNECT function to establish a task connection explicitly minimizes this overhead by ensuring that the connection to Db2 remains after the CLOSE function deallocates a plan. In this case, the connection terminates only when you use the DISCONNECT function or when the task terminates.

The CONNECT function also enables the caller to learn the following items:

- That the operator has issued a STOP DB2 command. When this event occurs, Db2 posts the termination ECB, termecb. Your application can either wait on or just look at the ECB.
- That Db2 is abnormally terminating. When this event occurs happens, Db2 posts the termination ECB, termecb.
- That Db2 is available again after a connection attempt that failed because Db2 was down. Your application can either wait or look at the startup ECB, startecb. Db2 ignores this ECB if it was active at the time of the CONNECT request.
- The current release level of Db2. To find this information, access the RIBREL field in the release information block (RIB). If RIBREL is ‘999’, the actual version, release, and modification level of Db2 is indicated in the RIBRELX field and its subfields.

**Restriction:** Do not issue CONNECT requests from a TCB that already has an active Db2 connection.

**Recommendation:** Do not mix explicit CONNECT and OPEN requests with implicitly established connections in the same address space. Either explicitly specify which Db2 subsystem you want to use or allow all requests to use the default subsystem.

The following diagram shows the syntax for the CONNECT function.
Parameters point to the following areas:

**function**
A 12-byte area that contains CONNECT followed by five blanks.

**ssnm**
A 4-byte Db2 subsystem name or group attachment or subgroup attachment name (if used in a data sharing group) to which the connection is made.

If ssnm is less than four characters long, pad it on the right with blanks to a length of four characters.

**termecb**
A 4-byte integer representing the application's event control block (ECB) for Db2 termination. Db2 posts this ECB when the operator enters the STOP DB2 command or when Db2 is abnormally terminating. The ECB indicates the type of termination by a POST code, as shown in the following table:

<table>
<thead>
<tr>
<th>POST code</th>
<th>Termination type</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>QUIESCE</td>
</tr>
<tr>
<td>12</td>
<td>FORCE</td>
</tr>
<tr>
<td>16</td>
<td>ABTERM</td>
</tr>
</tbody>
</table>

Before you check termecb in your CAF application program, first check the return code and reason code from the CONNECT call to ensure that the call completed successfully.

**startecb**
A 4-byte integer representing the application's startup ECB. If Db2 has not yet started when the application issues the call, Db2 posts the ECB when it successfully completes its startup processing. Db2 posts at most one startup ECB per address space. The ECB is the one associated with the most recent CONNECT call from that address space. Your application program must examine any nonzero CAF and Db2 reason codes before issuing a WAIT on this ECB.

If ssnm is a group attachment or subgroup attachment name, the first Db2 subsystem that starts on the local z/OS system and matches the specified group attachment name posts the ECB.

**ribptr**
A 4-byte area in which CAF places the address of the release information block (RIB) after the call. You can determine what release level of Db2 you are currently running by examining the RIBREL field. If RIBREL is ‘999’, the actual version, release, and modification level of Db2 is indicated in the RIBRELX field and its subfields. You can determine the modification level within the release level by examining the RIBCNUMB and RIBCINFO fields. If the value in the RIBCNUMB field is greater than zero, check the RIBCINFO field for modification levels.
If the RIB is not available (for example, if you name a subsystem that does not exist), Db2 sets the 4-byte area to zeros.

The area to which ribptr points is below the 16-MB line.

Your program does not have to use the release information block, but it cannot omit the ribptr parameter.

Macro DSNDRIB maps the release information block (RIB). It can be found in prefix.SDSNMACS(DSNDRIB).

**retcode**

A 4-byte area in which CAF places the return code.

This field is optional. If you do not specify retcode, CAF places the return code in register 15 and the reason code in register 0.

**reascode**

A 4-byte area in which CAF places a reason code.

This field is optional. If you do not specify reascode, CAF places the reason code in register 0. If you specify reascode, you must also specify retcode.

**srdura**

A 10-byte area that contains the string ‘SRDURA(CD)’. This field is optional. If you specify srdura, the value in the CURRENT DEGREE special register stays in effect from the time of the CONNECT call until the time of the DISCONNECT call. If you do not specify srdura, the value in the CURRENT DEGREE special register stays in effect from the time of the OPEN call until the time of the CLOSE call. If you specify this parameter in any language except assembler, you must also specify retcode and reascode. In assembler language, you can omit these parameters by specifying commas as placeholders.

The area to which eibptr points is above the 16-MB line.

You can omit this parameter when you make a CONNECT call.

If you specify this parameter in any language except assembler, you must also specify retcode, reascode, and srdura. In assembler language, you can omit retcode, reascode, and srdura by specifying commas as placeholders.

Macro DSNDEIB maps the EIB. It can be found in prefix.SDSNMACS(DSNDEIB).

**groupoverride**

An 8-byte area that the application provides. This parameter is optional. If you do not want group attach to be attempted, specify ‘NOGROUP’. This string indicates that the subsystem name that is specified by ssnm is to be used as a Db2 subsystem name, even if ssnm matches a group attachment or subgroup attachment name. If groupoverride is not provided, ssnm is used as the group attachment or subgroup attachment name if it matches a group attachment or subgroup attachment name.

If you specify this parameter in any language except assembler, you must also specify retcode, reascode, srdura, and eibptr. In assembler language, you can omit retcode, reascode, srdura, and eibptr by specifying commas as placeholders.

**Recommendation:** Avoid using the groupoverride parameter when possible, because it limits the ability to do dynamic workload routing in a Parallel Sysplex®. However, you should use this parameter in a data sharing environment when you want to connect to a specific member of a data sharing group, and the subsystem name of that member is the same as the group attachment or subgroup attachment name.
**decpptr**
A 4-byte area in which CAF is to put the address of the DSNHDECP control block or user-specified application defaults module that was loaded by subsystem *ssnm* when that subsystem was started. This 4-byte area is a 31-bit pointer. If *ssnm* is not found, the 4-byte area is set to 0.

The area to which decpptr points may be above the 16-MB line.

If you specify this parameter in any language except assembler, you must also specify the retcode, reascode, srdura, eibptr, and groupoverride parameters. In assembler language, you can omit the retcode, reascode, srdura, eibptr, and groupoverride parameters by specifying commas as placeholders.

**Example of CAF CONNECT function calls**
The following table shows a CONNECT call in each language.

<table>
<thead>
<tr>
<th>Table 8. Examples of CAF CONNECT function calls</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Language</strong></td>
</tr>
<tr>
<td>Assembler</td>
</tr>
<tr>
<td>C</td>
</tr>
<tr>
<td>COBOL</td>
</tr>
<tr>
<td>Fortran</td>
</tr>
<tr>
<td>PL/I</td>
</tr>
</tbody>
</table>

**Note:**
- For C and PL/I applications, you must include the appropriate compiler directives, because DSNALI is an assembler language program. These compiler directives are described in the instructions for invoking CAF.

**Related concepts**
Examples of invoking CAF
The call attachment facility (CAF) enables programs to communicate with Db2. If you explicitly invoke CAF in your program, you can use the CAF connection functions to control the state of the connection.

**Related tasks**
Invoking the call attachment facility
Invoke the call attachment facility (CAF) when you want your application program to establish and control its own connection to Db2. Applications that use CAF can explicitly control the state of their connections to Db2 by using connection functions that CAF supplies.

**Related reference**
Synchronizing Tasks (WAIT, POST, and EVENTS Macros) (MVS Programming: Assembler Services Guide)
OPEN function for CAF

The OPEN function allocates Db2 resources that are needed to run the specified plan or to issue IFI requests. If the requesting task does not already have a connection to the named Db2 subsystem, the OPEN function establishes it.

Using the OPEN function is optional. If you do not call the OPEN function, the actions that the OPEN function perform occur implicitly on the first SQL or IFI call from the task.

Restriction: Do not use the OPEN function if the task already has a plan allocated.

The following diagram shows the syntax for the OPEN function.

```
CALL DSNALI (function, ssnm, plan, retcode, reascode, groupoverride)
```

Parameters point to the following areas:

**function**
A 12-byte area that contains the word OPEN followed by eight blanks.

**ssnm**
A 4-byte Db2 subsystem name or group attachment or subgroup attachment name (if used in a data sharing group). The OPEN function allocates the specified plan to this Db2 subsystem. Also, if the requesting task does not already have a connection to the named Db2 subsystem, the OPEN function establishes it.

You must specify the ssnm parameter, even if the requesting task also issues a CONNECT call. If a task issues a CONNECT call followed by an OPEN call, the subsystem names for both calls must be the same.

If ssnm is less than four characters long, pad it on the right with blanks to a length of four characters.

**plan**
An 8-byte Db2 plan name.

**retcode**
A 4-byte area in which CAF places the return code.

This field is optional. If you do not specify retcode, CAF places the return code in register 15 and the reason code in register 0.

**reascode**
A 4-byte area in which CAF places a reason code.

This field is optional. If you do not specify reascode, CAF places the reason code in register 0. If you specify reascode, you must also specify retcode.

**groupoverride**
An 8-byte area that the application provides. This field is optional. If you do not want group attach to be attempted, specify 'NOGROUP'. This string indicates that the subsystem name that is specified by ssnm is to be used as a Db2 subsystem name, even if ssnm matches a group attachment or subgroup attachment name. If you do not specify groupoverride, ssnm is used as the group attachment and subgroup attachment name if it matches a group attachment or subgroup attachment name. If you specify this parameter in any language except assembler, you must also specify retcode and reascode. In assembler language, you can omit these parameters by specifying commas as placeholders.
**Recommendation:** Avoid using the `groupoverride` parameter when possible, because it limits the ability to do dynamic workload routing in a Parallel Sysplex. However, you should use this parameter in a data sharing environment when you want to connect to a specific member of a data sharing group, and the subsystem name of that member is the same as the group attachment or subgroup attachment name.

**Examples of CAF OPEN calls**

The following table shows an OPEN call in each language.

<table>
<thead>
<tr>
<th>Language</th>
<th>Call example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Assembler</td>
<td>CALL DSNALI,(FUNCTN,SSID,PLANNAME, RETCODE,REASCODE,GRPOVER)</td>
</tr>
<tr>
<td>C</td>
<td>fnret=dsnali(&amp;functn[0],&amp;ssid[0], &amp;planname[0],&amp;retcode, &amp;reascode,&amp;grpover[0]);</td>
</tr>
<tr>
<td>COBOL</td>
<td>CALL 'DSNALI' USING FUNCTN SSID PLANNAME RETCODE REASCODE GRPOVER.</td>
</tr>
<tr>
<td>Fortran</td>
<td>CALL DSNALI(FUNCTN,SSID,PLANNAME, RETCODE,REASCODE,GRPOVER)</td>
</tr>
<tr>
<td>PL/I</td>
<td>CALL DSNALI(FUNCTN,SSID,PLANNAME, RETCODE,REASCODE,GRPOVER);</td>
</tr>
</tbody>
</table>

**Note:**
- For C and PL/I applications, you must include the appropriate compiler directives, because DSNALI is an assembler language program. These compiler directives are described in the instructions for invoking CAF.

**Related concepts**

Implicit connections to CAF

If the CAF language interface (DSNALI) is available and you do not explicitly specify CALL DSNALI statements in your application, CAF initiates implicit CONNECT and OPEN requests to Db2. These requests are subject to the same Db2 return codes and reason codes as explicitly specified requests.

**Related tasks**

Invoking the call attachment facility

Invoke the call attachment facility (CAF) when you want your application program to establish and control its own connection to Db2. Applications that use CAF can explicitly control the state of their connections to Db2 by using connection functions that CAF supplies.

**CLOSE function for CAF**

The CAF CLOSE function deallocates the plan that was created either explicitly by a call to the OPEN function or implicitly at the first SQL call. Optionally, the CLOSE function also disconnects the task, and possibly the address space, from Db2.

If you did not issue an explicit CONNECT call for the task, the CLOSE function deletes the task's connection to Db2. If no other task in the address space has an active connection to Db2, Db2 also deletes the control block structures that were created for the address space and removes the cross memory authorization.

Using the CLOSE function is optional. Consider the following rules and recommendations about when to use and not use the CLOSE function:
- Do not use the CLOSE function when your current task does not have a plan allocated.
• If you want to use a new plan, you must issue an explicit CLOSE call, followed by an OPEN call with the new plan name.

• When shutting down your application you can improve the performance of this shut down by explicitly calling the CLOSE function before the task terminates. If you omit the CLOSE call, Db2 performs an implicit CLOSE. In this case, Db2 performs the same actions when your task terminates, by using the SYNC parameter if termination is normal and the ABRT parameter if termination is abnormal.

• If Db2 terminates, issue an explicit CLOSE call for any task that did not issue a CONNECT call. This action enables CAF to reset its control blocks to allow for future connections. This CLOSE call returns the reset accomplished return code (+004) and reason code X'00C10824'. If you omit the CLOSE call in this case, when Db2 is back on line, the task's next connection request fails. You get either the message YOUR TCB DOES NOT HAVE A CONNECTION, with X'00F30018' in register 0, or the CAF error message DSNA201I or DSNA202I, depending on what your application tried to do. The task must then issue a CLOSE call before it can reconnect to Db2.

• A task that issued an explicit CONNECT call should issue a DISCONNECT call instead of a CLOSE call. This action causes CAF to reset its control blocks when Db2 terminates.

The following diagram shows the syntax for the CLOSE function.

```
DSNALI CLOSE function
CALL DSNALI, (function, termop, retcode, reascode)
```

Parameters point to the following areas:

**function**
A 12-byte area that contains the word CLOSE followed by seven blanks.

**termop**
A 4-byte terminate option, with one of the following values:

- **SYNC**
  Specifies that Db2 is to commit any modified data.

- **ABRT**
  Specifies that Db2 is to roll back data to the previous commit point.

**retcode**
A 4-byte area in which CAF is to place the return code.

This field is optional. If you do not specify retcode, CAF places the return code in register 15 and the reason code in register 0.

**reascode**
A 4-byte area in which CAF places a reason code.

This field is optional. If you do not specify reascode, CAF places the reason code in register 0. If you specify reascode, you must also specify retcode.

**Examples of CAF CLOSE calls**
The following table shows a CLOSE call in each language.

<table>
<thead>
<tr>
<th>Language</th>
<th>Call example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Assembler</td>
<td>CALL DSNALI, (FUNCTN, TERMOP, RETCODE, REASCODE)</td>
</tr>
</tbody>
</table>

Table 10. Examples of CAF CLOSE calls
Table 10. Examples of CAF CLOSE calls (continued)

<table>
<thead>
<tr>
<th>Language</th>
<th>Call example</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>fnret=dsnali(&amp;functn[0], &amp;termop[0], &amp;retcode,&amp;reascode);</td>
</tr>
<tr>
<td>COBOL</td>
<td>CALL  'DSNALI' USING FUNCTN TERMOP RETCODE REASCODE.</td>
</tr>
<tr>
<td>Fortran</td>
<td>CALL  DSNALI(FUNCTN,TERMOP, RETCODE,REASCODE)</td>
</tr>
<tr>
<td>PL/I</td>
<td>CALL  DSNALI(FUNCTN,TERMOP, RETCODE,REASCODE);</td>
</tr>
</tbody>
</table>

**Note:**
- For C and PL/I applications, you must include the appropriate compiler directives, because DSNALI is an assembler language program. These compiler directives are described in the instructions for invoking CAF.

**Related tasks**
- **Invoking the call attachment facility**
  Invoke the call attachment facility (CAF) when you want your application program to establish and control its own connection to Db2. Applications that use CAF can explicitly control the state of their connections to Db2 by using connection functions that CAF supplies.

**DISCONNECT function for CAF**

The CAF DISCONNECT function terminates a connection to Db2.

DISCONNECT removes the calling task’s connection to Db2. If no other task in the address space has an active connection to Db2, Db2 also deletes the control block structures that were created for the address space and removes the cross memory authorization.

If an OPEN call is in effect, which means that a plan is allocated, when the DISCONNECT call is issued, CAF issues an implicit CLOSE with the SYNC parameter.

Using the DISCONNECT function is optional. Consider the following rules and recommendations about when to use and not use the DISCONNECT function:

- Only those tasks that explicitly issued a CONNECT call can issue a DISCONNECT call. If a CONNECT call was not used, a DISCONNECT call causes an error.
- When shutting down your application you can improve the performance of this shut down by explicitly calling the DISCONNECT function before the task terminates. If you omit the DISCONNECT call, Db2 performs an implicit DISCONNECT. In this case, Db2 performs the same actions when your task terminates.
- If Db2 terminates, any task that issued a CONNECT call must issue a DISCONNECT call to reset the CAF control blocks. The DISCONNECT function returns the reset accomplished return codes and reason codes (+004 and X'00C10824'). This action ensures that future connection requests from the task work when Db2 is back on line.
- A task that did not explicitly issue a CONNECT call must issue a CLOSE call instead of a DISCONNECT call. This action resets the CAF control blocks when Db2 terminates.

The following diagram shows the syntax for the DISCONNECT function.
The single parameter points to the following area:

**function**
A 12-byte area that contains the word DISCONNECT followed by two blanks.

**retcode**
A 4-byte area in which CAF places the return code.
This field is optional. If you do not specify retcode, CAF places the return code in register 15 and the reason code in register 0.

**reascode**
A 4-byte area in which CAF places a reason code.
This field is optional. If you do not specify reascode, CAF places the reason code in register 0. If you specify reascode, you must also specify retcode.

### Examples of CAF DISCONNECT calls
The following table shows a DISCONNECT call in each language.

<table>
<thead>
<tr>
<th>Language</th>
<th>Call example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Assembler</td>
<td>CALL DSNALI(FUNCTN, RETCODE, REASCODE)</td>
</tr>
<tr>
<td>C(^1)</td>
<td>fnret=dsnali(&amp;functn[0], &amp;retcode, &amp;reascode);</td>
</tr>
<tr>
<td>COBOL</td>
<td>CALL 'DSNALI' USING FUNCTN RETCODE REASCODE.</td>
</tr>
<tr>
<td>Fortran</td>
<td>CALL DSNALI(FUNCTN, RETCODE, REASCODE)</td>
</tr>
<tr>
<td>PL/I(^1)</td>
<td>CALL DSNALI(FUNCTN, RETCODE, REASCODE);</td>
</tr>
</tbody>
</table>

**Note:**
- For C and PL/I applications, you must include the appropriate compiler directives, because DSNALI is an assembler language program. These compiler directives are described in the instructions for invoking CAF.

**Related tasks**
- [Invoking the call attachment facility](#)
Invoke the call attachment facility (CAF) when you want your application program to establish and control its own connection to Db2. Applications that use CAF can explicitly control the state of their connections to Db2 by using connection functions that CAF supplies.

**TRANSLATE function for CAF**

The TRANSLATE function converts a Db2 hexadecimal error reason code from a failed OPEN request into an SQL error code and printable error message text. Db2 places the information into the SQLCODE and SQLSTATE host variables or related fields of the SQLCA of the caller.

The Db2 error reason code that is converted is read from register 0. The TRANSLATE function does not change the contents of registers 0 and 15, unless the TRANSLATE request fails; in that case, register 0 is set to X'C10205' and register 15 is set to 200.

Consider the following rules and recommendations about when to use and not use the TRANSLATE function:

- You cannot call the TRANSLATE function from the Fortran language.
- The TRANSLATE function is useful only if you used an explicit CONNECT call before an OPEN request that fails. For errors that occur during SQL or IFI requests, the TRANSLATE function performs automatically.
- The TRANSLATE function can translate those codes that begin with X'00F3', but it does not translate CAF reason codes that begin with X'00C1'.

If you receive error reason code X'00F30040' (resource unavailable) after an OPEN request, the TRANSLATE function returns the name of the unavailable database object in the last 44 characters of the SQLERRM field.

If the TRANSLATE function does not recognize the error reason code, it returns SQLCODE -924 (SQLSTATE '58006') and places a printable copy of the original Db2 function code and the return and error reason codes in the SQLERRM field.

The following diagram shows the syntax for the TRANSLATE function.

```
DSNALI TRANSLATE function
CALL DSNALI (function, sqlca, retcode, reascode)
```

Parameters point to the following areas:

**function**
A 12-byte area that contains the word TRANSLATE followed by three blanks.

**sqlca**
The program's SQL communication area (SQLCA).

**retcode**
A 4-byte area in which CAF places the return code.

This field is optional. If you do not specify retcode, CAF places the return code in register 15 and the reason code in register 0.

**reascode**
A 4-byte area in which CAF places a reason code.

This field is optional. If you do not specify reascode, CAF places the reason code in register 0. If you specify reascode, you must also specify retcode.

**Examples of CAF TRANSLATE calls**
The following table shows a TRANSLATE call in each language.
### Table 12. Examples of CAF TRANSLATE calls

<table>
<thead>
<tr>
<th>Language</th>
<th>Call example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Assembler</td>
<td>CALL DSNALI,(FUNCTN,SQLCA,RETCODE, REASCODE)</td>
</tr>
<tr>
<td>C¹</td>
<td>fnret=dsnali(&amp;functn[0], &amp;sqlca, &amp;retcode, &amp;reascode);</td>
</tr>
<tr>
<td>COBOL</td>
<td>CALL ‘DSNALI’ USING FUNCTN SQLCA RETCODE REASCODE.</td>
</tr>
<tr>
<td>PL/I¹</td>
<td>CALL DSNALI(FUNCTN,SQLCA,RETCODE, REASCODE);</td>
</tr>
</tbody>
</table>

**Note:**
- For C and PL/I applications, you must include the appropriate compiler directives, because DSNALI is an assembler language program. These compiler directives are described in the instructions for invoking CAF.

### Related tasks
- **Invoking the call attachment facility**
  - Invoke the call attachment facility (CAF) when you want your application program to establish and control its own connection to Db2. Applications that use CAF can explicitly control the state of their connections to Db2 by using connection functions that CAF supplies.

### Turning on a CAF trace
CAF does not capture any diagnostic trace messages unless you tell it to by turning on a trace.

**Procedure**
- Allocate a DSNTRACE data set either dynamically or by including a DSNTRACE DD statement in your JCL.
- CAF writes diagnostic trace messages to that data set. The trace message numbers contain the last three digits of the reason codes.

### Related concepts
- **Examples of invoking CAF**
  - The call attachment facility (CAF) enables programs to communicate with Db2. If you explicitly invoke CAF in your program, you can use the CAF connection functions to control the state of the connection.

### CAF return codes and reason codes
CAF provides the return codes either to the corresponding parameters that are specified in a CAF function call or, if you choose not to use those parameters, to registers 15 and 0.

When the reason code begins with X'00F3' except for X'00F30006', you can use the CAF TRANSLATE function to obtain error message text that can be printed and displayed. These reason codes are issued by the subsystem support for allied memories, a part of the Db2 subsystem support subcomponent that services all Db2 connection and work requests.

For SQL calls, CAF returns standard SQL codes in the SQLCA. CAF returns IFI return codes and reason codes in the instrumentation facility communication area (IFCA).

The following table lists the CAF return codes and reason codes.

### Table 13. CAF return codes and reason codes

<table>
<thead>
<tr>
<th>Return code</th>
<th>Reason code</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>X'000000000'</td>
<td>Successful completion.</td>
</tr>
</tbody>
</table>
Table 13. CAF return codes and reason codes (continued)

<table>
<thead>
<tr>
<th>Return code</th>
<th>Reason code</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>X'00C10824'</td>
<td>CAF reset complete. CAF is ready to make a new connection.</td>
</tr>
<tr>
<td>8</td>
<td>X'00C10831'</td>
<td>Release level mismatch between Db2 and the CAF code.</td>
</tr>
<tr>
<td>200¹</td>
<td>X'00C10201'</td>
<td>Received a second CONNECT request from the same TCB. The first CONNECT request could have been implicit or explicit.</td>
</tr>
<tr>
<td>200¹</td>
<td>X'00C10202'</td>
<td>Received a second OPEN request from the same TCB. The first OPEN request could have been implicit or explicit.</td>
</tr>
<tr>
<td>200¹</td>
<td>X'00C10203'</td>
<td>CLOSE request issued when no active OPEN request exists.</td>
</tr>
<tr>
<td>200¹</td>
<td>X'00C10204'</td>
<td>DISCONNECT request issued when no active CONNECT request exists, or the AXSET macro was issued between the CONNECT request and the DISCONNECT request.</td>
</tr>
<tr>
<td>200¹</td>
<td>X'00C10205'</td>
<td>TRANSLATE request issued when no connection to Db2 exists.</td>
</tr>
<tr>
<td>200¹</td>
<td>X'00C10206'</td>
<td>Incorrect number of parameters was specified or the end-of-list bit was off.</td>
</tr>
<tr>
<td>200¹</td>
<td>X'00C10207'</td>
<td>Unrecognized function parameter.</td>
</tr>
<tr>
<td>200¹</td>
<td>X'00C10208'</td>
<td>Received requests to access two different Db2 subsystems from the same TCB.</td>
</tr>
<tr>
<td>204</td>
<td>²</td>
<td>CAF system error. Probable error in the attach or Db2.</td>
</tr>
</tbody>
</table>

Notes:

1. A CAF error probably caused by errors in the parameter lists from the application programs. CAF errors do not change the current state of your connection to Db2; you can continue processing with a corrected request.
2. System errors cause abends. If tracing is on, a descriptive message is written to the DSNTRACE data set just before the abend.

Sample CAF scenarios

One or more tasks can use call attachment facility (CAF) to connect to Db2. This connection can be made either implicitly or explicitly. For explicit connections, a task calls one or more of the CAF connection functions.

A single task with implicit connections

The simplest connection scenario is a single task that makes calls to Db2 without using explicit CALL DSNALI statements. The task implicitly connects to the default subsystem name and uses the default plan name.

When the task terminates, the following events occur:

- If termination was normal, any database changes are committed.
- If termination was abnormal, any database changes are rolled back.
- The active plan and all database resources are deallocated.
- The task and address space connections to Db2 are terminated.

A single task with explicit connections

The following example pseudocode illustrates a more complex scenario with a single task.
CONNECT
OPEN allocate a plan
SQL or IFI call
...
CLOSE deallocate the current plan
OPEN allocate a new plan
SQL or IFI call
...
CLOSE
DISCONNECT

A task can have a connection to only one Db2 subsystem at any point in time. A CAF error occurs if the subsystem name in the OPEN call does not match the subsystem name in the CONNECT call. To switch to a different subsystem, the application must first disconnect from the current subsystem and then issue a connect request with a new subsystem name.

Multiple tasks

In the following scenario, multiple tasks within the address space use Db2 services. Each task must explicitly specify the same subsystem name on either the CONNECT function request or the OPEN function request. Task 1 makes no SQL or IFI calls. Its purpose is to monitor the Db2 termination and startup ECBs and to check the Db2 release level.

<table>
<thead>
<tr>
<th>TASK 1</th>
<th>TASK 2</th>
<th>TASK 3</th>
<th>TASK n</th>
</tr>
</thead>
<tbody>
<tr>
<td>CONNECT</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>OPEN</td>
<td>OPEN</td>
<td>OPEN</td>
<td></td>
</tr>
<tr>
<td>SQL</td>
<td>SQL</td>
<td>SQL</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td></td>
</tr>
<tr>
<td>CLOSE</td>
<td>CLOSE</td>
<td>CLOSE</td>
<td></td>
</tr>
<tr>
<td>OPEN</td>
<td>OPEN</td>
<td>OPEN</td>
<td></td>
</tr>
<tr>
<td>SQL</td>
<td>SQL</td>
<td>SQL</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td></td>
</tr>
<tr>
<td>CLOSE</td>
<td>CLOSE</td>
<td>CLOSE</td>
<td></td>
</tr>
<tr>
<td>DISCONNECT</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Examples of invoking CAF

The call attachment facility (CAF) enables programs to communicate with Db2. If you explicitly invoke CAF in your program, you can use the CAF connection functions to control the state of the connection.

Example JCL for invoking CAF

The following sample JCL shows how to use CAF in a batch (non-TSO) environment. The DSNTRACE statement in this example is optional.

```plaintext
//jobname JOB z/OS_jobcard_information
//CAFJCL EXEC PGM=CAF_application_program
//STEPLIB DD DSN=application_load_library
// DD DSN=DB2_load_library
:
//SYSPRINT DD SYSOUT=*
//DSNTRACE DD SYSOUT=*
//SYSUDUMP DD SYSOUT=* 
```

Example of assembler code that invokes CAF

The following examples show parts of a sample assembler program that uses CAF. They demonstrate the basic techniques for making CAF calls, but do not show the code and z/OS macros needed to support those calls. For example, many applications need a two-task structure so that attention-handling routines can detach connected subtasks to regain control from Db2. This structure is not shown in the following code examples. Also, these code examples assume the existence of a WRITE macro. Wherever this macro is included in the example, substitute code of your own. You must decide what you want your application to do in those situations; you probably do not want to write the error messages shown.
Example of loading and deleting the CAF language interface

The following code segment shows how an application can load entry points DSNALI and DSNHLI2 for the CAF language interface. Storing the entry points in variables LIALI and LISQL ensures that the application has to load the entry points only once. When the module is done with Db2, you should delete the entries.

```assembly
* ****************************** GET LANGUAGE INTERFACE ENTRY ADDRESSES
* LOAD EP=DSNALI          Load the CAF service request EP
* ST R0,LIALI             Save this for CAF service requests
* LOAD EP=DSNHLI2         Load the CAF SQL call Entry Point
* ST R0,LISQL             Save this for SQL calls
* DELETE EP=DSNALI        Correctly maintain use count
* DELETE EP=DSNHLI2       Correctly maintain use count
```

Example of connecting to Db2 with CAF

The following example code shows how to issue explicit requests for certain actions, such as CONNECT, OPEN, CLOSE, DISCONNECT, and TRANSLATE, and uses the CHEKCODE subroutine to check the return reason codes from CAF.

```assembly
* *************** CONNECT ********************
L R15,LIALI           Get the Language Interface address
MVC FUNCTN,CONNECT    Get the function to call
CALL (15),(FUNCTN,SSID,TECB,SECB,RIBPTR),VL,MF=(E,CAFCALL)
BAL R14,CHEKCODE      Check the return and reason codes
BNE EXIT              If CONTROL not 'CONTINUE', shut down

BE USERLX             If RIBREL = '999', use RIBRELX
WRITE 'The current DB2 release level is' RIBREL
B OPEN                 Continue with signon

*************** OPEN ********************
OPEN L R15,LIALI       Get the Language Interface address
MVC FUNCTN,OPEN        Get the function to call
CALL (15),(FUNCTN,SSID,PLAN),VL,MF=(E,CAFCALL)
BAL R14,CHEKCODE       Check the return and reason codes

*************** SQL **********************
* Insert your SQL calls here. The DB2 Precompiler
generates calls to entry point DSNHLI. You should
* specify the precompiler option ATTACH(CAF), or code
* a dummy entry point named DSNHLI to intercept
* all SQL calls. A dummy DSNHLI is shown below.

*************** CLOSE *******************
CLC CONTROL,CONTINUE   Is everything still OK?
BNE EXIT               If CONTROL not 'CONTINUE', shut down
MVC TRMOP,ABRT         Assume termination with ABRT parameter
L R4,SQLCODE           Put the SQLCODE into a register
C R4,CODE0             Examine the SQLCODE
BZ SYNTAXTERM          If zero, then CLOSE with SYNC parameter
C R4,CODE100           See if SQLCODE was 100
BNE DISC               If not 100, CLOSE with ABRT parameter

SYNTAXTERM MVC TRMOP,SYNC Good code, terminate with SYNC parameter
DISC DS OH             Now build the CAF parmlist
L R15,LIALI            Get the Language Interface address
MVC FUNCTN,CLOSE       Get the function to call
CALL (15),(FUNCTN,TRMOP),VL,MF=(E,CAFCALL)
BAL R14,CHEKCODE       Check the return and reason codes

*************** DISCONNECT ******************
CLC CONTROL,CONTINUE   Is everything still OK?
BNE EXIT               If CONTROL not 'CONTINUE', stop loop
L R15,LIALI            Get the Language Interface address
MVC FUNCTN,DISCON      Get the function to call
CALL (15),(FUNCTN),VL,MF=(E,CAFCALL)
BAL R14,CHEKCODE       Check the return and reason codes
```
This example code does not show a task that waits on the Db2 termination ECB. If you want such a task, you can code it by using the z/OS WAIT macro to monitor the ECB. You probably want this task to detach the sample code if the termination ECB is posted. That task can also wait on the Db2 startup ECB. This sample waits on the startup ECB at its own task level.

This example code assumes that the variables in the following table are already set:

<table>
<thead>
<tr>
<th>Variable</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>LIALI</td>
<td>The entry point that handles Db2 connection service requests.</td>
</tr>
<tr>
<td>LISQL</td>
<td>The entry point that handles SQL calls.</td>
</tr>
<tr>
<td>SSID</td>
<td>The Db2 subsystem identifier.</td>
</tr>
<tr>
<td>TECB</td>
<td>The address of the Db2 termination ECB.</td>
</tr>
<tr>
<td>SECB</td>
<td>The address of the Db2 startup ECB.</td>
</tr>
<tr>
<td>RIBPTR</td>
<td>A fullword that CAF sets to contain the RIB address.</td>
</tr>
<tr>
<td>PLAN</td>
<td>The plan name to use in the OPEN call.</td>
</tr>
<tr>
<td>CONTROL</td>
<td>This variable is used to shut down processing because of unsatisfactory return or reason codes. The CHECKCODE subroutine sets this value.</td>
</tr>
<tr>
<td>CAFCALL</td>
<td>List-form parameter area for the CALL macro.</td>
</tr>
</tbody>
</table>

**Example of checking return codes and reason codes when using CAF**

The following example code illustrates a way to check the return codes and the Db2 termination ECB after each connection service request and SQL call. The routine sets the variable CONTROL to control further processing within the module.
/* *** Subroutine CHEKCODE checks return codes from DB2 and Call Attach. */
/* When CHEKCODE receives control, R13 should point to the caller's */
/* save area. */
/* *************************************************************** */
CHEKCODE DS OH
STM R14,R12,12(R13) Prolog
ST R15,RETCODE Save the return code
ST R0,REASCODE Save the reason code
LA R15,SAVEAREA Get save area address
ST R13,4(R15) Chain the save areas
ST R15,8(R13) Chain the save areas
LR R13,R15 Put save area address in R13

* ********************* HUNT FOR FORCE OR ABTERM *********************
TM TECB,POSTBIT See if TECB was Posted
BZ DOCHECKS Branch if TECB was not Posted
CLC TECB CODE(3),QUIESCE Is this 'STOP DB2 MODE=FORCE'
BE DOCHECKS If not QUIESCE, was FORCE or ABTERM
MVC CONTROL,SHUTDOWN Shutdown
WRITE 'CAE found found FORCE or ABTERM, shutting down'
ENDCCODE Go to the end of CHEKCODE

DOCHECKS DS OH

* ********************* HUNT FOR 0 ****************************
CLC RETCODE,ZERO Was it a zero?
BE ENDCCODE Nothing to do in CHEKCODE for zero

* ********************* HUNT FOR 4 ****************************
CLC RETCODE,FOUR Was it a 4?
BNE HUNT8 If not a 4, hunt eights
CLC REASCODE,C10831 Was it a release level mismatch?
BNE HUNT24 If not B31, hunt 831
WRITE 'Found a mismatch between DB2 and CAF release levels'
ENDCCODE We are done. Go to end of CHEKCODE

HUNT8 DS OH
CLC RETCODE,EIGHT Hunt return code of 8
BE GOT8OR12 If not 8, hunt 12s
CLC RETCODE,TWELVE Hunt return code of 12
BNE HUNT24 Hunt for 24
WRITE 'Found RETCODE of 8 or 12'
ENDCCODE We are done. Go to end of CHEKCODE

GOT8OR12 DS OH
CLC REASCODE,F30012 Hunt for X'00F30012'
BE DB2DOWN
WRITE 'Got RETCODE = 4 and an unrecognized reason code'
MVC CONTROL,RESTART Indicate that we should re-CONNECT
BE ENDCCODE We are done. Go to end of CHEKCODE

* ********************* HUNT FOR 8 ****************************
HUNT8 DS OH
CLC RETCODE,EIGHT Hunt return code of 8
BE GOT8OR12 If not 8, hunt 12s
CLC RETCODE,TWELVE Hunt return code of 12
BNE HUNT24 Hunt for 24
WRITE 'Found RETCODE of 8 or 12'
CLC REASCODE,F30002 Hunt for X'00F30002'
BE DB2DOWN
WRITE 'DB2 connection failure with an unrecognized REASCODE'
MVC SQLCODE,ZERO See if we need TRANSLATE
BNE ATRANS If not blank, skip TRANSLATE

* ********************** TRANSLATE unrecognized RETCODES **********
WRITE 'SQLCODE 0 but R15 not, so TRANSLATE to get SQLCODE'
L R15,LIALI Get the Language Interface address
CALL (15),(TRANSLAT,SQLCA),VL,ME=(E,CAFSCALL)
Example of invoking CAF when you do not specify the precompiler option ATTACH(CAF)

Each of the four Db2 attachment facilities contains an entry point named DSNHLI. When you use CAF but do not specify the precompiler option ATTACH(CAF), SQL statements result in BALR instructions to DSNHLI in your program. To find the correct DSNHLI entry point without including DSNALI in your load module, code a subroutine with entry point DSNHLI that passes control to entry point DSNHLI2 in the DSNALI module. DSNHLI2 is unique to DSNALI and is at the same location in DSNALI as DSNHLI. DSNALI uses 31-bit addressing. If the application that calls this intermediate subroutine uses 24-bit addressing, this subroutine should account for the difference.

In the following example, LISQL is addressable because the calling CSECT used the same register 12 as CSECT DSNHLI. Your application must also establish addressability to LISQL.

---

Example of invoking CAF when you do not specify the precompiler option ATTACH(CAF)

Each of the four Db2 attachment facilities contains an entry point named DSNHLI. When you use CAF but do not specify the precompiler option ATTACH(CAF), SQL statements result in BALR instructions to DSNHLI in your program. To find the correct DSNHLI entry point without including DSNALI in your load module, code a subroutine with entry point DSNHLI that passes control to entry point DSNHLI2 in the DSNALI module. DSNHLI2 is unique to DSNALI and is at the same location in DSNALI as DSNHLI. DSNALI uses 31-bit addressing. If the application that calls this intermediate subroutine uses 24-bit addressing, this subroutine should account for the difference.

In the following example, LISQL is addressable because the calling CSECT used the same register 12 as CSECT DSNHLI. Your application must also establish addressability to LISQL.

---

Example of invoking CAF when you do not specify the precompiler option ATTACH(CAF)

Each of the four Db2 attachment facilities contains an entry point named DSNHLI. When you use CAF but do not specify the precompiler option ATTACH(CAF), SQL statements result in BALR instructions to DSNHLI in your program. To find the correct DSNHLI entry point without including DSNALI in your load module, code a subroutine with entry point DSNHLI that passes control to entry point DSNHLI2 in the DSNALI module. DSNHLI2 is unique to DSNALI and is at the same location in DSNALI as DSNHLI. DSNALI uses 31-bit addressing. If the application that calls this intermediate subroutine uses 24-bit addressing, this subroutine should account for the difference.

In the following example, LISQL is addressable because the calling CSECT used the same register 12 as CSECT DSNHLI. Your application must also establish addressability to LISQL.

---

Example of invoking CAF when you do not specify the precompiler option ATTACH(CAF)

Each of the four Db2 attachment facilities contains an entry point named DSNHLI. When you use CAF but do not specify the precompiler option ATTACH(CAF), SQL statements result in BALR instructions to DSNHLI in your program. To find the correct DSNHLI entry point without including DSNALI in your load module, code a subroutine with entry point DSNHLI that passes control to entry point DSNHLI2 in the DSNALI module. DSNHLI2 is unique to DSNALI and is at the same location in DSNALI as DSNHLI. DSNALI uses 31-bit addressing. If the application that calls this intermediate subroutine uses 24-bit addressing, this subroutine should account for the difference.

In the following example, LISQL is addressable because the calling CSECT used the same register 12 as CSECT DSNHLI. Your application must also establish addressability to LISQL.

---
Example of variable declarations when using CAF

The following example code shows declarations for some of the variables that were used in the previous subroutines.

```assembly
****************************** VARIABLES ******************************
SECB     DS    F                  DB2 Startup ECB
TECB     DS    F                  DB2 Termination ECB
LIALI    DS    F                  DSNALI Entry Point address
LISQL    DS    F                  DSNHLI2 Entry Point address
SSID     DS    CL4                DB2 Subsystem ID. CONNECT parameter
PLAN     DS    CL8                DB2 Plan name. OPEN parameter
TRMOP    DS    CL4                CLOSE termination option (SYNC|ABRT)
FUNCTN   DS    CL12               CAF function to be called
RIBPTR   DS    F                  DB2 puts Release Info Block addr here
RETCODE  DS    F                  Chekcode saves R15 here
REASCODE DS    F                  Chekcode saves R0 here
CONTROL  DS    CL8                GO, SHUTDOWN, or RESTART
SAVEAREA DS  18F                  Save area for CHEKCODE
****************************** CONSTANTS ******************************
SHUTDOWN DC    CL8'SHUTDOWN'      CONTROL value: Shutdown execution
RESTART  DC    CL8'RESTART '      CONTROL value: Restart execution
CONTINUE DC    CL8'CONTINUE'      CONTROL value: Everything OK, cont
CODE0    DC    F'0'               SQLCODE of 0
CODE100  DC    F'100'             SQLCODE of 100
QUIESC   DC    CL12'QUIESC'      TECB postcode: STOP DB2 MODE=QUIESC
CONNECT  DC    CL12'CONNECT     ' Name of a CAF service. Must be CL12!
OPEN     DC    CL12'OPEN        ' Name of a CAF service. Must be CL12!
CLOSE    DC    CL12'CLOSE       ' Name of a CAF service. Must be CL12!
DISCON   DC    CL12'DISCONNECT ' Name of a CAF service. Must be CL12!
TRANS    DC    CL12'TRANSLATE  ' Name of a CAF service. Must be CL12!
SYNC     DC    CL4'SYNC'          Termination option (COMMIT)
ABRT     DC    CL4'ABRT'          Termination option (ROLLBACK)
****************************** RETURN CODES (R15) FROM CALL ATTACH ****
ZERO     DC    F'0'               0
FOUR     DC    F'4'               4
EIGHT    DC    F'8'               8
TWELVE   DC    F'12'              12  (Call Attach return code in R15)
NUM200   DC    F'200'             200 (User error)
NUM204   DC    F'204'             204 (Call Attach system error)
****************************** REASON CODES (R00) FROM CALL ATTACH ****
C10205   DC    XL4'00C10205'      Call attach could not TRANSLATE
C10831   DC    XL4'00C10831'      Call attach found a release mismatch
C10824   DC    XL4'00C10824'      Call attach ready for more input
F30002   DC    XL4'00F30002'      DB2 subsystem not up
F30011   DC    XL4'00F30011'      DB2 subsystem not up
F30012   DC    XL4'00F30012'      DB2 subsystem not up
F30025   DC    XL4'00F30025'      DB2 is stopping (REASCODE)
*        Insert more codes here as necessary for your application
*
EXEC SQL INCLUDE SQLCA
DSNDRIB               Get the DB2 Release Information Block
CALEDCALL CALL ,(*,*),VL,MF=L
```

Invoking the Resource Recovery Services attachment facility

The Resource Recovery Services attachment facility (RRSAF) enables your program to communicate with Db2. Invoke RRSAF as an alternative to invoking CAF or when using stored procedures that run in a WLM-established address space. RRSAF has more capabilities than CAF.

**Before you begin**

Before you invoke RRSAF, perform the following actions:

- Ensure that the RRSAF language interface load module, DSNRLI, is available.
- Ensure that your application satisfies the requirements for programs that access RRSAF.
- Ensure that your application satisfies the general environment characteristics for connecting to Db2.
- Ensure that you are familiar with the following z/OS concepts and facilities:
  - The CALL macro and standard module linkage conventions
Program addressing and residency options (AMODE and RMODE)
Creating and controlling tasks; multitasking
Functional recovery facilities such as ESTAE, ESTAI, and FRRs
Synchronization techniques such as WAIT/POST
z/OS RRS functions, such as SRRCMIT and SRRBACK

About this task
Applications that use RRSAF can be written in assembler language, C, COBOL, Fortran, and PL/I. When choosing a language to code your application in, consider the following restrictions:

- If you use z/OS macros (ATTACH, WAIT, POST, and so on), choose a programming language that supports them.
- The RRSAF TRANSLATE function is not available in Fortran. To use this function, code it in a routine that is written in another language, and then call that routine from Fortran.

Procedure
To invoke RRSAF:
1. Perform one of the following actions:

   - Explicitly invoke RRSAF by including in your program CALL DSNRLI statements with the appropriate options.
     The first option is an RRSAF connection function, which describes the action that you want RRSAF to take. The effect of any function depends in part on what functions the program has already performed.
     To code RRSAF functions in C, COBOL, Fortran, or PL/I, follow the individual language's rules for making calls to assembler language routines. Specify the return code and reason code parameters in the parameter list for each RRSAF call.

     **Requirement:** For C, C++, and PL/I applications, you must also include in your program the compiler directives that are listed in the following table, because DSNRLI is an assembler language program.

     **Table 15. Compiler directives to include in C, C++, and PL/I applications that contain CALL DSNRLI statements**

     | Language | Compiler directive to include |
     |----------|-------------------------------|
     | C        | #pragma linkage(dsnrli, OS)   |
     | C++      | extern "OS" { int DSNRLI( char * functn, ...); } |
     | PL/I      | DCL DSNRLI ENTRY OPTIONS(ASM,INTER,RETCODE); |

   - Implicitly invoke RRSAF by including SQL statements or IFI calls in your program just as you would in any program. The RRSAF facility establishes the connection to Db2 with the default values for the subsystem name, plan name and authorization ID.

     **Restriction:** If your program can make its first SQL call from different modules with different DBRMs, you cannot use a default plan name and thus, you cannot implicitly invoke RRSAF. Instead, you must explicitly invoke RRSAF by calling the CREATE THREAD function.

     **Requirement:** If your application includes both SQL and IFI calls, you must issue at least one SQL call before you issue any IFI calls. This action ensures that your application uses the correct plan.
2. If you implicitly invoked RRSAF, determine if the implicit connection was successful by examining the return code and reason code immediately after the first executable SQL statement within the application program. Your program can check these codes by performing one of the following actions:

- Examine registers 0 and 15 directly.
- Examine the SQLCA, and if the SQLCODE is -981, obtain the return and reason code from the message text. The return code is the first token, and the reason code is the second token.

If the implicit connection is successful, the application can examine the SQLCODE for the first, and subsequent, SQL statements.

**Example of an RRSAF configuration**

The following figure shows an conceptual example of invoking and using RRSAF.
Resource Recovery Services attachment facility

An attachment facility enables programs to communicate with Db2. The Resource Recovery Services attachment facility (RRSAF) provides such a connection for programs that run in z/OS batch, TSO foreground, and TSO background. The RRSAF is an alternative to CAF and has more functionality.

An application program using RRSAF can perform the following actions:
• Use Db2 to process SQL statements, commands, or instrumentation facility interface (IFI) calls.
• Coordinate Db2 updates with updates made by all other resource managers that also use z/OS RRS in an z/OS system.
• Use the z/OS System Authorization Facility and an external security product, such as RACF, to sign on to Db2 with the authorization ID of a user.
• Sign on to Db2 using a new authorization ID and an existing connection and plan.
• Access Db2 from multiple z/OS tasks in an address space.
• Switch a Db2 thread among z/OS tasks within a single address space.
• Access the Db2 IFI.
• Run with or without the TSO terminal monitor program (TMP).
• Run without being a subtask of the DSN command processor (or of any Db2 code).
• Run above or below the 16-MB line.
• Establish an explicit connection to Db2, through a call interface, with control over the exact state of the connection.
• Establish an implicit connection to Db2 (with a default subsystem identifier and a default plan name) by using SQL statements or IFI calls without first calling RRSAF.
• Supply event control blocks (ECBs), for Db2 to post, that signal start-up or termination.
• Intercept return codes, reason codes, and abend codes from Db2 and translate them into messages as required.

RRSAF uses z/OS Transaction Management and Recoverable Resource Manager Services (z/OS RRS).

Any task in an address space can establish a connection to Db2 through RRSAF. Each task control block (TCB) can have only one connection to Db2. A Db2 service request that is issued by a program that runs under a given task is associated with that task's connection to Db2. The service request operates independently of any Db2 activity under any other task.

Each connected task can run a plan. Tasks within a single address space can specify the same plan, but each instance of a plan runs independently from the others. A task can terminate its plan and run a different plan without completely breaking its connection to Db2.

RRSAF does not generate task structures.

When you design your application, consider that using multiple simultaneous connections can increase the possibility of deadlocks and Db2 resource contention.

Restriction: RRSAF does not provide attention processing exits or functional recovery routines. You can provide whatever attention handling and functional recovery your application needs, but you must use ESTAE/ESTAI type recovery routines only.

A tracing facility provides diagnostic messages that help you debug programs and diagnose errors in the RRSAF code. The trace information is available only in a SYSABEND or SYSUDUMP dump.

To commit work in RRSAF applications, use the CPIC SRRCMIT function or the Db2 COMMIT statement. To roll back work, use the CPIC SRRBACK function or the Db2 ROLLBACK statement.

Use the following guidelines to decide whether to use the Db2 statements or the CPIC functions for commit and rollback operations:
• Use Db2 COMMIT and ROLLBACK statements when all of the following conditions are true:
  – The only recoverable resource that is accessed by your application is Db2 data that is managed by a single Db2 instance.
    Db2 COMMIT and ROLLBACK statements fail if your RRSAF application accesses recoverable resources other than Db2 data that is managed by a single Db2 instance.
  – The address space from which syncpoint processing is initiated is the same as the address space that is connected to Db2.
• If your application accesses other recoverable resources, or syncpoint processing and Db2 access are initiated from different address spaces, use SRRCMIT and SRRBACK.

Related reference
COMMIT (Db2 SQL)
ROLLBACK (Db2 SQL)

Related information
Using Protected Resources (MVS Programming: Callable Services for High-Level Languages)

Properties of RRSAF connections
RRSAF enables programs to communicate with Db2 to process SQL statements, commands, or IFI calls.

Restriction: Do not mix RRSAF connections with other connection types in a single address space. The first connection that is made from an address space to Db2 determines the type of connection allowed.

The connection that RRSAF makes with Db2 has the basic properties that are listed in the following table.

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Connection name</td>
<td>RRSAF</td>
<td>You can use the DISPLAY THREAD command to list RRSAF applications that have the connection name RRSAF.</td>
</tr>
<tr>
<td>Connection type</td>
<td>RRSAF</td>
<td>None.</td>
</tr>
<tr>
<td>Property</td>
<td>Value</td>
<td>Comments</td>
</tr>
<tr>
<td>---------------------</td>
<td>----------------------------------------------------------------------</td>
<td>------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Authorization ID</td>
<td>Authorization IDs that are associated with each Db2 connection</td>
<td>A connection must have a primary ID and can have one or more secondary IDs. Those identifiers are used for the following purposes: • Validating access to Db2 • Checking privileges on Db2 objects • Assigning ownership of Db2 objects • Identifying the user of a connection for audit, performance, and accounting traces. RRSASF relies on the z/OS System Authorization Facility (SAF) and a security product, such as RACF, to verify and authorize the authorization IDs. An application that connects to Db2 through RRSASF must pass those identifiers to SAF for verification and authorization checking. RRSASF retrieves the identifiers from SAF. A location can provide an authorization exit routine for a Db2 connection to change the authorization IDs and to indicate whether the connection is allowed. The actual values that are assigned to the primary and secondary authorization IDs can differ from the values that are provided by a SIGNON or AUTH SIGNON request. A site's Db2 signon exit routine can access the primary and secondary authorization IDs and can modify the IDs to satisfy the site's security requirements. The exit routine can also indicate whether the signon request should be accepted.</td>
</tr>
</tbody>
</table>
Table 16. Properties of RRSAF connections (continued)

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scope</td>
<td>RRSAF processes connections as if each task is entirely isolated. When a task requests a function, RRSAF passes the function to Db2, regardless of the connection status of other tasks in the address space. However, the application program and the Db2 subsystem have access to the connection status of multiple tasks in an address space.</td>
<td>None.</td>
</tr>
</tbody>
</table>

If an application that is connected to Db2 through RRSAF terminates normally before the TERMINATE THREAD or TERMINATE IDENTIFY functions deallocate the plan, RRS commits any changes made after the last commit point. If the application terminates abnormally before the TERMINATE THREAD or TERMINATE IDENTIFY functions deallocate the plan, z/OS RRS rolls back any changes made after the last commit point. In either case, Db2 deallocates the plan, if necessary, and terminates the application's connection.

If Db2 abends while an application is running, Db2 rolls back changes to the last commit point. If Db2 terminates while processing a commit request, Db2 either commits or rolls back any changes at the next restart. The action taken depends on the state of the commit request when Db2 terminates.

**Making the RRSAF language interface (DSNRLI) available**

Before you can invoke the Resource Recovery Services attachment facility (RRSAF), you must first make available the RRSAF language interface load module, DSNRLI.

**About this task**

Part of RRSAF is a Db2 load module, DSNRLI, which is also known as the RRSAF language interface module. DSNRLI has the alias names DSNHLIR and DSNWLIR. The module has five entry points: DSNRLI, DSNHLI, DSNHLIR, DSNWLI, and DSNWLIR. These entry points serve the following functions:

- Entry point DSNRLI handles explicit Db2 connection service requests.
- DSNHLI and DSNHLIR handle SQL calls. Use DSNHLI if your application program link-edits RRSAF. Use DSNHLIR if your application program loads RRSAF.
- DSNWLI and DSNWLIR handle IFI calls. Use DSNWLI if your application program link-edits RRSAF. Use DSNWLIR if your application program loads RRSAF.

**Procedure**

To make DSNRLI available:

1. Decide which of the following methods you want to use to make DSNRLI available:
   - Explicitly issuing LOAD requests when your program runs.
     By explicitly loading the DSNRLI module, you can isolate the maintenance of your application from future IBM maintenance to the language interface. If the language interface changes, the change will probably not affect your load module.
   - Including the DSNRLI module in your load module when you link-edit your program.
     A disadvantage of link-editing DSNRLI into your load module is that if IBM makes a change to DSNRLI, you must link-edit your program again.
Alternatively, if using explicit connections via CALL DSNALI, you can link-edit your program with DSNULI, the Universal Language Interface.

2. Depending on the method that you chose in step 1, perform one of the following actions:

- **If you want to explicitly issue LOAD requests when your program runs:**

  In your program, issue z/OS LOAD service requests for entry points DSNRLI and DSNHLIR. If you use IFI services, you must also load DSNWLIR. Save the entry point address that LOAD returns and use it in the CALL macro.

  Indicate to Db2 which entry point to use in one of the following two ways:
  - Specify the precompiler option ATTACH(RRSAF).
    
    This option causes Db2 to generate calls that specify entry point DSNHLIR.
    
    **Restriction:** You cannot use this option if your application is written in Fortran.
  - Code a dummy entry point named DSNHLI within your load module.

  If you do not specify the precompiler option ATTACH, the Db2 precompiler generates calls to entry point DSNHLI for each SQL request. The precompiler does not know about and is independent of the different Db2 attachment facilities. When the calls that are generated by the Db2 precompiler pass control to DSNHLI, your code that corresponds to the dummy entry point must preserve the option list that is passed in register 1 and call DSNHLIR with the same option list.

- **If you want to include the DSNRLI module in your load module when you link-edit your program:**

  Include DSNRLI in your load module during a link-edit step. For example, you can use a linkage editor control statement that is similar to the following statement in your JCL:

  ```
  INCLUDE DB2LIB(DSNRLI).
  ```

  By coding this statement, you avoid inadvertently picking up the wrong language interface module.

  When you include the DSNRLI module during the link-edit, do not include a dummy DSNHLI entry point in your program or specify the precompiler option ATTACH. Module DSNRLI contains an entry point for DSNHLI, which is identical to DSNHLIR, and an entry point for DSNWLI, which is identical to DSNWLIR.

**Related concepts**

- Program examples for RRSAF
  - The Resource Recovery Services attachment facility (RRSAF) enables programs to communicate with Db2. You can use RRSAF as an alternative to CAF.

- “Universal language interface (DSNULI)” on page 113
  - The universal language interface (DSNULI) subcomponent determines the runtime environment and dynamically loads and branches to the appropriate language interface module.

**Related tasks**

- Making the CAF language interface (DSNALI) available
  - Before you can invoke the call attachment facility (CAF), you must first make DSNALI available.

- Link-editing an application with DSNULI
  - To create a single load module that can be used in more than one attachment environment, you can link-edit your program or stored procedure with the Universal Language Interface module (DSNULI) instead of with one of the environment-specific language interface modules (DSNELI, DSNALI, DSNRLI, DSNCLI, or DFSLI000).

**Requirements for programs that use RRSAF**

The Resource Recovery Services attachment facility (RRSAF) enables programs to communicate with Db2. Before you invoke RRSAF in your program, ensure that your program satisfies any requirements for using RRSAF.

When you write programs that use RRSAF, ensure that they meet the following requirements:
The program accounts for the size of the RRSAF code. The RRSAF code requires about 10 KB of virtual storage per address space and an additional 10 KB for each TCB that uses RRSAF.

If your local environment intercepts and replaces the z/OS LOAD SVC that RRSAF uses, you must ensure that your version of LOAD manages the load list element (LLE) and contents directory entry (CDE) chains like the standard z/OS LOAD macro. RRSAF uses z/OS SVC LOAD to load a module as part of the initialization after your first service request. The module is loaded into fetch-protected storage that has the job-step protection key.

You can prepare application programs to run in RRSAF similar to how you prepare applications to run in other environments, such as CICS, IMS, and TSO. You can prepare an RRSAF application either in the batch environment or by using the Db2 program preparation process. You can use the program preparation system either through DB2I or through the DSNH CLIST.

Related tasks
Preparing an application to run on Db2 for z/OS
To prepare and run applications that contain embedded static SQL statements or dynamic SQL statements, you must precompile, compile, link-edit, and bind them.

How RRSAF modifies the content of registers
If you do not specify the return code and reason code parameters in your RRSAF function calls or if you invoke RRSAF implicitly, RRSAF puts a return code in register 15 and a reason code in register 0. RRSAF preserves the contents of registers 2 through 14.

If you specify the return code and reason code parameters, RRSAF places the return code in register 15 and in the return code parameter to accommodate high-level languages that support special return code processing.

The following table summarizes the register conventions for RRSAF calls.

<table>
<thead>
<tr>
<th>Register</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>R1</td>
<td>Parameter list pointer</td>
</tr>
<tr>
<td>R13</td>
<td>Address of caller's save area</td>
</tr>
<tr>
<td>R14</td>
<td>Caller's return address</td>
</tr>
<tr>
<td>R15</td>
<td>RRSAF entry point address</td>
</tr>
</tbody>
</table>

Implicit connections to RRSAF
Resource Recovery Services attachment facility (RRSAF) establishes an implicit connection to Db2 under certain situations. The connection is established if the following are true: the RRSAF language interface load module (DSNRLI) is available, you do not explicitly specify the IDENTIFY function in a CALL DSNRLI statement in your program, and the application includes SQL statements or IFI calls.

An implicit connection causes RRSAF to initiate implicit IDENTIFY and CREATE THREAD requests to Db2. These requests are subject to the same Db2 return codes and reason codes as explicitly specified requests.

Implicit connections use the following defaults:

Subsystem name
The default name that is specified in the module DSNHDECP. RRSAF uses the installation default DSNHDECP, unless your own DSNHDECP module is in a library in a STEPLIB statement of the JOBLIB concatenation or in the link list. In a data sharing group, the default subsystem name is the group attachment name.

Be certain that you know what the default name is and that it names the specific Db2 subsystem that you want to use.
Plan name
The member name of the database request module (DBRM) that Db2 produced when you precompiled the source program that contains the first SQL call.

Authorization ID
The 7-byte user ID that is associated with the address space, unless an authorized function has built an Accessor Environment Element (ACEE) for the address space. If an authorized function has built an ACEE, Db2 passes the 8-byte user ID from the ACEE.

For an implicit connection request, your application should not explicitly specify either the IDENTIFY function or the CREATE THREAD function. Your application can execute other explicit RRSASF calls after the implicit connection is made. An implicit connection does not perform any SIGNON processing. Your application can execute the SIGNON function at any point of consistency. To terminate an implicit connection, you must use the proper function calls.

For implicit connection requests, register 15 contains the return code, and register 0 contains the reason code. The return code and reason code are also in the message text for SQLCODE -981.

Related concepts
Summary of RRSASF behavior
The effect of any Resource Recovery Services attachment facility (RRSAF) function depends in part on what functions the program has already run. You should plan the RRSASF function calls that your program makes to avoid any errors and major structural problems in your application.

CALL DSNRLI statement parameter list
The CALL DSNRLI statement explicitly invokes RRSASF. When you include CALL DSNRLI statements in your program, you must specify all parameters that precede the return code parameter.

In CALL DSNRLI statements, you cannot omit any of parameters that come before the return code parameter by coding zeros or blanks. No defaults exist for those parameters for explicit connection requests. Defaults are provided for only implicit connections. All parameters starting with the return code parameter are optional.

When you want to use the default value for a parameter but specify subsequent parameters, code the CALL DSNRLI statement as follows:

- For C-language, when you code CALL DSNRLI statements in C, you need to specify the address of every parameter, using the “address of” operator (&), and not the parameter itself. For example, to pass the pklistptr parameter on the “CREATE THREAD” specify the address of the 4-byte pointer to the structure (&pklistptr):

  fnret=dsnrli(&crthrdfn[0], &plan[0], &collid[0], &reuse[0], &retcode, &reascode, &pklistptr);

  For all languages except assembler language, code zero for that parameter in the CALL DSNRLI statement. For example, suppose that you are coding an IDENTIFY call in a COBOL program, and you want to specify all parameters except the return code parameter. You can write a statement similar to the following statement:

  CALL 'DSNRLI' USING IDFYFN SSNM RIBPTR EIBPTR TERMECB STARTECB BY CONTENT ZERO BY REFERENCE REASCODE.

  For assembler language, code a comma for that parameter in the CALL DSNRLI statement. For example, suppose that you are coding an IDENTIFY call, and you want to specify all parameters except the return code parameter. You can write a statement similar to the following statement:

  CALL DSNRLI, (IDFYFN, SSNM, RIBPTR, EIBPTR, TERMECB, STARTECB, , REASCODE)

For assembler programs that invoke RRSASF, use a standard parameter list for an z/OS CALL. Register 1 must contain the address of a list of pointers to the parameters. Each pointer is a 4-byte address. The last address must contain the value 1 in the high-order bit.
Summary of RRSAF behavior

The effect of any Resource Recovery Services attachment facility (RRSAF) function depends in part on what functions the program has already run. You should plan the RRSAF function calls that your program makes to avoid any errors and major structural problems in your application.

The following tables summarize RRSAF behavior after various inputs from application programs. The contents of each table cell indicate the result of calling the function in the first column for that row followed by the function in the current column heading. For example, if you issue TERMINATE THREAD and then IDENTIFY, RRSAF returns reason code X'00C12201'. Use these tables to understand the order in which your application must issue RRSAF calls, SQL statements, and IFI requests.

The RRSAF FIND_DB2_SYSTEMS function is omitted from these tables, because it does not affect the operation of any of the other functions.

The following table summarizes RRSAF behavior when the next call is to the IDENTIFY function, the SWITCH TO function, the SIGNON function, or the CREATE THREAD function.

Table 18. Effect of call order when next call is IDENTIFY, SWITCH TO, SIGNON, or CREATE THREAD

<table>
<thead>
<tr>
<th>Previous function</th>
<th>IDENTIFY</th>
<th>SWITCH TO</th>
<th>SIGNON, AUTH SIGNON, or CONTEXT SIGNON</th>
<th>CREATE THREAD</th>
</tr>
</thead>
<tbody>
<tr>
<td>Empty: first call</td>
<td>IDENTIFY</td>
<td>X'00C12205'</td>
<td>X'00C12204'</td>
<td>X'00C12204'</td>
</tr>
<tr>
<td>IDENTIFY</td>
<td>X'00F30049'</td>
<td>Switch to ssnm</td>
<td>Signon</td>
<td>X'00C12217'</td>
</tr>
<tr>
<td>SWITCH TO</td>
<td>IDENTIFY</td>
<td>Switch to ssnm</td>
<td>Signon</td>
<td>CREATE THREAD</td>
</tr>
<tr>
<td>SIGNON, AUTH SIGNON, or CONTEXT SIGNON</td>
<td>X'00F30049'</td>
<td>Switch to ssnm</td>
<td>Signon</td>
<td>CREATE THREAD</td>
</tr>
<tr>
<td>CREATE THREAD</td>
<td>X'00F30049'</td>
<td>Switch to ssnm</td>
<td>Signon</td>
<td>X'00C12202'</td>
</tr>
<tr>
<td>TERMINATE THREAD</td>
<td>X'00C12201'</td>
<td>Switch to ssnm</td>
<td>Signon</td>
<td>CREATE THREAD</td>
</tr>
<tr>
<td>IFI</td>
<td>X'00F30049'</td>
<td>Switch to ssnm</td>
<td>Signon</td>
<td>X'00C12202'</td>
</tr>
<tr>
<td>SQL</td>
<td>X'00F30049'</td>
<td>Switch to ssnm</td>
<td>X'00F30092'</td>
<td>X'00C12202'</td>
</tr>
<tr>
<td>SRRRCMIT or SRRBACK</td>
<td>X'00F30049'</td>
<td>Switch to ssnm</td>
<td>Signon</td>
<td>X'00C12202'</td>
</tr>
</tbody>
</table>

Notes:

1. Errors are identified by the Db2 reason code that RRSAF returns.
2. Signon means either the SIGNON function, the AUTH SIGNON function, or the CONTEXT SIGNON function.
3. The SIGNON, AUTH SIGNON, or CONTEXT SIGNON functions are not allowed if any SQL operations are requested after the CREATE THREAD function or after the last SRRRCMIT or SRRBACK request.

The following table summarizes RRSAF behavior when the next call is an SQL statement or an IFI call or to the TERMINATE THREAD function, the TERMINATE IDENTIFY function, or the TRANSLATE function.

Table 19. Effect of call order when next call is SQL or IFI, TERMINATE THREAD, TERMINATE IDENTIFY, or TRANSLATE

<table>
<thead>
<tr>
<th>Previous function</th>
<th>SQL or IFI</th>
<th>TERMINATE THREAD</th>
<th>TERMINATE IDENTIFY</th>
<th>TRANSLATE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Empty: first call</td>
<td>SQL or IFI call</td>
<td>X'00C12204'</td>
<td>X'00C12204'</td>
<td>X'00C12204'</td>
</tr>
<tr>
<td>IDENTIFY</td>
<td>SQL or IFI call</td>
<td>X'00C12203'</td>
<td>TERMINATE IDENTIFY</td>
<td>TRANSLATE</td>
</tr>
<tr>
<td>SWITCH TO</td>
<td>SQL or IFI call</td>
<td>TERMINATE THREAD</td>
<td>TERMINATE IDENTIFY</td>
<td>TRANSLATE</td>
</tr>
</tbody>
</table>
Table 19. Effect of call order when next call is SQL or IFI, TERMINATE THREAD, TERMINATE IDENTIFY, or TRANSLATE (continued)

<table>
<thead>
<tr>
<th>Previous function</th>
<th>SQL or IFI</th>
<th>TERMINATE THREAD</th>
<th>TERMINATE IDENTIFY</th>
<th>TRANSLATE</th>
</tr>
</thead>
<tbody>
<tr>
<td>SIGNON, AUTH SIGNON, or CONTEXT SIGNON</td>
<td>SQL or IFI call&lt;sup&gt;4&lt;/sup&gt;</td>
<td>TERMINATE THREAD</td>
<td>TERMINATE IDENTIFY</td>
<td>TRANSLATE</td>
</tr>
<tr>
<td>CREATE THREAD</td>
<td>SQL or IFI call&lt;sup&gt;4&lt;/sup&gt;</td>
<td>TERMINATE THREAD</td>
<td>TERMINATE IDENTIFY</td>
<td>TRANSLATE</td>
</tr>
<tr>
<td>TERMINATE THREAD</td>
<td>SQL or IFI call&lt;sup&gt;4&lt;/sup&gt;</td>
<td>X'00C12203'&lt;sup&gt;1&lt;/sup&gt;</td>
<td>TERMINATE IDENTIFY</td>
<td>TRANSLATE</td>
</tr>
<tr>
<td>IFI</td>
<td>SQL or IFI call&lt;sup&gt;4&lt;/sup&gt;</td>
<td>TERMINATE THREAD</td>
<td>TERMINATE IDENTIFY</td>
<td>TRANSLATE</td>
</tr>
<tr>
<td>SQL</td>
<td>SQL or IFI call&lt;sup&gt;4&lt;/sup&gt;</td>
<td>X'00F30093'&lt;sup&gt;12&lt;/sup&gt;</td>
<td>X'00F30093'&lt;sup&gt;13&lt;/sup&gt;</td>
<td>TRANSLATE</td>
</tr>
<tr>
<td>SRRCMIT or SRRBACK</td>
<td>SQL or IFI call&lt;sup&gt;4&lt;/sup&gt;</td>
<td>TERMINATE THREAD</td>
<td>TERMINATE IDENTIFY</td>
<td>TRANSLATE</td>
</tr>
</tbody>
</table>

Notes:

1. Errors are identified by the Db2 reason code that RRSAF returns.
2. TERMINATE THREAD is not allowed if any SQL operations are requested after the CREATE THREAD function or after the last SRRCMIT or SRRBACK request.
3. TERMINATE IDENTIFY is not allowed if any SQL operations are requested after the CREATE THREAD function or after the last SRRCMIT or SRRBACK request.
4. If you are using an implicit connection to RRSAF and issue SQL or IFI calls, RRSAF issues implicit IDENTIFY and CREATE THREAD requests. If you continue with explicit RRSAF statements, you must follow the standard order of explicit RRSAF calls. Implicitly connecting to RRSAF does not cause an implicit SIGNON request. Therefore, you might need to issue an explicit SIGNON request to satisfy the standard order requirement. For example, an SQL statement followed by an explicit TERMINATE THREAD request results in an error. You must issue an explicit SIGNON request before issuing the TERMINATE THREAD request.

Related concepts

X'C1......' codes (Db2 Codes)
X'F3......' codes (Db2 Codes)

RRSAF connection functions

An Resource Recovery Services attachment facility (RRSAF) connection function specifies the action that you want RRSAF to take. You specify these functions when you invoke RRSAF through CALL DSNRLI statements.

Related concepts
CALL DSNRLI statement parameter list
The CALL DSNRLI statement explicitly invokes RRSAF. When you include CALL DSNRLI statements in your program, you must specify all parameters that precede the return code parameter.

Summary of RRSAF behavior
The effect of any Resource Recovery Services attachment facility (RRSAF) function depends in part on what functions the program has already run. You should plan the RRSAF function calls that your program makes to avoid any errors and major structural problems in your application.

IDENTIFY function for RRSAF
The RRSAF IDENTIFY function initializes a connection to Db2.

The IDENTIFY function establishes the caller's task as a user of Db2 services. If no other task in the address space currently is connected to the specified subsystem, the IDENTIFY function also initializes the address space to communicate with the Db2 address spaces. The IDENTIFY function establishes the cross-memory authorization of the address space to Db2 and builds address space control blocks.
The following diagram shows the syntax for the IDENTIFY function.

```
CALL DSNRLI (function, ssnm, ribptr, eibptr, termecb, startecb, retcode, reascode, groupoverride, decpptr)
```

Parameters point to the following areas:

**function**
An 18-byte area that contains IDENTIFY followed by 10 blanks.

**ssnm**
A 4-byte Db2 subsystem name, or group attachment or subgroup attachment name (if used in a data sharing group) to which the connection is made. If ssnm is less than four characters long, pad it on the right with blanks to a length of four characters.

**ribptr**
A 4-byte area in which RRSAF places the address of the release information block (RIB) after the call. You can use the RIB to determine the release level of the Db2 subsystem to which the application is connected. You can determine the modification level within the release level by examining the RIBCNUMB and RIBCINFO fields. If the value in the RIBCNUMB field is greater than zero, check the RIBCINFO field for modification levels.

If the RIB is not available (for example, if ssnm names a subsystem that does not exist), Db2 sets the 4-byte area to zeros.

The area to which ribptr points is below the 16-MB line.

This parameter is required. However, the application does not need to refer to the returned information.

**eibptr**
A 4-byte area in which RRSAF places the address of the environment information block (EIB) after the call. The EIB contains environment information, such as the data sharing group, the name of the Db2 member to which the IDENTIFY request was issued, and whether new functions are activated in the subsystem. If the Db2 subsystem is not in a data sharing group, RRSAF sets the data sharing group and member names to blanks. If the EIB is not available (for example, if ssnm names a subsystem that does not exist), RRSAF sets the 4-byte area to zeros.

The area to which eibptr points is above the 16-MB line.

This parameter is required. However, the application does not need to refer to the returned information.

**termecb**
The address of the application's event control block (ECB) that is used for Db2 termination. Db2 posts this ECB when the system operator enters the STOP DB2 command or when Db2 is terminating abnormally. Specify a value of 0 if you do not want to use a termination ECB.

The ECB is ignored when Db2 is already stopped. The application program must examine any nonzero RRSAF or Db2 reason codes before issuing a WAIT request on this ECB.

Chapter 2. Connecting to Db2 from your application program 77
RRSAF puts a POST code in the ECB to indicate the type of termination as shown in the following table.

<table>
<thead>
<tr>
<th>POST code</th>
<th>Termination type</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>QUIESCE</td>
</tr>
<tr>
<td>12</td>
<td>FORCE</td>
</tr>
<tr>
<td>16</td>
<td>ABTERM</td>
</tr>
</tbody>
</table>

**startecb**

The address of the application's startup ECB. If Db2 has not started when the application issues the IDENTIFY call, Db2 posts the ECB when Db2 has started. If Db2 is already started, the startup ECB is ignored, and is not applied to the next Db2 startup. If Db2 is not started, and the startup ECB is queued, the termination ECB is ignored.

Enter a value of zero if you do not want to use a startup ECB. Db2 posts no more than one startup ECB per address space. The ECB that is posted is associated with the most recent IDENTIFY call from that address space. The application program must examine any nonzero RRSAF or Db2 reason codes before issuing a WAIT request on this ECB.

**retcode**

A 4-byte area in which RRSAF places the return code.

This parameter is optional. If you do not specify retcode, RRSAF places the return code in register 15 and the reason code in register 0.

**reascode**

A 4-byte area in which RRSAF places a reason code.

This parameter is optional. If you do not specify reascode, RRSAF places the reason code in register 0.

If you specify reascode, you must also specify retcode or its default. You can specify a default for retcode by specifying a comma or zero, depending on the language.

**groupoverride**

An 8-byte area that the application provides. This parameter is optional. If you do not want group attach to be attempted, specify 'NOGROUP'. This string indicates that the subsystem name that is specified by ssnm is to be used as a Db2 subsystem name, even if ssnm matches a group attachment or subgroup attachment name. If groupoverride is not provided, ssnm is used as the group attachment or subgroup attachment name if it matches a group attachment or subgroup attachment name.

If you specify this parameter in any language except assembler, you must also specify the retcode and reascode parameters. In assembler language, you can omit the retcode and reascode parameters by specifying commas as place-holders.

**Recommendation**: Avoid using the groupoverride parameter when possible, because it limits the ability to do dynamic workload routing in a Parallel Sysplex. However, you should use this parameter in a data sharing environment when you want to connect to a specific member of a data sharing group, and the subsystem name of that member is the same as the group attachment or subgroup attachment name.

**decpptr**

A 4-byte area in which RRSAF is to put the address of the DSNHDECP or a user-specified application defaults module that was loaded by subsystem ssnm when that subsystem was started. This 4-byte area is a 31-bit pointer. If ssnm is not found, the 4-byte area is set to 0.

The area to which decpptr points is above the 16-MB line.

If you specify this parameter in any language except assembler, you must also specify the retcode, reascode, and groupoverride parameters. In assembler language, you can omit the retcode, reascode, and groupoverride parameters by specifying commas as placeholders.
Example of RRSAF IDENTIFY function calls

The following table shows an IDENTIFY call in each language.

Table 21. Examples of RRSAF IDENTIFY calls

<table>
<thead>
<tr>
<th>Language</th>
<th>Call example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Assembler</td>
<td>CALL DSNRLI,(IDFYFN,SSNM,RIBPTR,EIBPTR,TERMECB,STARTECB, RETCODE,REASCODE,GRPOVER,DECPPTR)</td>
</tr>
<tr>
<td>C¹</td>
<td>fnret=dsnrl1(&amp;idfyfn[0],&amp;ssnm[0], &amp;ribptr, &amp;eibptr, &amp;termecb, &amp;startecb, &amp;retcode, &amp;reascode,&amp;grpover[0],&amp;decpptr);</td>
</tr>
<tr>
<td>COBOL</td>
<td>CALL 'DSNRLI' USING IDFYFN SSNM RIBTPR EIBPTR TERMECB STARTECB RETCODE REASCODE GRPOVER DECPPTR.</td>
</tr>
<tr>
<td>Fortran</td>
<td>CALL DSNRLI(IDFYFN,SSNM,RIBPTR,EIBPTR,TERMECB,STARTECB, RETCODE,REASCODE,GRPOVER,DECPPTR)</td>
</tr>
<tr>
<td>PL/I¹</td>
<td>CALL DSNRLI(IDFYFN,SSNM,RIBPTR,EIBPTR,TERMECB,STARTECB, RETCODE,REASCODE,GRPOVER,DECPPTR);</td>
</tr>
</tbody>
</table>

Note:
1. For C, C++, and PL/I applications, you must include the appropriate compiler directives, because DSNRLI is an assembler language program. These compiler directives are described in the instructions for invoking RRSAF.

Internal processing for the IDENTIFY function

When you call the IDENTIFY function, Db2 performs the following steps:

1. Db2 determines whether the user address space is authorized to connect to Db2. Db2 invokes the z/OS SAF and passes a primary authorization ID to SAF. That authorization ID is the 7-byte user ID that is associated with the address space, unless an authorized function has built an ACEE for the address space. If an authorized function has built an ACEE, Db2 passes the 8-byte user ID from the ACEE. SAF calls an external security product, such as RACF, to determine if the task is authorized to use the following items:
   - The Db2 resource class (CLASS=DSNR)
   - The Db2 subsystem (SUBSYS=ssnm)
   - Connection type RRSAF

2. If that check is successful, Db2 calls the Db2 connection exit routine to perform additional verification and possibly change the authorization ID.

3. Db2 searches for a matching trusted context in the system cache and then the catalog based on the following criteria:
   - The primary authorization ID matches a trusted context SYSTEM AUTHID.
   - The job or started task name matches the JOBNAME attribute that is defined for the identified trusted context.

   If a trusted context is defined, Db2 checks if SECURITY LABEL is defined in the trusted context. If SECURITY LABEL is defined, Db2 verifies the SECURITY LABEL with RACF by using the RACROUTE VERIFY request. This security label is used to verify multi-level security for SYSTEM AUTHID.

   If a matching trusted context is defined, Db2 establishes the connection as trusted. Otherwise, the connection is established without any additional privileges.
4. Db2 then sets the connection name to RRSAF and the connection type to RRSAF.

Related tasks
Invoking the Resource Recovery Services attachment facility
The Resource Recovery Services attachment facility (RRSAF) enables your program to communicate with Db2. Invoke RRSAF as an alternative to invoking CAF or when using stored procedures that run in a WLM-established address space. RRSAF has more capabilities than CAF.

SWITCH TO function for RRSAF
The RRSAF SWITCH TO function directs RRSAF, SQL, or IFI requests to a specified Db2 subsystem. Use the SWITCH TO function to establish connections to multiple Db2 subsystems from a single task.

The SWITCH TO function is useful only after a successful IDENTIFY call. If you have established a connection with one Db2 subsystem, you must issue a SWITCH TO call before you make an IDENTIFY call to another Db2 subsystem. Otherwise, Db2 returns return code X'200' and reason code X'00C12201'.

The first time that you make a SWITCH TO call to a new Db2 subsystem, Db2 returns return code 4 and reason code X'00C12205' as a warning to indicate that the current task has not yet been identified to the new Db2 subsystem.

The following diagram shows the syntax for the SWITCH TO function.

```
DSNRLI SWITCH TO function
CALL DSNRLI (function,ssnm,retcode,reascode,groupoverride)
```

Parameters point to the following areas:

**function**
An 18-byte area that contains SWITCH TO followed by nine blanks.

**ssnm**
A 4-byte Db2 subsystem name, or group attachment or subgroup attachment name (if used in a data sharing group) to which the connection is made. If ssnm is less than four characters long, pad it on the right with blanks to a length of four characters.

**retcode**
A 4-byte area in which RRSAF places the return code.

This parameter is optional. If you do not specify retcode, RRSAF places the return code in register 15 and the reason code in register 0.

**reascode**
A 4-byte area in which RRSAF places the reason code.

This parameter is optional. If you do not specify reascode, RRSAF places the reason code in register 0.

If you specify this parameter, you must also specify retcode.

**groupoverride**
An 8-byte area that the application provides. This parameter is optional. If you do not want group attach to be attempted, specify 'NOGROUP'. This string indicates that the subsystem name that is specified by ssnm is to be used as a Db2 subsystem name, even if ssnm matches a group attachment or subgroup attachment name. If groupoverride is not provided, ssnm is used as the group attachment or subgroup attachment name if it matches a group attachment or subgroup attachment name.
If you specify this parameter in any language except assembler, you must also specify the *retcode* and *reascode* parameters. In assembler language, you can omit the *retcode* and *reascode* parameters by specifying commas as place-holders.

**Recommendation:** Avoid using the *groupoverride* parameter when possible, because it limits the ability to do dynamic workload routing in a Parallel Sysplex. However, you should use this parameter in a data sharing environment when you want to connect to a specific member of a data sharing group, and the subsystem name of that member is the same as the group attachment or subgroup attachment name.

**Examples of RRSAF SWITCH TO calls**

The following table shows a SWITCH TO call in each language.

<table>
<thead>
<tr>
<th>Language</th>
<th>Call example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Assembler</td>
<td>CALL DSNRLI,(SWITCHFN,SSNM,RETCODE,REASCODE,GRPOVER)</td>
</tr>
<tr>
<td>C</td>
<td>fnret=dsnrl(&amp;switchfn[0], &amp;ssnm[0], &amp;retcode, &amp;reascode,&amp;grpover[0]);</td>
</tr>
<tr>
<td>COBOL</td>
<td>CALL 'DSNRLI' USING SWITCHFN RETCODE REASCODE GRPOVER.</td>
</tr>
<tr>
<td>Fortran</td>
<td>CALL DSNRLI(SWITCHFN,RETCODE,REASCODE,GRPOVER)</td>
</tr>
<tr>
<td>PL/1</td>
<td>CALL DSNRLI(SWITCHFN,RETCODE,REASCODE,GRPOVER);</td>
</tr>
</tbody>
</table>

1. For C, C++, and PL/1 applications, you must include the appropriate compiler directives, because DSNRLI is an assembler language program. These compiler directives are described in the instructions for invoking RRSAF.

**Example of using the SWITCH TO function to interact with multiple Db2 subsystems**

The following example shows how you can use the SWITCH TO function to interact with three Db2 subsystems.

```
RRSAF calls for subsystem db21:
  IDENTIFY
  SIGNON
  CREATE THREAD
  Execute SQL on subsystem db21
  SWITCH TO db22
  IF retcode = 4 AND reascode = '00C12205'X THEN
    DO;
    RRSAF calls on subsystem db22:
      IDENTIFY
      SIGNON
      CREATE THREAD
    END;
    Execute SQL on subsystem db22
  SWITCH TO db23
  IF retcode = 4 AND reascode = '00C12205'X THEN
    DO;
    RRSAF calls on subsystem db23:
      IDENTIFY
      SIGNON
      CREATE THREAD
    END;
    Execute SQL on subsystem 23
  SWITCH TO db21
  Execute SQL on subsystem 21
  SWITCH TO db22
```
Related tasks

Invoking the Resource Recovery Services attachment facility

The Resource Recovery Services attachment facility (RRSAF) enables your program to communicate with Db2. Invoke RRSAF as an alternative to invoking CAF or when using stored procedures that run in a WLM-established address space. RRSAF has more capabilities than CAF.

SIGNON function for RRSAF

The RRSAF SIGNON function establishes a primary authorization ID and, optionally, one or more secondary authorization IDs for a connection.

Requirement: Your program does not need to be an authorized program to issue the SIGNON call. For that reason, before you issue the SIGNON call, you must issue the RACF external security interface macro RACROUTE REQUEST=VERIFY to perform the following actions:

- Define and populate an ACEE to identify the user of the program.
- Associate the ACEE with the user's TCB.
- Verify that the user is defined to RACF and authorized to use the application.

Generally, you issue a SIGNON call after an IDENTIFY call and before a CREATE THREAD call. You can also issue a SIGNON call if the application is at a point of consistency, and one of the following conditions is true:

- The value of reuse in the CREATE THREAD call was RESET.
- The value of reuse in the CREATE THREAD call was INITIAL, no held cursors are open, the package or plan is bound with KEEPDYNAMIC(NO), and all special registers are at their initial state. If open held cursors exist or the package or plan is bound with KEEPDYNAMIC(YES), you can issue a SIGNON call only if the primary authorization ID has not changed.

After you issue a SIGNON call, subsequent SQL statements return an error (SQLCODE -900) if the both of following conditions are true:

- The connection was established as trusted when it was initialized.
- The primary authorization ID that was used when you issued the SIGNON call is not allowed to use the trusted connection.

If a trusted context is defined, Db2 checks if SECURITY LABEL is defined in the trusted context. If SECURITY LABEL is defined, Db2 verifies the security label with RACF by using the RACROUTE VERIFY request. This security label is used to verify multi-level security for SYSTEM AUTHID.

The following diagram shows the syntax for the SIGNON function.
Parameters point to the following areas:

**function**
An 18-byte area that contains SIGNON followed by twelve blanks.

**correlation-id**
A 12-byte area in which you can put a Db2 correlation ID. The correlation ID is displayed in Db2 accounting and statistics trace records. You can use the correlation ID to correlate work units. This token appears in the output from the DISPLAY THREAD command. If you do not want to specify a correlation ID, fill the 12-byte area with blanks.

**accounting-token**
A 22-byte area in which you can put a value for a Db2 accounting token. This value is displayed in Db2 accounting and statistics trace records in the QWHCTOKN field, which is mapped by DSNDQWHC DSECT. Setting the value of the accounting token sets the value of the CURRENT CLIENT_ACCTNG special register. If accounting-token is less than 22 characters long, you must pad it on the right with blanks to a length of 22 characters. If you do not want to specify an accounting token, fill the 22-byte area with blanks.

Alternatively, you change the value of the Db2 accounting token with RRSAF functions AUTH SIGNON, CONTEXT SIGNON or SET_CLIENT_ID. You can retrieve the Db2 accounting token with the CURRENT CLIENT_ACCTNG special register only if the DDF accounting string is not set.

**accounting-interval**
A 6-byte area that specifies when Db2 writes an accounting record.

If you specify COMMIT in that area, Db2 writes an accounting record each time that the application issues SRRCMIT. This accounting record is written at the end of the second phase of a two-phase commit. If the accounting interval is COMMIT, and an SRRCMIT is issued while a held cursor is open, the accounting interval spans that commit and ends at the next valid accounting interval end point (such as the next SRRCMIT that is issued without open held cursors, application termination, or SIGNON with a new authorization ID).

If you specify any other value, Db2 writes an accounting record when the application terminates or when you call the SIGNON function with a new authorization ID.

**retcode**
A 4-byte area in which RRSAF places the return code.

This parameter is optional. If you do not specify retcode, RRSAF places the return code in register 15 and the reason code in register 0.
reascode
A 4-byte area in which RRSAF places the reason code.
This parameter is optional. If you do not specify reascode, RRSAF places the reason code in register 0.
If you specify this parameter, you must also specify retcode.

user
A 16-byte area that contains the user ID of the client user. You can use this parameter to provide
the identity of the client user for accounting and monitoring purposes. Db2 displays this user ID in
the output from the DISPLAY THREAD command and in Db2 accounting and statistics trace records.
Setting the user ID sets the value of the CURRENT CLIENT_USERID special register. If user is less than
16 characters long, you must pad it on the right with blanks to a length of 16 characters.
This parameter is optional. If you specify user, you must also specify retcode and reascode. If you do
not specify user, no user ID is associated with the connection.

appl
A 32-byte area that contains the application or transaction name of the user's application. You can
use this parameter to provide the identity of the client user for accounting and monitoring purposes.
Db2 displays the application name in the output from the DISPLAY THREAD command and in Db2
accounting and statistics trace records. Setting the application name sets the value of the CURRENT
CLIENT_APPLNAME special register. If appl is less than 32 characters long, you must pad it on the
right with blanks to a length of 32 characters.
This parameter is optional. If you specify appl, you must also specify retcode, reascode, and user. If
you do not specify appl, no application or transaction is associated with the connection.

ws
An 18-byte area that contains the workstation name of the client user. You can use this parameter
to provide the identity of the client user for accounting and monitoring purposes. Db2 displays
the workstation name in the output from the DISPLAY THREAD command and in Db2 accounting
and statistics trace records. Setting the workstation name sets the value of the CURRENT
CLIENT_WRKSTNNAME special register. If ws is less than 18 characters long, you must pad it on
the right with blanks to a length of 18 characters.
This field is optional. If you specify ws, you must also specify retcode, reascode, user, and appl. If you
do not specify ws, no workstation name is associated with the connection.

xid
A 4-byte area that indicates whether the thread is part of a global transaction. A Db2 thread that is
part of a global transaction can share locks with other Db2 threads that are part of the same global
transaction and can access and modify the same data. A global transaction exists until one of the
threads that is part of the global transaction is committed or rolled back.
You can specify one of the following values for xid:

0
Indicates that the thread is not part of a global transaction. The value 0 must be specified as a
binary integer.

1
Indicates that the thread is part of a global transaction and that Db2 should retrieve the global
transaction ID from RRS. If a global transaction ID already exists for the task, the thread becomes
part of the associated global transaction. Otherwise, RRS generates a new global transaction ID.
The value 1 must be specified as a binary integer. Alternatively, if you want Db2 to return the
generated global transaction ID to the caller, specify an address instead of 1.

address
The 4-byte address of an area in which you enter a global transaction ID for the thread. If the
global transaction ID already exists, the thread becomes part of the associated global transaction.
Otherwise, RRS creates a new global transaction with the ID that you specify.
Alternatively, if you want Db2 to generate and return a global transaction ID, pass the address of
a null global transaction ID by setting the format ID field of the global transaction ID to binary -1
Db2 then replaces the contents of the area with the generated transaction ID. The area at the specified address must be in writable storage and have a length of at least 140 bytes to accommodate the largest possible transaction ID value.

The following table shows the format of a global transaction ID.

<table>
<thead>
<tr>
<th>Field description</th>
<th>Length in bytes</th>
<th>Data type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Format ID</td>
<td>4</td>
<td>Integer</td>
</tr>
<tr>
<td>Global transaction ID length (1 - 64)</td>
<td>4</td>
<td>Integer</td>
</tr>
<tr>
<td>Branch qualifier length (1 - 64)</td>
<td>4</td>
<td>Integer</td>
</tr>
<tr>
<td>Global transaction ID</td>
<td>1 to 64</td>
<td>Character</td>
</tr>
<tr>
<td>Branch qualifier</td>
<td>0 to 64</td>
<td>Character</td>
</tr>
</tbody>
</table>

**accounting-string**

A one-byte length field and a 255-byte area in which you can put a value for a Db2 accounting string. This value is placed in the DDF accounting trace records in the QMDASQLI field, which is mapped by DSNDQMDA DSECT. If accounting-string is less than 255 characters, you must pad it on the right with zeros to a length of 255 bytes. The entire 256 bytes is mapped by DSNDQMDA DSECT.

This parameter is optional. If you specify accounting-string, you must also specify reetcode, reascode, user, appl, and xid. If you do not specify accounting-string, no accounting string is associated with the connection.

You can also change the value of the accounting string with RRSAF functions AUTH SIGNON, CONTEXT SIGNON, or SET_CLIENT_ID.

You can retrieve the DDF suffix portion of the accounting string with the CURRENT CLIENT_ACCTNG special register. The suffix portion of accounting-string can contain a maximum of 200 characters. The QMDASFLN field contains the accounting suffix length, and the QMDASUXF field contains the accounting suffix value. If the DDF accounting string is set, you cannot query the accounting token with the CURRENT CLIENT_ACCTNG special register.

The following parameters are optional and positional. These parameters override values specified earlier in the parameter list. To provide a value for a length, value pair, you must provide a value or specify a 0 length for previous parameters in the parameter list.

**user-length, user-longname**

A pair of parameters that consist of a 2-byte integer length and 128-byte string area. A comma separates the parameters. You can provide the user ID of the client user for accounting and monitoring purposes in user-longname. Db2 displays this user ID in the output from the DISPLAY THREAD command and in Db2 accounting and statistics trace records. Setting the user ID sets the value of the CURRENT CLIENT_USERID special register. Trailing blanks in user-longname are truncated and the length in user-length is updated.

These parameters are optional, to specify them you must also specify a value for accounting-string. A value of 0 in user-length skips processing of user-longname.

**Important:** These parameters override any value that is provided in user.

**appl-length, appl-longname**

A pair of parameters that consist of a 2-byte integer length and 255-byte string area. A comma separates the parameters. You can provide the application or transaction name of the client user for accounting and monitoring purposes in appl-longname. Db2 displays this application name in the output from the DISPLAY THREAD command and in Db2 accounting and statistics trace records. Setting the application name sets the value of the CURRENT CLIENT_APPLNAME special register. Trailing blanks in appl-longname are truncated and the length in appl-length is updated.
These parameters are optional, to specify them you must also specify a value for `user-length`, `user-longname`. A value of 0 in `appl-length` skips processing of `appl-longname`.

**Important:** These parameters override any value that is provided in `appl`.

### ws-length, ws-longname

A pair of parameters that consist of a 2-byte integer length and 255-byte string area. A comma separates the parameters. You can provide the workstation name of the client user for accounting and monitoring purposes in `ws-longname`. `Db2` displays this workstation name in the output from the DISPLAY THREAD command and in `Db2` accounting and statistics trace records. Setting the workstation name sets the value of the CURRENT_CLIENT_WRKSTNNNAME special register. Trailing blanks in `ws-longname` are truncated and the length in `ws-length` is updated.

These parameters are optional, to specify them you must also specify a value for `appl-length`, `appl-longname`. A value of 0 in `ws-length` skips processing of `ws-longname`.

**Important:** These parameters override any value that is provided in `ws`.

### correlation-length, correlation-longname

A pair of parameters that consist of a 2-byte integer length and 255-byte string area. A comma separates the parameters. You can provide a unique value to correlate your business process names with `Db2` threads in `correlation-longname`. `Db2` displays this correlation token in the output from the DISPLAY THREAD DETAIL command. The CURRENT_CLIENT_CORR_TOKEN special register contains the client correlation token. Trailing blanks in `correlation-longname` are truncated and the length in `correlation-length` is updated.

These parameters are optional, to specify them you must also specify a value for `ws-length`, `ws-longname`. A value of 0 in `correlation-length` skips processing of `correlation-longname`.

You can also change the value of the client correlation token with the RRSAF AUTH SIGNON function and the SET_CLIENT_ID function.

**Example of RRSAF SIGNON calls**

The following table shows a SIGNON call in each language.

<table>
<thead>
<tr>
<th>Table 24. Examples of RRSAF SIGNON calls</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Language</strong></td>
</tr>
<tr>
<td>-------------</td>
</tr>
<tr>
<td>assembler</td>
</tr>
<tr>
<td>C</td>
</tr>
<tr>
<td>COBOL</td>
</tr>
<tr>
<td>Fortran</td>
</tr>
<tr>
<td>PL/I</td>
</tr>
</tbody>
</table>

**Note:**

1. For C, C++, and PL/I applications, you must include the appropriate compiler directives, because DSNRLI is an assembler language program. These compiler directives are described in the instructions for invoking RRSAF.
The following example shows a SIGNON call in C with all parameters passed in. Parameters that are numbers are passed in as integers and strings as character arrays. In this example, if &useridlen is larger than 0, then the value of CURRENT CLIENT_USERID special register is the value that is stored in &userid[0].

```c
fnret=dsnrli(&sgnonfn[0],&corrid[0],&accttkn[0],&acctint[0],&retcode,&reascode,
&userid[0],&applname[0],&wsname[0],&xidptr,&lacctngid[0],
&useridlen,&userid[0],&applidlen,&lapplid[0],&wsidlen,&lwsid[0],
&corrtkidlen,&lcorrtkid[0]);
```

**Note:**

1. For C applications, you must include the appropriate compiler directives, because DSNRLI is an assembler language program. These compiler directives are described in the instructions for invoking RRSAF.

**Related tasks**

Invoking the Resource Recovery Services attachment facility

The Resource Recovery Services attachment facility (RRSAF) enables your program to communicate with Db2. Invoke RRSAF as an alternative to invoking CAF or when using stored procedures that run in a WLM-established address space. RRSAF has more capabilities than CAF.

**Related reference**

RACROUTE REQUEST=VERIFY (standard form) (Security Server RACROUTE Macro Reference)

**AUTH SIGNON function for RRSAF**

The RRSAF AUTH SIGNON function enables an APF authorization program to pass an ID to Db2.

An APF-authorized program can pass to Db2 either a primary authorization ID and, optionally, one or more secondary authorization IDs, or an ACEE that is used for authorization checking. These IDs are then associated with the connection.

Generally, you issue an AUTH SIGNON call after an IDENTIFY call and before a CREATE THREAD call. You can also issue an AUTH SIGNON call if the application is at a point of consistency, and one of the following conditions is true:

- The value of `reuse` in the CREATE THREAD call was RESET.
- The value of `reuse` in the CREATE THREAD call was INITIAL, no held cursors are open, the package or plan is bound with KEEPDYNAMIC(NO), and all special registers are at their initial state. If open held cursors exist or the package or plan is bound with KEEPDYNAMIC(YES), a SIGNON call is permitted only if the primary authorization ID has not changed.

The following diagram shows the syntax for the AUTH SIGNON function.
Parameters point to the following areas:

**function**
An 18-byte area that contains AUTH SIGNON followed by seven blanks.

**correlation-id**
A 12-byte area in which you can put a Db2 correlation ID. The correlation ID is displayed in Db2 accounting and statistics trace records. You can use the correlation ID to correlate work units. This token appears in output from the DISPLAY THREAD command. If you do not want to specify a correlation ID, fill the 12-byte area with blanks.

**accounting-token**
A 22-byte area in which you can put a value for a Db2 accounting token. This value is displayed in Db2 accounting and statistics trace records in the QWHCTOKN field, which is mapped by DSNDQWHC DSECT. Setting the value of the accounting token sets the value of the CURRENT CLIENT_ACCTNG special register. If accounting-token is less than 22 characters long, you must pad it on the right with blanks to a length of 22 characters. If you do not want to specify an accounting token, fill the 22-byte area with blanks.

You can also change the value of the Db2 accounting token with RRSAF functions SIGNON, CONTEXT SIGNON, or SET_CLIENT_ID. You can retrieve the Db2 accounting token with the CURRENT CLIENT_ACCTNG special register only if the DDF accounting string is not set.

**accounting-interval**
A 6-byte area with that specifies when Db2 writes an accounting record.

If you specify COMMIT in that area, Db2 writes an accounting record each time that the application issues SRRCMIT. This accounting record is written at the end of the second phase of a two-phase commit. If the accounting interval is COMMIT, and an SRRCMIT is issued while a held cursor is open, the accounting interval spans that commit and ends at the next valid accounting interval end point (such as the next SRRCMIT that is issued without open held cursors, application termination, or SIGNON with a new authorization ID).

If you specify any other value, Db2 writes an accounting record when the application terminates or when you call the SIGNON function with a new authorization ID.

**primary-authid**
An 8-byte area in which you can put a primary authorization ID. If you are not passing the authorization ID to Db2 explicitly, put X'00' or a blank in the first byte of the area.

**ACEE-address**
The 4-byte address of an ACEE that you pass to Db2. If you do not want to provide an ACEE, specify 0 in this field.

**secondary-authid**
An 8-byte area in which you can put a secondary authorization ID. If you do not pass the authorization ID to Db2 explicitly, put X'00' or a blank in the first byte of the area. If you enter a secondary authorization ID, you must also enter a primary authorization ID.

**retcode**
A 4-byte area in which RRSAF places the return code.

This parameter is optional. If you do not specify retcode, RRSAF places the return code in register 15 and the reason code in register 0.

**reascode**
A 4-byte area in which RRSAF places the reason code.

This parameter is optional. If you do not specify reascode, RRSAF places the reason code in register 0. If you specify reascode, you must also specify retcode.
**user**

A 16-byte area that contains the user ID of the client user. You can use this parameter to provide the identity of the client user for accounting and monitoring purposes. Db2 displays this user ID in the output from the DISPLAY THREAD command and in Db2 accounting and statistics trace records. Setting the user ID sets the value of the CURRENT CLIENT_USERID special register. If user is less than 16 characters long, you must pad it on the right with blanks to a length of 16 characters.

This parameter is optional. If you specify user, you must also specify retcode and reascode. If you do not specify this parameter, no user ID is associated with the connection.

**appl**

A 32-byte area that contains the application or transaction name of the user’s application. You can use this parameter to provide the identity of the client user for accounting and monitoring purposes. Db2 displays the application name in the output from the DISPLAY THREAD command and in Db2 accounting and statistics trace records. Setting the application name sets the value of the CURRENT CLIENT_APPLNAME special register. If appl is less than 32 characters long, you must pad it on the right with blanks to a length of 32 characters.

This parameter is optional. If you specify appl, you must also specify retcode, reascode, and user. If you do not specify this parameter, no application or transaction is associated with the connection.

**ws**

An 18-byte area that contains the workstation name of the client user. You can use this parameter to provide the identity of the client user for accounting and monitoring purposes. Db2 displays the workstation name in the output from the DISPLAY THREAD command and in Db2 accounting and statistics trace records. Setting the workstation name sets the value of the CURRENT CLIENT_WRKSTNNAME special register. If ws is less than 18 characters long, you must pad it on the right with blanks to a length of 18 characters.

This parameter is optional. If you specify ws, you must also specify retcode, reascode, user, and appl. If you do not specify this parameter, no workstation name is associated with the connection.

You can also change the value of the workstation name with RRSAF functions SIGNON, CONTEXT SIGNON, or SET_CLIENT_ID. You can retrieve the workstation name with the CURRENT CLIENT_WRKSTNNAME special register.

**xid**

A 4-byte area that indicates whether the thread is part of a global transaction. A Db2 thread that is part of a global transaction can share locks with other Db2 threads that are part of the same global transaction and can access and modify the same data. A global transaction exists until one of the threads that is part of the global transaction is committed or rolled back.

You can specify one of the following values for xid:

0 Indicates that the thread is not part of a global transaction. The value 0 must be specified as a binary integer.

1 Indicates that the thread is part of a global transaction and that Db2 should retrieve the global transaction ID from RRS. If a global transaction ID already exists for the task, the thread becomes part of the associated global transaction. Otherwise, RRS generates a new global transaction ID. The value 1 must be specified as a binary integer. Alternatively, if you want Db2 to return the generated global transaction ID to the caller, specify an address instead of 1.

**address**

The 4-byte address of an area into which you enter a global transaction ID for the thread. If the global transaction ID already exists, the thread becomes part of the associated global transaction. Otherwise, RRS creates a new global transaction with the ID that you specify.

Alternatively, if you want Db2 to generate and return a global transaction ID, pass the address of a null global transaction ID by setting the format ID field of the global transaction ID to binary -1 ("FFFFFFF"X). Db2 then replaces the contents of the area with the generated transaction ID. The
area at the specified address must be in writable storage and have a length of at least 140 bytes to accommodate the largest possible transaction ID value.

The format of a global transaction ID is shown in the description of the RRSAF SIGNON function.

**accounting-string**
A 1-byte length field and a 255-byte area in which you can put a value for a Db2 accounting string. This value is placed in the DDF accounting trace records in the QMDASQLI field, which is mapped by DSNDQMDA DSECT. If accounting-string is less than 255 characters, you must pad it on the right with zeros to a length of 255 bytes. The entire 256 bytes is mapped by DSNDQMDA DSECT.

This parameter is optional. If you specify this accounting-string, you must also specify reatcode, reascode, user, appl, and xid. If you do not specify this parameter, no accounting string is associated with the connection.

You can also change the value of the accounting string with RRSAF functions AUTH SIGNON, CONTEXT SIGNON, or SET_CLIENT_ID.

You can retrieve the DDF suffix portion of the accounting string with the CURRENT CLIENT_ACCTNG special register. The suffix portion of accounting-string can contain a maximum of 200 characters. The QMDASFLN field contains the accounting suffix length, and the QMDASUFX field contains the accounting suffix value. If the DDF accounting string is set, you cannot query the accounting token with the CURRENT CLIENT_ACCTNG special register.

The following parameters are optional and positional. These parameters override values specified earlier in the parameter list. To provide a value for a length, value pair, you must provide a value or specify a 0 length for previous parameters in the parameter list.

**user-length, user-longname**
A pair of parameters that consist of a 2-byte integer length and 128-byte string area. A comma separates the parameters. You can provide the user ID of the client user for accounting and monitoring purposes in user-longname. Db2 displays this user ID in the output from the DISPLAY THREAD command and in Db2 accounting and statistics trace records. Setting the user ID sets the value of the CURRENT CLIENT_USERID special register. Trailing blanks in user-longname are truncated and the length in user-length is updated.

These parameters are optional, to specify them you must also specify a value for accounting-string. A value of 0 in user-length skips processing of user-longname.

**appl-length, appl-longname**
A pair of parameters that consist of a 2-byte integer length and 255-byte string area. A comma separates the parameters. You can provide the application or transaction name of the client user for accounting and monitoring purposes in appl-longname. Db2 displays this application name in the output from the DISPLAY THREAD command and in Db2 accounting and statistics trace records. Setting the application name sets the value of the CURRENT CLIENT_APPLNAME special register. Trailing blanks in appl-longname are truncated and the length in appl-length is updated.

These parameters are optional, to specify them you must also specify a value for user-length, user-longname. A value of 0 in appl-length skips processing of appl-longname.

**ws-length, ws-longname**
A pair of parameters that consist of a 2-byte integer length and 255-byte string area. A comma separates the parameters. You can provide the workstation name of the client user for accounting and monitoring purposes in ws-longname. Db2 displays this workstation name in the output from the DISPLAY THREAD command and in Db2 accounting and statistics trace records. Setting the workstation name sets the value of the CURRENT CLIENT_WRKSTNNAME special register. Trailing blanks in ws-longname are truncated and the length in ws-length is updated.

These parameters are optional, to specify them you must also specify a value for appl-length, appl-longname. A value of 0 in ws-length skips processing of ws-longname.
Important: These parameters override any value that is provided in ws.

**correlation-length, correlation-longname**

A pair of parameters that consist of a 2-byte integer length and 255-byte string area. A comma separates the parameters. You can provide a unique value to correlate your business process names with Db2 threads in correlation-longname. Db2 displays this correlation token in the output from the DISPLAY THREAD DETAIL command. The CURRENT CLIENT_CORR_TOKEN special register contains the client correlation token. Trailing blanks in correlation-longname are truncated and the length in correlation-length is updated.

These parameters are optional, to specify them you must also specify a value for ws-length, ws-longname. A value of 0 in correlation-length skips processing of correlation-longname.

You can also change the value of the client correlation token with the RRSAF AUTH SIGNON function and the SET_CLIENT_ID function.

**Example of RRSAF AUTH SIGNON calls**

The following table shows a AUTH SIGNON call in each language.

<table>
<thead>
<tr>
<th>Language</th>
<th>Call example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Assembler</td>
<td>CALL DSNRLI,(ASGNONFN,CORRID,ACCTTKN,ACCTINT,PAUTHID,ACEEPTR, SAUTHID,RETCODE,REASCODE, USERID,APPLNAME,WSNAME,XIDPTR)</td>
</tr>
<tr>
<td>C(^1)</td>
<td>fnret=dsnrli(&amp;asgnonfn[0], &amp;corrid[0], &amp;acctkn[0], &amp;acctint[0], &amp;pauthid[0], &amp;aceeptr, &amp;sauthid[0], &amp;retcode, &amp;reascode, &amp;userid[0], &amp;applname[0], &amp;wsname[0], &amp;xidptr);</td>
</tr>
<tr>
<td>COBOL</td>
<td>CALL 'DSNRLI' USING ASGNONFN CORRID ACCTTKN ACCTINT PAUTHID ACEEPTR SAUTHID RETCODE REASCODE USERID APPLNAME WSNAME XIDPTR.</td>
</tr>
<tr>
<td>Fortran</td>
<td>CALL DSNRLI(ASGNONFN,CORRID,ACCTTKN,ACCTINT,PAUTHID,ACEEPTR, SAUTHID,RETCODE,REASCODE,USERID, APPLNAME,WSNAME,XIDPTR)</td>
</tr>
<tr>
<td>PL/I(^1)</td>
<td>CALL DSNRLI(ASGNONFN,CORRID,ACCTTKN,ACCTINT,PAUTHID,ACEEPTR, SAUTHID,RETCODE,REASCODE,USERID, APPLNAME,WSNAME,XIDPTR);</td>
</tr>
</tbody>
</table>

**Note:**

1. For C, C++, and PL/I applications, you must include the appropriate compiler directives, because DSNRLI is an assembler language program. These compiler directives are described in the instructions for invoking RRSAF.

The following example shows an AUTH SIGNON call in C\(^1\) with all parameters passed in. Parameters that are numbers are passed in as integers and strings as character arrays. In this example, if &useridlen is larger than 0, then the value of CURRENT CLIENT_USERID special register is the value that is stored in &userid[0].

fnret = dsnrli(&authsgnfn[0],&corrid[0],&acctkn[0],&acctint[0],&pauthid[0], &aceeptr,&sauthid[0],&retcode,&reascode,&userid[0],&applname[0], &wsname[0],&xidptr,&lacctngid[0],&useridlen,&applidlen, &wsidlen,&lwsid[0],&corrtkidlen,&lcorrtkid[0]);

**Note:**
1. For C applications, you must include the appropriate compiler directives, because DSNRLI is an assembler language program. These compiler directives are described in the instructions for invoking RRSAF.

Related tasks
Invoking the Resource Recovery Services attachment facility
The Resource Recovery Services attachment facility (RRSAF) enables your program to communicate with Db2. Invoke RRSAF as an alternative to invoking CAF or when using stored procedures that run in a WLM-established address space. RRSAF has more capabilities than CAF.

Related reference
SIGNON function for RRSAF
The RRSAF SIGNON function establishes a primary authorization ID and, optionally, one or more secondary authorization IDs for a connection.

CONTEXT SIGNON function for RRSAF
The RRSAF CONTEXT SIGNON function establishes a primary authorization ID and one or more secondary authorization IDs for a connection.

Requirement: Before you invoke CONTEXT SIGNON, you must have called the RRS context services function Set Context Data (CTXSDTA) to store a primary authorization ID and optionally, the address of an ACEE in the context data whose context key you supply as input to CONTEXT SIGNON.

The CONTEXT SIGNON function uses the context key to retrieve the primary authorization ID from data that is associated with the current RRS context. Db2 uses the RRS context services function Retrieve Context Data (CTXRDTA) to retrieve context data that contains the authorization ID and ACEE address. The context data must have the following format:

Version number
A 4-byte area that contains the version number of the context data. Set this area to 1.

Server product name
An 8-byte area that contains the name of the server product that set the context data.

ALET
A 4-byte area that can contain an ALET value. Db2 does not reference this area.

ACEE address
A 4-byte area that contains an ACEE address or 0 if an ACEE is not provided. Db2 requires that the ACEE is in the home address space of the task.

If you pass an ACEE address, the CONTEXT SIGNON function uses the value in ACEEGRPN as the secondary authorization ID if the length of the group name (ACEEGRPL) is not 0.

primary-authid
An 8-byte area that contains the primary authorization ID to be used. If the authorization ID is less than 8 bytes in length, pad it on the right with blank characters to a length of 8 bytes.

If the new primary authorization ID is not different than the current primary authorization ID (which was established when the IDENTIFY function was invoked or at a previous SIGNON invocation), Db2 invokes only the signon exit. If the value has changed, Db2 establishes a new primary authorization ID and new SQL authorization ID and then invokes the signon exit.

Generally, you issue a CONTEXT SIGNON call after an IDENTIFY call and before a CREATE THREAD call. You can also issue a CONTEXT SIGNON call if the application is at a point of consistency, and one of the following conditions is true:

• The value of reuse in the CREATE THREAD call was RESET.

• The value of reuse in the CREATE THREAD call was INITIAL, no held cursors are open, the package or plan is bound with KEEP_DYNAMIC(NO), and all special registers are at their initial state. If open held cursors exist or the package or plan is bound with KEEP_DYNAMIC(YES), a SIGNON call is permitted only if the primary authorization ID has not changed.

The following diagram shows the syntax for the CONTEXT SIGNON function.
**DSNRI CONTEXT SIGNON function**

CALL DSNRI (function, correlation-id, accounting-token, accounting-interval, context-key, retcode, reascode, user, appl, ws, xid, accounting-string)

Parameters point to the following areas:

**function**
An 18-byte area that contains CONTEXT SIGNON followed by four blanks.

**correlation-id**
A 12-byte area in which you can put a Db2 correlation ID. The correlation ID is displayed in Db2 accounting and statistics trace records. You can use the correlation ID to correlate work units. This token appears in output from the DISPLAY THREAD command. If you do not want to specify a correlation ID, fill the 12-byte area with blanks.

**accounting-token**
A 22-byte area in which you can put a value for a Db2 accounting token. This value is displayed in Db2 accounting and statistics trace records in the QWHCTOKN field, which is mapped by DSNDQWHC DSECT. Setting the value of the accounting token sets the value of the CURRENT CLIENT_ACCTNG special register. If accounting-token is less than 22 characters long, you must pad it on the right with blanks to a length of 22 characters. If you do not want to specify an accounting token, fill the 22-byte area with blanks.

You can also change the value of the Db2 accounting token with RRSAF functions SIGNON, AUTH SIGNON, or SET_CLIENT_ID. You can retrieve the Db2 accounting token with the CURRENT CLIENT_ACCTNG special register only if the DDF accounting string is not set.

**accounting-interval**
A 6-byte area that specifies when Db2 writes an accounting record.

If you specify COMMIT in that area, Db2 writes an accounting record each time that the application issues SRRCMIT. This accounting record is written at the end of the second phase of a two-phase commit. If the accounting interval is COMMIT, and an SRRCMIT is issued while a held cursor is open, the accounting interval spans that commit and ends at the next valid accounting interval end point (such as the next SRRCMIT that is issued without open held cursors, application termination, or SIGNON with a new authorization ID).

If you specify any other value, Db2 writes an accounting record when the application terminates or when you call the SIGNON function with a new authorization ID.

**context-key**
A 32-byte area in which you put the context key that you specified when you called the RRS Set Context Data (CTXSDTA) service to save the primary authorization ID and an optional ACEE address.

**retcode**
A 4-byte area in which RRSAF places the return code.

This parameter is optional. If you do not specify retcode, RRSAF places the return code in register 15 and the reason code in register 0.
reascode
A 4-byte area in which RRSAF places the reason code.

This parameter is optional. If you do not specify reascode, RRSAF places the reason code in register 0.

If you specify reascode, you must also specify retcode.

user
A 16-byte area that contains the user ID of the client user. You can use this parameter to provide
the identity of the client user for accounting and monitoring purposes. Db2 displays this user ID in
the output from the DISPLAY THREAD command and in Db2 accounting and statistics trace records.
Setting the user ID sets the value of the CURRENT CLIENT_USERID special register. If user is less than
16 characters long, you must pad it on the right with blanks to a length of 16 characters.

This parameter is optional. If you specify user, you must also specify retcode and reascode. If you do
not specify user, no user ID is associated with the connection.

appl
A 32-byte area that contains the application or transaction name of the user’s application. You can
use this parameter to provide the identity of the client user for accounting and monitoring purposes.
Db2 displays the application name in the output from the DISPLAY THREAD command and in Db2
accounting and statistics trace records. Setting the application name sets the value of the CURRENT
CLIENT_APPLNAME special register. If appl is less than 32 characters long, you must pad it on the
right with blanks to a length of 32 characters.

This parameter is optional. If you specify appl, you must also specify retcode, reascode, and user. If
you do not specify appl, no application or transaction is associated with the connection.

ws
An 18-byte area that contains the workstation name of the client user. You can use this parameter
to provide the identity of the client user for accounting and monitoring purposes. Db2 displays
the workstation name in the output from the DISPLAY THREAD command and in Db2 accounting
and statistics trace records. Setting the workstation name sets the value of the CURRENT
CLIENT_WRKSTNNAME special register. If ws is less than 18 characters long, you must pad it on
the right with blanks to a length of 18 characters.

This parameter is optional. If you specify ws, you must also specify retcode, reascode, user, and appl.
If you do not specify ws, no workstation name is associated with the connection.

You can also change the value of the workstation name with the RRSAF functions SIGNON, AUTH
SIGNON, or SET_CLIENT_ID. You can retrieve the workstation name with the CLIENT_WRKSTNNAME
special register.

xid
A 4-byte area that indicates whether the thread is part of a global transaction. A Db2 thread that is
part of a global transaction can share locks with other Db2 threads that are part of the same global
transaction and can access and modify the same data. A global transaction exists until one of the
threads that is part of the global transaction is committed or rolled back.

You can specify one of the following values for xid:

0
Indicates that the thread is not part of a global transaction. The value 0 must be specified as a
binary integer.

1
Indicates that the thread is part of a global transaction and that Db2 should retrieve the global
transaction ID from RRS. If a global transaction ID already exists for the task, the thread becomes
part of the associated global transaction. Otherwise, RRS generates a new global transaction ID.
The value 1 must be specified as a binary integer. Alternatively, if you want Db2 to return the
generated global transaction ID to the caller, specify an address instead of 1.
address
The 4-byte address of an area into which you enter a global transaction ID for the thread. If the global transaction ID already exists, the thread becomes part of the associated global transaction. Otherwise, RRS creates a new global transaction with the ID that you specify.

Alternatively, if you want Db2 to generate and return a global transaction ID, pass the address of a null global transaction ID by setting the format ID field of the global transaction ID to binary -1 ('FFFFFFF'X). Db2 then replaces the contents of the area with the generated transaction ID. The area at the specified address must be in writable storage and have a length of at least 140 bytes to accommodate the largest possible transaction ID value.

The format of a global transaction ID is shown in the description of the RRSAF SIGNON function.

accounting-string
A one-byte length field and a 255-byte area in which you can put a value for a Db2 accounting string. This value is placed in the DDF accounting trace records in the QMDASQLI field, which is mapped by DSNDQMDA DSECT. If accounting-string is less than 255 characters, you must pad it on the right with zeros to a length of 255 bytes. The entire 256 bytes is mapped by DSNDQMDA DSECT.

This parameter is optional. If you specify this accounting-string, you must also specify retcode, reascode, user, appl and xid. If you do not specify this parameter, no accounting string is associated with the connection.

You can also change the value of the accounting string with RRSAF functions AUTH SIGNON, CONTEXT SIGNON, or SET_CLIENT_ID.

You can retrieve the DDF suffix portion of the accounting string with the CURRENT CLIENT_ACCTNG special register. The suffix portion of accounting-string can contain a maximum of 200 characters. The QMDASFLN field contains the accounting suffix length, and the QMDASUFX field contains the accounting suffix value. If the DDF accounting string is set, you cannot query the accounting token with the CURRENT CLIENT_ACCTNG special register.

Example of RRSAF CONTEXT SIGNON calls
The following table shows a CONTEXT SIGNON call in each language.

<table>
<thead>
<tr>
<th>Language</th>
<th>Call example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Assembler</td>
<td>CALL DSNRLI,(CSGNONFN,CORRID,ACCTTKN,ACCTINT,CTXTKY, RETCODE,REASCODE,USERID,APPLNAME, WSNNAME,XIDPTR)</td>
</tr>
<tr>
<td>C¹</td>
<td>fnxrt=dsnrl1(&amp;csgnonfn[0], &amp;corrid[0], &amp;accttkn[0], &amp;acctint[0], &amp;ctxtkey[0], &amp;retcode, &amp;reascode, &amp;userid[0], &amp;applname[0], &amp;wsname[0], &amp;xidptr);</td>
</tr>
<tr>
<td>COBOL</td>
<td>CALL ‘DSNRLI’ USING CSGNONFN CORRID ACCTTKN ACCTINT CTXTKEY RETCODE REASCODE USERID APPLNAME WSNNAME XIDPTR.</td>
</tr>
<tr>
<td>Fortran</td>
<td>CALL DSNRLI(CSGNONFN,CORRID,ACCTTKN,ACCTINT,CTXTKY, RETCODE,REASCODE, USERID,APPLNAME, WSNNAME,XIDPTR)</td>
</tr>
<tr>
<td>PL/I¹</td>
<td>CALL DSNRLI(CSGNONFN,CORRID,ACCTTKN,ACCTINT,CTXTKY, RETCODE,REASCODE,USERID,APPLNAME, WSNNAME,XIDPTR);</td>
</tr>
</tbody>
</table>

Note:
1. For C, C++, and PL/I applications, you must include the appropriate compiler directives, because DSNRLI is an assembler language program. These compiler directives are described in the instructions for invoking RRSAF.
Related tasks
Invoking the Resource Recovery Services attachment facility
The Resource Recovery Services attachment facility (RRSAF) enables your program to communicate with Db2. Invoke RRSAF as an alternative to invoking CAF or when using stored procedures that run in a WLM-established address space. RRSAF has more capabilities than CAF.

Related reference
SIGNON function for RRSAF
The RRSAF SIGNON function establishes a primary authorization ID and, optionally, one or more secondary authorization IDs for a connection.

SET_ID function for RRSAF
The RRSAF SET_ID function sets a new value for the client program ID that can be used to identify the user. The function then passes this information to Db2 when the next SQL request is processed.

The following diagram shows the syntax of the SET_ID function.

```
DSNRLI SET_ID function
CALL DSNRLI (function, program-id, retcode, reascode)
```

Parameters point to the following areas:

**function**
An 18-byte area that contains SET_ID followed by 12 blanks.

**program-id**
An 80-byte area that contains the caller-provided string to be passed to Db2. If `program-id` is less than 80 characters, you must pad it with blanks on the right to a length of 80 characters.

Db2 places the contents of `program-id` into IFCID 316 records, along with other statistics, so that you can identify which program is associated with a particular SQL statement.

**retcode**
A 4-byte area in which RRSAF places the return code.

This parameter is optional. If you do not specify `retcode` RRSAF places the return code in register 15 and the reason code in register 0.

**reascode**
A 4-byte area in which RRSAF places the reason code.

This parameter is optional. If you do not specify `reascode`, RRSAF places the reason code in register 0.

If you specify `reascode`, you must also specify `retcode`.

Example of RRSAF SET_ID calls
The following table shows a SET_ID call in each language.

<table>
<thead>
<tr>
<th>Language</th>
<th>Call example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Assembler</td>
<td>CALL DSNRLI (SETIDFN, PROGID, RETCODE, REASCODE)</td>
</tr>
<tr>
<td>C¹</td>
<td>fnret=dsnrli(&amp;setidfn[0], &amp;progid[0], &amp;retcode, &amp;reascode);</td>
</tr>
</tbody>
</table>
Table 27. Examples of RRSAF SET_ID calls (continued)

<table>
<thead>
<tr>
<th>Language</th>
<th>Call example</th>
</tr>
</thead>
<tbody>
<tr>
<td>COBOL</td>
<td>CALL 'DSNRLI' USING SETIDFN PROGID RETCODE REASCODE.</td>
</tr>
<tr>
<td>Fortran</td>
<td>CALL DSNRLI(SETIDFN,PROGID,RETCODE,REASCODE)</td>
</tr>
<tr>
<td>PL/I</td>
<td>CALL DSNRLI(SETIDFN,PROGID,RETCODE,REASCODE);</td>
</tr>
</tbody>
</table>

**Note:**

1. For C, C++, and PL/I applications, you must include the appropriate compiler directives, because DSNRLI is an assembler language program. These compiler directives are described in the instructions for invoking RRSAF.

**Related tasks**

- Invoking the Resource Recovery Services attachment facility
- The Resource Recovery Services attachment facility (RRSAF) enables your program to communicate with Db2. Invoke RRSAF as an alternative to invoking CAF or when using stored procedures that run in a WLM-established address space. RRSAF has more capabilities than CAF.

**SET_CLIENT_ID function for RRSAF**

The RRSAF SET_CLIENT_ID function sets new values for the client user ID, the application program name, the workstation name, the accounting token, the DDF client accounting string, the correlation token, and the long name. The function then passes this information to Db2 when the next SQL request is processed.

These values can be used to identify the end user. The calling program defines the contents of these parameters. Db2 places the parameter values in the output from the DISPLAY THREAD command and in Db2 accounting and statistics trace records.

The following diagram shows the syntax of the SET_CLIENT_ID function.

**DSNRLI SET_CLIENT_ID function**

```
CALL DSNRLI ( function , accounting-token , user , appl , ws , retcode , reascode , accounting-string , corr-token , long-name )
```

Parameters point to the following areas:

- **function**
  An 18-byte area that contains SET_CLIENT_ID followed by 5 blanks.

- **accounting-token**
  A 22-byte area in which you can put a value for a Db2 accounting token. This value is placed in the Db2 accounting and statistics trace records in the QWHCTOKN field, which is mapped by DSNDQWHC DSECT. If `accounting-token` is less than 22 characters long, you must pad it on the right with blanks to a length of 22 characters.
You can omit this parameter by specifying a value of 0 in the parameter list.

Alternatively, you can change the value of the Db2 accounting token with the RRSAF functions SIGNON, AUTH SIGNON, or CONTEXT SIGNON. You can retrieve the Db2 accounting token with the CURRENT CLIENT_ACCTNG special register only if the DDF accounting string is not set.

**user**

A 16-byte or 128-byte area that contains the user ID of the client end user. You can use this parameter to provide the identity of the client end user for accounting and monitoring purposes. Db2 places this user ID in the output from the DISPLAY THREAD command and in Db2 accounting and statistics trace records. If `user` is less than 16 characters long, you must pad it on the right with blanks to a length of 16 characters.

You can omit this parameter by specifying a value of 0 in the parameter list.

If the `long-name` parameter is specified, the maximum length of the `user` parameter is 128 bytes. If `user` is less than 128 characters long, you must pad it on the right with blanks to a length of 128 characters.

You can also change the value of the client user ID with the RRSAF functions SIGNON, AUTH SIGNON, or CONTEXT SIGNON. You can retrieve the client user ID with the CLIENT_USERID special register.

**appl**

An 32-byte or 255-byte area that contains the application or transaction name of the end user's application. You can use this parameter to provide the identity of the client end user for accounting and monitoring purposes. Db2 places the application name in the output from the DISPLAY THREAD command and in Db2 accounting and statistics trace records. If `appl` is less than 32 characters, you must pad it on the right with blanks to a length of 32 characters.

You can omit this parameter by specifying a value of 0 in the parameter list.

If the `long-name` parameter is specified, the maximum length of the `appl` parameter is 255 bytes. If `appl` is less than 255 characters long, you must pad it on the right with blanks to a length of 255 characters.

You can also change the value of the application name with the RRSAF functions SIGNON, AUTH SIGNON, or CONTEXT SIGNON. You can retrieve the application name with the CLIENT_APPLNAME special register.

**ws**

An 18-byte or 255-byte area that contains the workstation name of the client end user. You can use this parameter to provide the identity of the client end user for accounting and monitoring purposes. Db2 places this workstation name in the output from the DISPLAY THREAD command and in Db2 accounting and statistics trace records. If `ws` is less than 18 characters, you must pad it on the right with blanks to a length of 18 characters.

You can omit this parameter by specifying a value of 0 in the parameter list.

If the `long-name` parameter is specified, the maximum length of the `ws` parameter is 255 bytes. If `ws` is less than 255 characters long, you must pad it on the right with blanks to a length of 255 characters.

You can also change the value of the workstation name with the RRSAF functions SIGNON, AUTH SIGNON, or CONTEXT SIGNON. You can retrieve the workstation name with the CLIENT_WRKSTNNAME special register.

**retcode**

A 4-byte area in which RRSAF places the return code.

You can omit this parameter by specifying a value of 0 in the parameter list.

This parameter is optional. If you do not specify `retcode`, RRSAF places the return code in register 15 and the reason code in register 0.

**reascode**

A 4-byte area in which RRSAF places the reason code.
You can omit this parameter by specifying a value of 0 in the parameter list. This parameter is optional. If you do not specify \texttt{reascode}, RRSAF places the reason code in register 0. If you specify \texttt{reascode}, you must also specify \texttt{recode}.

\textbf{accounting-string}

A one-byte length field and a 255-byte area in which you can put a value for a Db2 accounting string. This value is placed in the DDF accounting trace records in the QMDASUFX field, which is mapped by DSNDQMDA DSECT. If \texttt{accounting-string} is less than 255 characters, you must pad it on the right with zeros to a length of 255 bytes. The entire 256 bytes is mapped by DSNDQMDA DSECT.

You can omit this parameter by specifying a value of 0 in the parameter list. This parameter is optional. If you specify this \texttt{accounting-string}, you must also specify \texttt{recode}, \texttt{reascode}, \texttt{user}, and \texttt{appl}. If you do not specify this parameter, no accounting string is associated with the connection.

You can also change the value of the accounting string with RRSAF functions \texttt{AUTH SIGNON}, \texttt{CONTEXT SIGNON}, or \texttt{SET_CLIENT_ID}.

You can retrieve the DDF suffix portion of the accounting string with the CURRENT CLIENT\_ACCTNG special register. The suffix portion of \texttt{accounting-string} can contain a maximum of 200 characters. The QMDASFLN field contains the accounting suffix length, and the QMDASUFX field contains the accounting suffix value. If the DDF accounting string is set, you cannot query the accounting token with the CURRENT CLIENT\_ACCTNG special register.

\textbf{corr-token}

An 255-byte area where you specify a client correlation token. You can specify a unique value to correlate your business process within Db2 and your entire business enterprise. The value of \texttt{corr-token} is displayed by the \texttt{DISPLAY THREAD DETAIL} command. The CURRENT CLIENT\_CORR\_TOKEN special register contains the client correlation token. If \texttt{corr-token} is less than 255 characters, you must pad it on the right with blanks to a length of 255 bytes.

You can omit this parameter by specifying a value of 0 in the parameter list. If you specify \texttt{corr-token} you must also specify \texttt{long-name}.

You can also change the value of the client correlation token with the RRSAF \texttt{SIGNON} function.

\textbf{long-name}

An 8-byte area that contains the value LONGNAME.

This optional parameter is used to indicate to the RRSAF function that the input parameters \texttt{user}, \texttt{appl}, \texttt{ws}, \texttt{accounting-string}, and \texttt{corr-token} can accept longer lengths. You cannot selectively associate the \texttt{long-name} parameter with any individual parameter.

\textbf{Example of RRSAF SET\_CLIENT\_ID calls}

The following table shows a SET\_CLIENT\_ID call in each language.

<table>
<thead>
<tr>
<th>Language</th>
<th>Call example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Assembler</td>
<td>CALL DSNRLI,(SECLIDFN,ACCT,USER,APPL,WS,RETCODE,REASCODE,ACCOUNTINGSTRING,CORRTOKEN,LONGNAME)</td>
</tr>
<tr>
<td>C</td>
<td>fnret=dsnrli(&amp;seclidfn[0], &amp;acct[0], &amp;user[0], &amp;appl[0], &amp;ws[0], &amp;retcode, &amp;reascode, &amp;accountingstring[0], &amp;corrtoken[0], &amp;longname[0]);</td>
</tr>
<tr>
<td>COBOL</td>
<td>CALL 'DSNRLI' USING SECLIDFN ACCT USER APPL WS RETCODE REASCODE ACCOUNTING-STRING CORR-TOKEN LONG-NAME.</td>
</tr>
</tbody>
</table>
Table 28. Examples of RRSAF SET_CLIENT_ID calls (continued)

<table>
<thead>
<tr>
<th>Language</th>
<th>Call example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fortran</td>
<td>CALL  DSNRLI(SECLIDFN,ACCT,USER,APPL,WS,RETCODE,REASCODE,ACCOUNTINGSTRING,CORRTOKEN,LONGNAME)</td>
</tr>
<tr>
<td>PL/I</td>
<td>CALL  DSNRLI(SECLIDFN,ACCT,USER,APPL,WS,RETCODE,REASCODE,ACCOUNTINGSTRING,CORRTOKEN,LONGNAME);</td>
</tr>
</tbody>
</table>

**Note:**
1. For C, C++, and PL/I applications, you must include the appropriate compiler directives, because DSNRLI is an assembler language program. These compiler directives are described in the instructions for invoking RRSAF.

**Related tasks**
- Invoking the Resource Recovery Services attachment facility
  - The Resource Recovery Services attachment facility (RRSAF) enables your program to communicate with Db2. Invoke RRSAF as an alternative to invoking CAF or when using stored procedures that run in a WLM-established address space. RRSAF has more capabilities than CAF.

**SET_REPLICATION function for RRSAF**

The RRSAF SET_REPLICATION function enables an APF authorized program to identify to Db2 as a replication program.

Calling the SET_REPLICATION function is optional. If you do not call it, Db2 treats the application normally. The SET_REPLICATION function allows the application to perform insert, update, and delete operations then the tablespace or database is started access RREPL.

The following diagram shows the syntax for the SET REPLICATION function.

```
DSNRLI SET_REPLICATION function

CALL DSNRLI (function, retcode, reascode)
```

Parameters point to the following areas:

- **function**
  - An 18-byte area that contains SET_REPLICATION.

- **retcode**
  - A 4-byte area in which RRSAF places the return code.
  - This parameter is optional. If you do not specify retcode, RRSAF places the return code in register 15 and the reason code in register 0.

- **reascode**
  - A 4-byte area in which RRSAF places a reason code.
  - This parameter is optional. If you do not specify reascode, RRSAF places the reason code in register 0.
  - If you specify reascode, you must also specify retcode.
The Resource Recovery Services attachment facility (RRSAF) enables your program to communicate with Db2. Invoke RRSAF as an alternative to invoking CAF or when using stored procedures that run in a WLM-established address space. RRSAF has more capabilities than CAF.

CREATE THREAD function for RRSAF

The RRSAF CREATE THREAD function allocates the Db2 resources that are required for an application to issue SQL or IFI requests. This function must complete before the application can execute SQL statements or IFI requests.

The following diagram shows the syntax of the CREATE THREAD function.

```
DSNRLI CREATE THREAD function
   CALL DSNRLI (function, plan, collection, reuse, retcode, reascode, pklistptr)
```

Parameters point to the following areas:

**function**
An 18-byte area that contains CREATE THREAD followed by five blanks.

**plan**
An 8-byte Db2 plan name. RRSAF allocates the named plan.

If you provide a collection name instead of a plan name, specify the question mark character (?) in the first byte of this field. Db2 then allocates a special plan named ?RRSAF and uses the value that you specify for collection. When Db2 allocates a plan named ?RRSAF, Db2 checks authorization to execute the package in the same way as it checks authorization to execute a package from a requester other than Db2 for z/OS.

If you do not provide a collection name in the collection field, you must enter a valid plan name in this field.

**collection**
An 18-byte area in which you enter a collection name. Db2 uses the collection names to locate a package that is associated with the first SQL statement in the program.

When you provide a collection name and put the question mark character (?) in the plan field, Db2 allocates a plan named ?RRSAF and a package list that contains the following two entries:

- The specified collection name.
- An entry that contains * for the location, collection name, and package name. (This entry lets the application access remote locations and access packages in collections other than the default collection that is specified at create thread time.)

The application can use the SET CURRENT PACKAGESET statement to change the collection ID that Db2 uses to locate a package.

If you provide a plan name in the plan field, Db2 ignores the value in the collection field.

**reuse**
An 8-byte area that controls the action that Db2 takes if a SIGNON call is issued after a CREATE THREAD call. Specify one of the following values in this field:

**RESET**
Releases any held cursors and reinitializes the special registers.
**INITIAL**
Does not allow the SIGNON call

This parameter is required. If the 8-byte area does not contain either RESET or INITIAL, the default value is INITIAL.

**retcode**
A 4-byte area in which RRSAF places the return code.

This parameter is optional. If you do not specify retcode, RRSAF places the return code in register 15 and the reason code in register 0.

**reascode**
A 4-byte area in which RRSAF places the reason code.

This parameter is optional. If you do not specify reascode, RRSAF places the reason code in register 0.

If you specify reascode, you must also specify retcode.

**pklistptr**
A 4-byte field that contains a pointer to a user-supplied data area that contains a list of collection IDs. A collection ID is an SQL identifier of 1 to 128 letters, digits, or the underscore character that identifies a collection of packages. The length of the data area is a maximum of 2050 bytes. The data area contains a 2-byte length field, followed by up to 2048 bytes of collection ID entries, separated by commas.

When you specify pklistptr and the question mark character (?) in the plan field, Db2 allocates a special plan named ?RRSAF and a package list that contains the following entries:

- The collection names that you specify in the data area to which pklistptr points
- An entry that contains * for the location, collection ID, and package name

If you also specify collection, Db2 ignores that value.

Each collection entry must be of the form collection-ID.*, *.collection-ID*, or **. collection-ID and must follow the naming conventions for a collection ID, as described in the description of the BIND and REBIND options.

Db2 uses the collection names to locate a package that is associated with the first SQL statement in the program. The entry that contains **.* lets the application access remote locations and access packages in collections other than the default collection that is specified at create thread time.

The application can use the SET CURRENT PACKAGSET statement to change the collection ID that Db2 uses to locate a package.

This parameter is optional. If you specify this parameter, you must also specify retcode and reascode.

If you provide a plan name in the plan field, Db2 ignores the pklistptr value.

**Recommendation:** Using a package list can have a negative impact on performance. For better performance, specify a short package list.

**Example of RRSAF CREATE THREAD calls**
The following table shows a CREATE THREAD call in each language.

<table>
<thead>
<tr>
<th>Language</th>
<th>Call example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Assembler</td>
<td>CALL DSNRLI,(CRTHRDFN,PLAN,COLLID,REUSE,RETCODE,REASCODE,PKLISTPTR)</td>
</tr>
<tr>
<td>C¹</td>
<td>fnret=dsnrli(&amp;crthrdfn[0], &amp;plan[0], &amp;collid[0], &amp;reuse[0], &amp;retcode, &amp;reascode, &amp;pklistptr);</td>
</tr>
</tbody>
</table>

¹ Db2 12 for z/OS: Application Programming and SQL Guide
Table 29. Examples of RRSAF CREATE THREAD calls (continued)

<table>
<thead>
<tr>
<th>Language</th>
<th>Call example</th>
</tr>
</thead>
<tbody>
<tr>
<td>COBOL</td>
<td>CALL 'DSNRLI' USING CRTHRDFN PLAN COLLID REUSE RETCODE REASCODE PKLSTPTR.</td>
</tr>
<tr>
<td>Fortran</td>
<td>CALL DSNRLI(CRTHRDFN,PLAN,COLLID,REUSE,RETCODE,REASCODE,PKLSTPTR)</td>
</tr>
<tr>
<td>PL/I1</td>
<td>CALL DSNRLI(CRTHRDFN,PLAN,COLLID,REUSE,RETCODE,REASCODE,PKLSTPTR);</td>
</tr>
</tbody>
</table>

Note:
1. For C, C++, and PL/I applications, you must include the appropriate compiler directives, because DSNRLI is an assembler language program. These compiler directives are described in the instructions for invoking RRSAF.

Related tasks
Invoking the Resource Recovery Services attachment facility
The Resource Recovery Services attachment facility (RRSAF) enables your program to communicate with Db2. Invoke RRSAF as an alternative to invoking CAF or when using stored procedures that run in a WLM-established address space. RRSAF has more capabilities than CAF.

Authorizing plan or package access through applications (Managing Security)

Related reference
BIND and REBIND options for packages, plans, and services (Db2 Commands)

TERMINATE THREAD function for RRSAF
The RRSAF TERMINATE THREAD function deallocates Db2 resources that are associated with a plan and were previously allocated for an application by the CREATE THREAD function. You can then use the CREATE THREAD function to allocate another plan with the same connection.

If you call the TERMINATE THREAD function and the application is not at a point of consistency, RRSAF returns reason code X'00C12211'.

The following diagram shows the syntax of the TERMINATE THREAD function.

![DSNRLI TERMINATE THREAD function diagram]

Parameters point to the following areas:

function
An 18-byte area the contains TERMINATE THREAD followed by two blanks.

retcode
A 4-byte area in which RRSAF places the return code.

This parameter is optional. If you do not specify retcode, RRSAF places the return code in register 15 and the reason code in register 0.

reascode
A 4-byte area in which RRSAF places the reason code.

This parameter is optional. If you do not specify reascode, RRSAF places the reason code in register 0.
If you specify `reascode`, you must also specify `retcode`.

**Example of RRSAF TERMINATE THREAD calls**

The following table shows a TERMINATE THREAD call in each language.

<table>
<thead>
<tr>
<th>Language</th>
<th>Call example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Assembler</td>
<td><code>CALL DSNRLI,(TRMTHDFN,RETCODE,REASCODE)</code></td>
</tr>
<tr>
<td>C¹</td>
<td><code>fnret=dsnrli(&amp;trmthdfn[0], &amp;retcode, &amp;reascode);</code></td>
</tr>
<tr>
<td>COBOL</td>
<td><code>CALL 'DSNRLI' USING TRMTHDFN RETCODE REASCODE.</code></td>
</tr>
<tr>
<td>Fortran</td>
<td><code>CALL DSNRLI(TRMTHDFN,RETCODE,REASCODE)</code></td>
</tr>
<tr>
<td>PL/I¹</td>
<td><code>CALL DSNRLI(TRMTHDFN,RETCODE,REASCODE);</code></td>
</tr>
</tbody>
</table>

**Note:**

1. For C, C++, and PL/I applications, you must include the appropriate compiler directives, because DSNRLI is an assembler language program. These compiler directives are described in the instructions for invoking RRSAF.

**Related tasks**

Invoking the Resource Recovery Services attachment facility

The Resource Recovery Services attachment facility (RRSAF) enables your program to communicate with Db2. Invoke RRSAF as an alternative to invoking CAF or when using stored procedures that run in a WLM-established address space. RRSAF has more capabilities than CAF.

**TERMINATE IDENTIFY function for RRSAF**

The RRSAF TERMINATE IDENTIFY function terminates a connection to Db2. Calling the TERMINATE IDENTIFY function is optional. If you do not call it, Db2 performs the same functions when the task terminates.

If Db2 terminates, the application must issue TERMINATE IDENTIFY to reset the RRSAF control blocks. This action ensures that future connection requests from the task are successful when Db2 restarts.

The TERMINATE IDENTIFY function removes the calling task's connection to Db2. If no other task in the address space has an active connection to Db2, Db2 also deletes the control block structures that were created for the address space and removes the cross-memory authorization.

If the application is not at a point of consistency when you call the TERMINATE IDENTIFY function, RRSAF returns reason code `X'00C12211'`.

If the application allocated a plan, and you call the TERMINATE IDENTIFY function without first calling the TERMINATE THREAD function, Db2 deallocates the plan before terminating the connection.

The following diagram shows the syntax of the TERMINATE IDENTIFY function.
### DSNRLI TERMINATE IDENTIFY function

```c
CALL DSNRLI (function, retcode, reascode)
```

Parameters point to the following areas:

**function**
- An 18-byte area that contains TERMINATE IDENTIFY.

**retcode**
- A 4-byte area in which RRSAF places the return code.
  - This parameter is optional. If you do not specify `retcode`, RRSAF places the return code in register 15 and the reason code in register 0.

**reascode**
- A 4-byte area in which RRSAF places the reason code.
  - This parameter is optional. If you do not specify `reascode`, RRSAF places the reason code in register 0.
  - If you specify `reascode`, you must also specify `retcode`.

### Example of RRSAF TERMINATE IDENTIFY calls

The following table shows a TERMINATE IDENTIFY call in each language.

<table>
<thead>
<tr>
<th>Language</th>
<th>Call example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Assembler</td>
<td>CALL DSNRLI (TMIDFYFN, RETCODE, REASCODE)</td>
</tr>
<tr>
<td>C1</td>
<td>fnret=dsnrli(&amp;tmidfyfn[0], &amp;retcode, &amp;reascode);</td>
</tr>
<tr>
<td>COBOL</td>
<td>CALL ‘DSNRLI’ USING TMIDFYFN RETCODE REASCODE.</td>
</tr>
<tr>
<td>Fortran</td>
<td>CALL DSNRLI(TMIDFYFN, RETCODE, REASCODE)</td>
</tr>
<tr>
<td>PL/I1</td>
<td>CALL DSNRLI(TMIDFYFN, RETCODE, REASCODE);</td>
</tr>
</tbody>
</table>

**Note:**
1. For C, C++, and PL/I applications, you must include the appropriate compiler directives, because DSNRLI is an assembler language program. These compiler directives are described in the instructions for invoking RRSAF.

### Related tasks
- Invoking the Resource Recovery Services attachment facility
The Resource Recovery Services attachment facility (RRSAF) enables your program to communicate with Db2. Invoke RRSAF as an alternative to invoking CAF or when using stored procedures that run in a WLM-established address space. RRSAF has more capabilities than CAF.

**TRANSLATE function for RRSAF**

The RRSAF TRANSLATE function converts a hexadecimal reason code for a Db2 error into a signed integer SQL code and a printable error message. The SQL code and message text are placed in the SQLCODE and SQLSTATE host variables or related fields of the SQLCA.

Consider the following rules and recommendations about when to use and not use the TRANSLATE function:

- You cannot call the TRANSLATE function from the Fortran language.
- Call the TRANSLATE function only after a successful IDENTIFY operation. For errors that occur during SQL or IFI requests, the TRANSLATE function performs automatically.
- The TRANSLATE function translates codes that begin with X'00F3', but it does not translate RRSAF reason codes that begin with X'00C1'.

If you receive error reason code X'00F30040' (resource unavailable) after an OPEN request, the TRANSLATE function returns the name of the unavailable database object in the last 44 characters of the SQLERRM field.

If the TRANSLATE function does not recognize the error reason code, it returns SQLCODE -924 (SQLSTATE '58006') and places a printable copy of the original Db2 function code and the return and error reason codes in the SQLERRM field. The contents of registers 0 and 15 do not change, unless TRANSLATE fails. In this case, register 0 is set to X'00C12204', and register 15 is set to 200.

The following diagram shows the syntax of the TRANSLATE function.

```
DSNRLI TRANSLATE function
  CALL DSNRLI ( function, sqlca , retcode , reascode )
```

Parameters point to the following areas:

- **function**
  An 18-byte area that contains the word TRANSLATE followed by nine blanks.

- **sqlca**
  The program's SQL communication area (SQLCA).

- **retcode**
  A 4-byte area in which RRSAF places the return code.
  This parameter is optional. If you do not specify retcode, RRSAF places the return code in register 15 and the reason code in register 0.

- **reascode**
  A 4-byte area in which RRSAF places the reason code.
  This parameter is optional. If you do not specify reascode, RRSAF places the reason code in register 0.
  If you specify reascode, you must also specify retcode.

**Example of RRSAF TRANSLATE calls**

The following table shows a TRANSLATE call in each language.
### Table 32. Examples of RRSAF TRANSLATE calls

<table>
<thead>
<tr>
<th>Language</th>
<th>Call example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Assembler</td>
<td>CALL DSNRLI, (XLATFN, SQLCA, RETCODE, REASCODE)</td>
</tr>
<tr>
<td>C</td>
<td>fnret=dsnrli(&amp;connfn[0], &amp;sqlca, &amp;retcode, &amp;reascode);</td>
</tr>
<tr>
<td>COBOL</td>
<td>CALL 'DSNRLI' USING XLATFN SQLCA RETCODE REASCODE.</td>
</tr>
<tr>
<td>PL/I</td>
<td>CALL DSNRLI(XLATFN,SQLCA,RETCODE,REASCODE);</td>
</tr>
</tbody>
</table>

**Note:**
1. For C, C++, and PL/I applications, you must include the appropriate compiler directives, because DSNRLI is an assembler language program. These compiler directives are described in the instructions for invoking RRSAF.

**Related tasks**
- Invoking the Resource Recovery Services attachment facility
  The Resource Recovery Services attachment facility (RRSAF) enables your program to communicate with Db2. Invoke RRSAF as an alternative to invoking CAF or when using stored procedures that run in a WLM-established address space. RRSAF has more capabilities than CAF.

### FIND_DB2_SYSTEMS function for RRSAF

The RRSAF FIND_DB2_SYSTEMS function identifies all active Db2 subsystems on a z/OS LPAR. The following diagram shows the syntax of the FIND_DB2_SYSTEMS function.

![DSNRLI FIND_DB2_SYSTEMS function](image)

Parameters point to the following areas:

- **function**
  An 18-byte area that contains FIND_DB2_SYSTEMS followed by two blanks.

- **ssnma**
  A storage area for an array of 4-byte character strings into which RRSAF places the names of all the Db2 subsystems (SSIDs) that are defined for the current LPAR. You must provide the storage area. If the array is larger than the number of Db2 subsystems, RRSAF returns the value ' ' (four blanks) in all unused array members.

- **activea**
  A storage area for an array of 4-byte values into which RRSAF returns an indication of whether a defined subsystem is active. Each value is represented as a fixed 31-bit integer. The value 1 means that the subsystem is active. The value 0 means that the subsystem is not active. The size of this array must be the same as the size of the ssnma array. If the array is larger than the number of Db2 subsystems, RRSAF returns the value -1 in all unused array members.

The information in the activea array is the information that is available at the point in time that you requested it and might change at any time.
**arraysz**
A 4-byte area, represented as a fixed 31-bit integer, that specifies the number of entries for the `ssnma` and `activea` arrays. If the number of array entries is insufficient to contain all of the subsystems defined on the current LPAR, RRSAF uses all available entries and returns return code 4.

**retcode**
A 4-byte area in which RRSAF is to place the return code for this call to the FIND_DB2_SYSTEMS function.
This parameter is optional. If you do not set `retcode`, RRSAF places the return code in register 15 and the reason code in register 0.

**reascode**
A 4-byte area in which RRSAF is to place the reason code for this call to the FIND_DB2_SYSTEMS function.
This parameter is optional. If you do not specify `reascode`, RRSAF places the reason code in register 0.

**Example values that the FIND_DB2_SYSTEMS function returns**
Assume that two subsystems are defined on the current LPAR. Subsystem DB2A is active, and subsystem DB2B is stopped. Suppose that you invoke RRSAF with the function FIND_DB2_SYSTEMS and a value of 3 for `arraysz`. The `ssnma` array and `activea` array are set to the following values:

<table>
<thead>
<tr>
<th>Array element number</th>
<th>Values in <code>ssnma</code> array</th>
<th>Values in <code>activea</code> array</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>DB2A</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>DB2B</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>(four blanks)</td>
<td>-1</td>
</tr>
</tbody>
</table>

**Related tasks**
Invoking the Resource Recovery Services attachment facility
The Resource Recovery Services attachment facility (RRSAF) enables your program to communicate with Db2. Invoke RRSAF as an alternative to invoking CAF or when using stored procedures that run in a WLM-established address space. RRSAF has more capabilities than CAF.

**RRSAF return codes and reason codes**
If you specify return code and reason code parameters in an Resource Recovery Services attachment facility (RRSAF) function call, RRSAF returns the return code and reason code in those parameters. If you do not specify those parameters or implicitly invoke RRSAF, RRSAF puts the return code in register 15 and the reason code in register 0.

When the reason code begins with X'00F3', except for X'00F30006', you can use the RRSAF TRANSLATE function to obtain error message text that can be printed and displayed.

For SQL calls, RRSAF returns standard SQL return codes in the SQLCA. RRSAF returns IFI return codes and reason codes in the instrumentation facility communication area (IFCA).

The following table lists the RRSAF return codes.

<table>
<thead>
<tr>
<th>Return code</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>The call completed successfully.</td>
</tr>
<tr>
<td>4</td>
<td>Status information is available. See the reason code for details.</td>
</tr>
<tr>
<td>&gt;4</td>
<td>The call failed. See the reason code for details.</td>
</tr>
</tbody>
</table>
Related reference
TRANSLATE function for RRSAF
The RRSAF TRANSLATE function converts a hexadecimal reason code for a Db2 error into a signed integer SQL code and a printable error message. The SQL code and message text are placed in the SQLCODE and SQLSTATE host variables or related fields of the SQLCA.

Sample RRSAF scenarios
One or more tasks can use Resource Recovery Services attachment facility (RRSAF) to connect to Db2. This connection can be made either implicitly or explicitly. For explicit connections, a task calls one or more of the RRSAF connection functions.

A single task
The following example pseudocode illustrates a single task running in an address space that explicitly connects to Db2 through RRSAF. z/OS RRS controls commit processing when the task terminates normally.

```
IDENTIFY
SIGNON
CREATE THREAD
SQL or IFI
⋮
TERMINATE IDENTIFY
```

Multiple tasks
In the following scenario, multiple tasks in an address space explicitly connect to Db2 through RRSAF.

Task 1 executes no SQL statements and makes no IFI calls. Its purpose is to monitor Db2 termination and startup ECBs and to check the Db2 release level.

```
TASK 1  TASK 2  TASK 3  TASK n
IDENTIFY  IDENTIFY  IDENTIFY  IDENTIFY
SIGNON    SIGNON    SIGNON    SIGNON
CREATE THREAD  CREATE THREAD  CREATE THREAD  CREATE THREAD
SQL       SQL       SQL       SQL
⋮         ...
SRRCMIT   SRRCMIT   SRRCMIT   SRRCMIT
SQL       SQL       SQL       SQL
⋮         ...
TERMINATE IDENTIFY
```

Reusing a Db2 thread
The following example pseudocode shows a Db2 thread that is reused by another user at a point of consistency. When the application calls the SIGNON function for user B, Db2 reuses the plan that is allocated by the CREATE THREAD function for user A.

```
IDENTIFY
SIGNON user A
CREATE THREAD
SQL
⋮
SRRCMIT
SIGNON user B
SQL
⋮
SRRCMIT
```
Switching Db2 threads between tasks

The following scenario shows how you can switch the threads for four users (A, B, C, and D) among two tasks (1 and 2).

<table>
<thead>
<tr>
<th>Task 1</th>
<th>Task 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>CTXBEGC (create context a)</td>
<td>CTXBEGC (create context b)</td>
</tr>
<tr>
<td>CTXSWCH(a,0)</td>
<td>CTXSWCH(b,0)</td>
</tr>
<tr>
<td>IDENTIFY</td>
<td>IDENTIFY</td>
</tr>
<tr>
<td>SIGNON user A</td>
<td>SIGNON user B</td>
</tr>
<tr>
<td>CREATE THREAD (Plan A)</td>
<td>CREATE THREAD (plan B)</td>
</tr>
<tr>
<td>SQL</td>
<td>SQL</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>CTXSWCH(0,a)</td>
<td>CTXSWCH(0,b)</td>
</tr>
<tr>
<td>CTXBEGC (create context c)</td>
<td>CTXBEGC (create context d)</td>
</tr>
<tr>
<td>CTXSWCH(c,0)</td>
<td>CTXSWCH(d,0)</td>
</tr>
<tr>
<td>IDENTIFY</td>
<td>IDENTIFY</td>
</tr>
<tr>
<td>SIGNON user C</td>
<td>SIGNON user D</td>
</tr>
<tr>
<td>CREATE THREAD (plan C)</td>
<td>CREATE THREAD (plan D)</td>
</tr>
<tr>
<td>SQL</td>
<td>SQL</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>CTXSWCH(b,c)</td>
<td>CTXSWCH(0,d)</td>
</tr>
<tr>
<td>SQL (plan B)</td>
<td>CTXSWCH(0,d)</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

The applications perform the following steps:

• Task 1 creates context a, switches contexts so that context a is active for task 1, and calls the IDENTIFY function to initialize a connection to a subsystem. A task must always call the IDENTIFY function before a context switch can occur. After the IDENTIFY operation is complete, task 1 allocates a thread for user A, and performs SQL operations.

At the same time, task 2 creates context b, switches contexts so that context b is active for task 2, calls the IDENTIFY function to initialize a connection to the subsystem, allocates a thread for user B, and performs SQL operations.

When the SQL operations complete, both tasks perform RRS context switch operations. Those operations disconnect each Db2 thread from the task under which it was running.

• Task 1 then creates context c, calls the IDENTIFY function to initialize a connection to the subsystem, switches contexts so that context c is active for task 1, allocates a thread for user C, and performs SQL operations for user C.

Task 2 does the same operations for user D.

• When the SQL operations for user C complete, task 1 performs a context switch operation to perform the following actions:
  – Switch the thread for user C away from task 1.
  – Switch the thread for user B to task 1.

For a context switch operation to associate a task with a Db2 thread, the Db2 thread must have previously performed an IDENTIFY operation. Therefore, before the thread for user B can be associated with task 1, task 1 must have performed an IDENTIFY operation.

• Task 2 performs two context switch operations to perform the following actions:
  – Disassociate the thread for user D from task 2.
  – Associate the thread for user A with task 2.
Program examples for RRSAF

The Resource Recovery Services attachment facility (RRSAF) enables programs to communicate with Db2. You can use RRSAF as an alternative to CAF.

Example JCL for invoking RRSAF

The following sample JCL shows how to use RRSAF in a batch environment. The DSNRRSAF DD statement starts the RRSAF trace. Use that DD statement only if you are diagnosing a problem.

```
//jobname JOB z/OS_jobcard_information
//RRSJC1 EXEC PGM=RRS_application_program
//STEPLIB DD DSN=application_load_library
// DD DSN=DB2_load_library

//SYSPRINT DD SYSOUT=* 
//DSNRRSAF DD DUMMY 
//SYSUDUMP DD SYSOUT=* 
```

Example of loading and deleting the RRSAF language interface

The following code segment shows how an application loads entry points DSNRLI and DSNHLIR of the RRSAF language interface. Storing the entry points in variables LIRLI and LISQL ensures that the application loads the entry points only once. Delete the loaded modules when the application no longer needs to access Db2.

```
******************************************************************************
* GET LANGUAGE INTERFACE ENTRY ADDRESSES
******************************************************************************
LOAD EP=DSNRLI Load the RRSAF service request EP
ST R0,LIRLI Save this for RRSAF service requests
LOAD EP=DSNHLIR Load the RRSAF SQL call Entry Point
ST R0,LISQL Save this for SQL calls
*
* Insert connection service requests and SQL calls here
*
DELETE EP=DSNRLI Correctly maintain use count
DELETE EP=DSNHLIR Correctly maintain use count
```

Example of using dummy entry point DSNHLI for RRSAF

Each of the Db2 attachment facilities contains an entry point named DSNHLI. When you use RRSAF but do not specify the ATTACH(RRSAF) precompiler option, the precompiler generates BALR instructions to DSNHLI for SQL statements in your program. To find the correct DSNHLI entry point without including DSNRLI in your load module, code a subroutine, with entry point DSNHLI, that passes control to entry point DSNHLIR in the DSNRLI module. DSNHLIR is unique to DSNRLI and is at the same location as DSNHLI in DSNRLI. DSNRLI uses 31-bit addressing. If the application that calls this intermediate subroutine uses 24-bit addressing, the intermediate subroutine must account for the difference.

In the following example, LISQL is addressable because the calling CSECT used the same register 12 as CSECT DSNHLI. Your application must also establish addressability to LISQL.

```
******************************************************************************
* Subroutine DSNHLI intercepts calls to LI EP=DSNHLI
******************************************************************************
DS D0 
DSHLI CSECT Begin CSECT
STM R14,R12,12(R13) Prologue
LA R15,SAVEHLI Get save area address
ST R13,4(R15) Chain the save areas
ST R15,8(R13) Chain the save areas
LR R13,R15 Put save area address in R13
L R15,LISQL Get the address of real DSNHLI
BASSM R14,R15 Branch to DSNRLI to do an SQL call
*
DSNRLI is in 31-bit mode, so use
*
BASSM to assure that the addressing
*
mode is preserved.
*
L R13,4(R13) Restore R13 (caller's save area addr)
```
Example of connecting to Db2 with RRSAF

This example uses the variables that are declared in the following code.

*************** VARIABLES SET BY APPLICATION ****************************
LIRLI       DS    F                  DSNRLI entry point address
LISQL       DS    F                  DSNHLIR entry point address
SSNM        DS    CL4                DB2 subsystem name for IDENTIFY
CORRID      DS    CL12               Correlation ID for SIGNON
ACCTTKN     DS    CL22               Accounting token for SIGNON
ACCTINT     DS    CL6                Accounting interval for SIGNON
PLAN        DS    CL8                DB2 plan name for CREATE THREAD
*            PLAN contains a plan name, not used.
COLLID      DS    CL18               Collection ID for CREATE THREAD
REUSE       DS    CL8                Controls SIGNON after CREATE THREAD
CONTROL     DS    CL8                Action that application takes based
*            on return code from RRSAF

*************** VARIABLES SET BY DB2 ***********************************
STARTECB    DS    F                  DB2 startup ECB
TERMECB     DS    F                  DB2 termination ECB
EIBPTR      DS    F                  Address of environment info block
RIBPTR      DS    F                  Address of release info block

****************** CONSTANTS ******************************************
CONTINUE DC    CL8'CONTINUE'      CONTROL value: Everything OK
IDFYFN      DC    CL18'IDENTIFY          ' Name of RRSAF service
SGNONFN     DC    CL18'SIGNON            ' Name of RRSAF service
CRTHRDFN    DC    CL18'CREATE THREAD     ' Name of RRSAF service
TRMTHDFN    DC    CL18'TERMINATE THREAD  ' Name of RRSAF service
TMIDFYFN    DC    CL18'TERMINATE IDENTIFY' Name of RRSAF service

*************************** SQLCA and RIB *******************************
EXEC SQL INCLUDE SQLCA
DSNDRIB                  Map the DB2 Release Information Block

************** Parameter list for RRSAF calls ****************************
RRSAFCLL CALL  ,(*,*,*,*,*,*,*,*),VL,MF=L

The following example code shows how to issue requests for the RRSAF functions IDENTIFY, SIGNON, CREATE THREAD, TERMINATE THREAD, and TERMINATE IDENTIFY. This example does not show a task that waits on the Db2 termination ECB. You can code such a task and use the z/OS WAIT macro to monitor the ECB. The task that waits on the termination ECB should detach the sample code if the termination ECB is posted. That task can also wait on the Db2 startup ECB. This example waits on the startup ECB at its own task level.

*********************** IDENTIFY *******************************
L     R15,LIRLI          Get the Language Interface address
*     CALL (15),(IDFYFN,SSNM,RIBPTR,EIBPTR,TERMECB,STARTECB),VL,MF=X
*     (E,RRSAFCLL)
*     BAL   R14,CHEKCODE       Call a routine (not shown) to check
*                             return and reason codes
*     CLC   CONTROL,CONTINUE   Is everything still OK
*     BNE   EXIT               If CONTROL not 'CONTINUE', stop loop
*     USING R8,RIB             Prepare to access the RIB
*     L     R8,RIBPTR          Access RIB to get DB2 release level
*     L     RIBREL,RIBR999     DB2 V10 or later?
*     BE    USERLX             If RIBREL = '999', use RIBRELX
*     WRITE 'The current DB2 release level is' RIBREL
*     B     SIGNON             Continue with signon USERLX
*     WRITE 'The current DB2 release level is' RIBRELX

******************** SIGNON ****************************************
L     R15,LIRLI          Get the Language Interface address
*     CALL (15),(SGNONFN,CORRID,ACCTTKN,ACCINT),VL,MF=(E,RRSAFCLL)
*     BAL   R14,CHEKCODE       Check the return and reason codes

*********************** CREATE THREAD ****************************
L     R15,LIRLI          Get the Language Interface address
*     CALL (15),(CRTHRDFN,PLAN,COLLID,REUSE),VL,MF=(E,RRSAFCLL)
*     BAL   R14,CHEKCODE       Check the return and reason codes

************ SQL ***************************************
*              Insert your SQL calls here.  The DB2 Precompiler
*              generates calls to entry point DSNHLI.  You should
*              code a dummy entry point of that name to intercept
*              all SQL calls.  A dummy DSNHLI is shown in the following
*              section.

*************** TERMINATE THREAD ******************************
CLC   CONTROL,CONTINUE   Is everything still OK?
Universal language interface (DSNULI)

The universal language interface (DSNULI) subcomponent determines the runtime environment and dynamically loads and branches to the appropriate language interface module.

The following figure shows the general structure of DSNULI and a program that uses it:
The Db2 load module, DSNULI, is the Universal Language Interface module. DSNULI has no aliases. DSNULI has the following entry points:

**DSNALI**
For explicit Db2 Call Attach Facility connection service requests
DSNRLI
For explicit Db2 Resource Recovery Services Attach Facility connection service requests

DSNCLI
For link-editing with CICS

DSNHLI
For generic SQL calls from applications that are designed to run in any environment.

DSNHLI2
For explicit SQL calls by way of the Call Attachment Facility

DSNHLIR
For explicit SQL calls by way of the Resource Recovery Services Attachment Facility

DSNWLI
For generic IFI calls from applications that are designed to run in any environment.

DSNWLI2
For explicit IFI calls by way of the Call Attachment Facility.

DSNWLIR
For explicit IFI call by way of the Resource Recovery Services Attachment Facility

PLITDLI
For PL/I specific IMS database access

DFSPLI
For PL/I specific IMS database access

CBLTDLI
For COBOL specific IMS database access

DFSCOBOL
For COBOL specific IMS database access

ASMTDLI
For assembler specific IMS database access

DFSSASM
For assembler specific IMS database access

DSNUIL dynamically loads and branches to the appropriate language interface module, which is based on the entry point name (for attachment-specific entry points), or based on the current environment (for the generic entry points DSNHLI and DSNWLI).

Related tasks
Link-editing an application with DSNULI
To create a single load module that can be used in more than one attachment environment, you can link-edit your program or stored procedure with the Universal Language Interface module (DSNUIL) instead of with one of the environment-specific language interface modules (DSNELI, DSNALI, DSNRLI, DSNCLI, or DFSLI000).

Link-editing an application with DSNULI
To create a single load module that can be used in more than one attachment environment, you can link-edit your program or stored procedure with the Universal Language Interface module (DSNUIL) instead of with one of the environment-specific language interface modules (DSNELI, DSNALI, DSNRLI, DSNCLI, or DFSLI000).

About this task
DSNUIL should be link-edited with TSO, CAF, RRSAF applications (including Stored Procedures), CICS applications and IMS applications. DSNUIL determines the run time environment, then dynamically loads and branches to the appropriate language interface module (DSNELI, DSNALI, DSNRLI, DSNCLI, or DFSLI000).

The following considerations apply:
• If maximum performance is the primary requirement, link-edit with DSNELI, DSNALI, DSNRLI, DSNCLI, or DFSLI000 rather than DSNULI. If maintaining a single copy of a load module is the primary requirement, link-edit with DSNULI.

• If CAF implicit connect functionality is required, link-edit your application with DSNALI instead of with DSNULI. DSNULI defaults to RRSAF implicit connections if an attachment environment has not been established upon entry to DSNHLI. Attachment environments are established by calling DSNRLI or DSNALI initially, or by running an SQL application under the TSO command processor or under CICS or IMS.

• DSNULI will not explicitly delete the loaded DSNELI, DSNALI, DSNRLI or DSNCLI. If an application cannot tolerate having these modules deleted only at task termination, use DSNELI, DSNALI, DSNRLI or DSNCLI instead of DSNULI.

• DSNULI is shipped with the linkage attributes AMODE(31) and RMODE(ANY) and must be entered in AMODE(31).

**Procedure**

You can include DSNULI when you link-edit your load module. For example, you can use a linkage editor control statement like this in your JCL:

```
INCLUDE SYSLIB(DSNULI)
```

**Results**

By coding this statement, you avoid linking to one of the environment-specific language interface modules.

### Controlling the CICS attachment facility from an application

Use the CICS attachment facility to access Db2 from CICS application programs.

**About this task**

You can start and stop the CICS attachment facility from within an application program.

**Procedure**

To control the CICS attachment facility:

1. To start the CICS attachment facility, perform one of the following actions:
   - Include the following statement in your application:
     ```
     EXEC CICS LINK PROGRAM('DSN2COM0')
     ```
   - Use the system programming interface SET DB2CONN for the CICS Transaction Server.

2. To stop the CICS attachment facility, perform one of the following actions:
   - Include the following statement in your application:
     ```
     EXEC CICS LINK PROGRAM('DSN2COM2')
     ```
   - Use the system programming interface SET DB2CONN for the CICS Transaction Server.

**Related information**

SET DB2CONN (CICS Transaction Server for z/OS)
Detecting whether the CICS attachment facility is operational

Before you execute SQL statements in a CICS program, you should determine if the CICS attachment facility is available. You do not need to do this test if the CICS attachment facility is started and you are using standby mode.

About this task

When an SQL statement is executed, and the CICS attachment facility is in standby mode, the attachment issues SQLCODE -923 with a reason code that indicates that Db2 is not available.

Procedure

Use the INQUIRE EXITPROGRAM command for the CICS Transaction Server in your application.

The following example shows how to use this command. In this example, the INQUIRE EXITPROGRAM command tests whether the resource manager for SQL, DSNCSQL, is up and running. CICS returns the results in the EIBRESP field of the EXEC interface block (EIB) and in the field whose name is the argument of the CONNECTST parameter (in this case, STST). If the EIBRESP value indicates that the command completed normally and the STST value indicates that the resource manager is available, you can then execute SQL statements.

```cics
STST     DS    F
ENTNAME  DS    CL8
EXITPROG DS    CL8

MVC   ENTNAME,=CL8'DSNCSQL'
MVC   EXITPROG,=CL8'DSN2EXT1'
EXEC CICS INQUIRE EXITPROGRAM(EXITPROG) ENTRYNAME(ENTNAME) CONNECTST(STST) NOHANDLE
CLC   EIBRESP,DFHRESP(NORMAL)
BNE   NOTREADY
CLC   STST,DFHVALUE(CONNECTED)
BNE   NOTREADY
UPNREADY DS    0H
attache is up
NOTREADY DS    0H
attache is not up yet
```

If you use the INQUIRE EXITPROGRAM command to avoid AEY9 abends and the CICS attachment facility is down, the storm drain effect can occur. The storm drain effect is a condition that occurs when a system continues to receive work, even though that system is down.

Related concepts

Storm-drain effect (Db2 Installation and Migration)

Related information

INQUIRE EXITPROGRAM (CICS Transaction Server for z/OS)
-923 (Db2 Codes)

Improving thread reuse in CICS applications

Having transactions reuse threads is generally recommended because each thread creation is associated with a high processor cost.

Procedure

Close all cursors that are declared with the WITH HOLD option before each sync point. Db2 does not automatically close them. A thread for an application that contains an open cursor cannot be reused. You should close all cursors immediately after you finish using them.

Related concepts

Held and non-held cursors
A held cursor does not close after a commit operation. A cursor that is not held closes after a commit operation. You specify whether you want a cursor to be held or not held by including or omitting the WITH HOLD clause when you declare the cursor.
Chapter 3. Db2 SQL programming

You can use the SQL language to write a statement that describes what you want to do with the data in a database and under what conditions you want to do it.

Structured Query Language (SQL) is a standardized language based on the relational model of data that is used for defining and manipulating data in a relational database. SQL statements can be contained in user-defined functions, user-defined procedures, or triggers, embedded in high-level language programs, dynamically prepared and run, or run interactively.

For information about embedded SQL, see Chapter 4, “Embedded SQL programming,” on page 461.

Creating and modifying Db2 objects from application programs

Your application program can create and manipulate Db2 objects, such as tables, views, triggers, distinct types, user-defined functions, and stored procedures. You must have the appropriate authorizations to create such objects.

Creating tables from application programs

Creating a table provides a logical place to store related data on a Db2 subsystem.

Procedure

Use a CREATE TABLE statement that includes the following elements:

- The name of the table. See Guidelines for table names (Db2 Administration Guide).
- A list of the columns that make up the table. Separate each column description from the next with a comma, and enclose the entire list of column descriptions in parentheses.

For each column, specify the following information:

- The name of the column (for example, SERIAL). See Column names (Db2 SQL).
- The data type and length attribute (for example, CHAR(8)). See Data types of columns (Introduction to Db2 for z/OS).
- Optionally, specify a default value, or a constraint on the value. You can use the following values:

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>NOT NULL</td>
<td>Specifies the column cannot contain null values</td>
</tr>
<tr>
<td>UNIQUE</td>
<td>The value for each row must be unique, and the column cannot contain null values.</td>
</tr>
<tr>
<td>DEFAULT</td>
<td>The column has one of the following Db2-assigned default values:</td>
</tr>
<tr>
<td></td>
<td>- For numeric columns, 0 (zero) is the default value.</td>
</tr>
<tr>
<td></td>
<td>- For character or graphic fixed-length strings, blank is the default value.</td>
</tr>
<tr>
<td></td>
<td>- For binary fixed-length strings, a set of hexadecimal zeros is the default value.</td>
</tr>
<tr>
<td></td>
<td>- For variable-length strings, including LOB strings, the empty string (a string of zero-length) is the default value.</td>
</tr>
</tbody>
</table>
Keyword | Result
--- | ---

- For datetime columns, the current value of the associated special register is the default value.

**DEFAULT value**
The default value is specified as one of the following values:
- A constant
- NULL
- SESSION_USER, which specifies the value of the SESSION_USER special register at the time when a default value is needed for the column
- CURRENT SQLID, which specifies the value of the CURRENT SQLID special register at the time when a default value is needed for the column
- The name of a cast function that casts a default value (of a built-in data type) to the distinct type of a column

- Optionally, specify the partitioning method for the data in the table. Db2 uses size-based partitions by default if you do not specify how to partition the data when you create the table. For more information, see Partitioning data in Db2 tables (Db2 Administration Guide).

- Optionally, a referential constraint or check constraint. For more information, see “Check constraints” on page 127 and “Referential constraints” on page 128.

**Example**
For example, the following SQL statement creates a table named PRODUCT:

```
CREATE TABLE PRODUCT
(SERIAL CHAR(8) NOT NULL,
DESCRIPTION VARCHAR(60) DEFAULT,
MFGCOST DECIMAL(8,2),
MFGDEPT CHAR(3),
MARKUP SMALLINT,
SALESDEPT CHAR(3),
CURDATE DATE DEFAULT);
```

**Related concepts**
Db2 tables (Introduction to Db2 for z/OS)

**Related tasks**
Creating base tables (Db2 Administration Guide)

**Related reference**
CREATE TABLE (Db2 SQL)

**Related information**
Lesson 1.2: Creating a table (Introduction to Db2 for z/OS)

**Data types of columns**
When you create a Db2 table, you define each column to have a specific data type. The data type of a column determines what you can and cannot do with the column.

When you perform operations on columns, the data must be compatible with the data type of the referenced column. For example, you cannot insert character data, such as a last name, into a column whose data type is numeric. Similarly, you cannot compare columns that contain incompatible data types.
The data type for a column can be a distinct type, which is a user-defined data type, or a Db2 built-in data type. As shown in the following figure, Db2 built-in data types have four general categories: datetime, string, numeric, and row identifier (ROWID).

**Figure 3. Db2 built-in data types**

- **Related concepts**
  - Assignment and comparison (Db2 SQL)
  - Casting between data types (Db2 SQL)
  - Rules for result data types (Db2 SQL)

- **Distinct types**
  - A distinct type is a user-defined data type that shares its internal representation with a built-in data type (its source type), but is considered to be a separate and incompatible data type for most operations.

- **Data types (Db2 SQL)**
Storing LOB data in a table

Db2 handles LOB data differently than it handles other kinds of data. As a result, in some cases, you need to take additional actions when you define LOB columns and insert the LOB data.

About this task

Large object and LOB refer to Db2 objects that you can use to store large amounts of data. A LOB is a varying-length character string that can contain up to 2 GB - 1 of data. Db2 supports the following LOB data types:

**Binary large object (BLOB)**
Use a BLOB to store binary data such as pictures, voice, and mixed media.

**Character large object (CLOB)**
Use a CLOB to store SBCS or mixed character data, such as documents.

**Double-byte character large object (DBCLOB)**
Use a DBCLOB to store data that consists of only DBCS data.

For more information about LOB data types, see [Large objects (LOBs) (Db2 SQL)](https://www.ibm.com/docs/en/db2/12.1?topic=large-objects-lobs).

You can use Db2 to store LOB data, but this data is stored differently than other kinds of data.

Although a table can have a LOB column, the actual LOB data is stored in another table, which called the auxiliary table. This auxiliary table exists in a separate table space called a LOB table space. One auxiliary table must exist for each LOB column. The table with the LOB column is called the base table. The base table has a ROWID column that Db2 uses to locate the data in the auxiliary table. The auxiliary table must have exactly one index.

Procedure

To store LOB data in Db2:

1. Define one or more columns of the appropriate LOB type and optionally a row identifier (ROWID) column by executing a CREATE TABLE statement or one or more ALTER TABLE statements.

   Define only one ROWID column, even if the table is to have multiple LOB columns. If you do not create a ROWID column before you define a LOB column, Db2 implicitly creates a ROWID column with the IMPLICITLY HIDDEN attribute and appends it as the last column of the table.

   If you add a ROWID column after you add a LOB column, the table has two ROWID columns: the implicitly-created column and the explicitly-created column. In this case, Db2 ensures that the values of the two ROWID columns are always identical.

   If Db2 implicitly creates the table space for this table or CURRENT RULES is set to STD, Db2 creates the necessary auxiliary objects for you and you can skip steps 2 and 3.

2. If you explicitly created the table space for this table and the CURRENT RULES special register is not set to STD, create a LOB table space and auxiliary table by using the CREATE LOB TABLESPACE and CREATE AUXILIARY TABLE statements.

   - If your base table is nonpartitioned, create one LOB table space and for each column create one auxiliary table.
   - If your base table is partitioned, create one LOB table space for each partition and one auxiliary table for each column. For example, if your base table has three partitions, you must create three LOB table spaces and three auxiliary tables for each LOB column.

3. If you explicitly created the table space for this table and the CURRENT RULES special register is not set to STD, create one index for each auxiliary table by using the CREATE INDEX statement.

4. Insert the LOB data into Db2 by using one of the following techniques:

   - If the total length of a LOB column and the base table row is less than 32 KB, use the LOAD utility and specify the base table.
• Otherwise, use INSERT, UPDATE, or MERGE statements and specify the base table. If you use the
INSERT statement, ensure that your application has enough storage available to hold the entire value
that is to be put into the LOB column.

Example

Suppose that you want to add a resume for each employee to the employee table. The employee resumes
are no more than 5 MB in size. Because the employee resumes contain single-byte characters, you can
define the resumes to Db2 as CLOBs. You therefore need to add a column of data type CLOB with a length
of 5 MB to the employee table. If you want to define a ROWID column explicitly, you must define it before
you define the CLOB column.

First, execute an ALTER TABLE statement to add the ROWID column, and then execute another ALTER
TABLE statement to add the CLOB column. The following statements create these columns:

```
ALTER TABLE EMP
  ADD ROW_ID ROWID NOT NULL GENERATED ALWAYS;
COMMIT;
ALTER TABLE EMP
  ADD EMP_RESUME CLOB(5M);
COMMIT;
```

If you explicitly created the table space for this table and the CURRENT RULES special register is not set
to STD, you then need to define a LOB table space and an auxiliary table to hold the employee resumes.
You also need to define an index on the auxiliary table. You must define the LOB table space in the same
database as the associated base table. The following statements create these objects:

```
CREATE LOB TABLESPACE RESUMETS
  IN DSN8D12A
  LOG NO
COMMIT;
CREATE AUXILIARY TABLE EMP_RESUME_TAB
  IN DSN8D12A.RESUMETS
  STORES DSN8C10.EMP
  COLUMN EMP_RESUME;
CREATE UNIQUE INDEX XEMP_RESUME
  ON EMP_RESUME_TAB;
COMMIT;
```

You can then load your employee resumes into Db2. In your application, you can define a host variable
to hold the resume, copy the resume data from a file into the host variable, and then execute an UPDATE
statement to copy the data into Db2. Although the LOB data is stored in the auxiliary table, your UPDATE
statement specifies the name of the base table. The following code declares a host variable to store the
resume in the C language:

```
SQL TYPE is CLOB (5M) resumedata;
```

The following UPDATE statement copies the data into Db2:

```
UPDATE EMP SET EMP_RESUME=:resumedata
  WHERE EMPNO=:employeenum;
```

In this statement, employeenum is a host variable that identifies the employee who is associated with a
resume.

Related concepts
Large objects (LOBs) (Db2 SQL)
Creation of large objects (Introduction to Db2 for z/OS)

Related reference
CREATE TABLE (Db2 SQL)
CREATE AUXILIARY TABLE (Db2 SQL)
CREATE LOB TABLESPACE (Db2 SQL)
CREATE INDEX (Db2 SQL)
Identity columns

An identity column contains a unique numeric value for each row in the table. Db2 can automatically generate sequential numeric values for this column as rows are inserted into the table. Thus, identity columns are ideal for primary key values, such as employee numbers or product numbers.

Using identity columns as keys

If you define a column with the AS IDENTITY attribute, and with the GENERATED ALWAYS and NO CYCLE attributes, Db2 automatically generates a monotonically increasing or decreasing sequential number for the value of that column when a new row is inserted into the table. However, for Db2 to guarantee that the values of the identity column are unique, you should define a unique index on that column.

You can use identity columns for primary keys that are typically unique sequential numbers, for example, order numbers or employee numbers. By doing so, you can avoid the concurrency problems that can result when an application generates its own unique counter outside the database.

Recommendation: Set the values of the foreign keys in the dependent tables after loading the parent table. If you use an identity column as a parent key in a referential integrity structure, loading data into that structure could be quite complicated. The values for the identity column are not known until the table is loaded because the column is defined as GENERATED ALWAYS.

You might have gaps in identity column values for the following reasons:

• If other applications are inserting values into the same identity column
• If Db2 terminates abnormally before it assigns all the cached values
• If your application rolls back a transaction that inserts identity values

Defining an identity column

You can define an identity column as either GENERATED BY DEFAULT or GENERATED ALWAYS:

• If you define the column as GENERATED BY DEFAULT, you can insert a value, and Db2 provides a default value if you do not supply one.
• If you define the column as GENERATED ALWAYS, Db2 always generates a value for the column, and you cannot insert data into that column. If you want the values to be unique, you must define the identity column with GENERATED ALWAYS and NO CYCLE and define a unique index on that column.

The values that Db2 generates for an identity column depend on how the column is defined. The START WITH option determines the first value that Db2 generates. The values advance by the INCREMENT BY value in ascending or descending order.

The MINVALUE and MAXVALUE options determine the minimum and maximum values that Db2 generates. However, the The CYCLE or NO CYCLE option determines whether Db2 wraps values when it has generated all values between the START WITH value and MAXVALUE if the values are ascending, or between the START WITH value and MINVALUE if the values are descending. MINVALUE and MAXVALUE do not constrain a START WITH or RESTART WITH value.

Example: Using GENERATED ALWAYS and CYCLE

Suppose that table T1 is defined with GENERATED ALWAYS and CYCLE:

```
CREATE TABLE T1
  (CHARCOL1 CHAR(1),
   IDENTCOL1 SMALLINT GENERATED ALWAYS AS IDENTITY
    (START WITH -1,
     INCREMENT BY 1,
     CYCLE,
     MINVALUE -3,
     MAXVALUE 3));
```
Now suppose that you execute the following INSERT statement eight times:

```
INSERT INTO T1 (CHARCOL1) VALUES ('A');
```

When Db2 generates values for IDENTCOL1, it starts with -1 and increments by 1 until it reaches the MAXVALUE of 3 on the fifth INSERT. To generate the value for the sixth INSERT, Db2 cycles back to MINVALUE, which is -3. T1 looks like this after the eight INSERT statements are executed:

<table>
<thead>
<tr>
<th>CHARCOL1</th>
<th>IDENTCOL1</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>-1</td>
</tr>
<tr>
<td>A</td>
<td>0</td>
</tr>
<tr>
<td>A</td>
<td>1</td>
</tr>
<tr>
<td>A</td>
<td>2</td>
</tr>
<tr>
<td>A</td>
<td>3</td>
</tr>
<tr>
<td>A</td>
<td>-3</td>
</tr>
<tr>
<td>A</td>
<td>-2</td>
</tr>
<tr>
<td>A</td>
<td>-1</td>
</tr>
</tbody>
</table>

The value of IDENTCOL1 for the eighth INSERT repeats the value of IDENTCOL1 for the first INSERT.

**Example: START WITH or RESTART WITH values outside the range for cycling**

The MINVALUE and MAXVALUE options do not constrain the START WITH value. That is, the START WITH clause can be used to start the generation of values outside the range that is used for cycles. However, the next generated value after the specified START WITH value is MINVALUE for an ascending identity column or MAXVALUE for a descending identity column. The same is true if you alter the identity column and specify a RESTART WITH value.

Consider T1 from the previous example, and suppose that you alter the table with a statement that specifies the following keywords.

```
ALTER TABLE T1
ALTER COLUMN IDENTCOL1 SET GENERATED ALWAYS RESTART WITH 99;
```

Now suppose that you execute the following INSERT statement three times:

```
INSERT INTO T1 (CHARCOL1) VALUES ('B');
```

When Db2 generates the IDENTCOL1 value, it starts with 99. However, for the next generated value, Db2 again cycles back to MINVALUE, which is -3. T1 looks like this after the three INSERT statements are executed:

<table>
<thead>
<tr>
<th>CHARCOL1</th>
<th>IDENTCOL1</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>-1</td>
</tr>
<tr>
<td>A</td>
<td>0</td>
</tr>
<tr>
<td>A</td>
<td>1</td>
</tr>
<tr>
<td>A</td>
<td>2</td>
</tr>
<tr>
<td>A</td>
<td>3</td>
</tr>
<tr>
<td>A</td>
<td>-3</td>
</tr>
<tr>
<td>A</td>
<td>-2</td>
</tr>
<tr>
<td>A</td>
<td>-1</td>
</tr>
<tr>
<td>B</td>
<td>99</td>
</tr>
<tr>
<td>B</td>
<td>-3</td>
</tr>
<tr>
<td>B</td>
<td>-2</td>
</tr>
</tbody>
</table>

**Identity columns as primary keys**

The SELECT from INSERT statement enables you to insert a row into a parent table with its primary key defined as a Db2-generated identity column, and retrieve the value of the primary or parent key. You can then use this generated value as a foreign key in a dependent table.

In addition, you can use the IDENTITY_VAL_LOCAL function to return the most recently assigned value for an identity column.
Example: Using SELECT from INSERT

Suppose that an EMPLOYEE table and a DEPARTMENT table are defined in the following way:

```
CREATE TABLE EMPLOYEE
(EMPNO      INTEGER GENERATED ALWAYS AS IDENTITY
  PRIMARY KEY NOT NULL,
NAME       CHAR(30) NOT NULL,
SALARY     DECIMAL(7,2) NOT NULL,
WORKDEPT   SMALLINT);

CREATE TABLE DEPARTMENT
(DEPTNO     SMALLINT NOT NULL PRIMARY KEY,
DEPTNAME   VARCHAR(30),
MGRNO      INTEGER NOT NULL,
CONSTRAINT REF_EMPNO FOREIGN KEY (MGRNO)
  REFERENCES EMPLOYEE (EMPNO) ON DELETE RESTRICT);

ALTER TABLE EMPLOYEE ADD
CONSTRAINT REF_DEPTNO FOREIGN KEY (WORKDEPT)
  REFERENCES DEPARTMENT (DEPTNO) ON DELETE SET NULL;
```

When you insert a new employee into the EMPLOYEE table, to retrieve the value for the EMPNO column, you can use the following SELECT from INSERT statement:

```
EXEC SQL
SELECT EMPNO INTO :hv_empno
FROM FINAL TABLE (INSERT INTO EMPLOYEE (NAME, SALARY, WORKDEPT)
VALUES ('New Employee', 75000.00, 11));
```

The SELECT statement returns the Db2-generated identity value for the EMPNO column in the host variable :hv_empno.

You can then use the value in :hv_empno to update the MGRNO column in the DEPARTMENT table with the new employee as the department manager:

```
EXEC SQL
UPDATE DEPARTMENT
SET MGRNO = :hv_empno
WHERE DEPTNO = 11;
```

Related concepts

Rules for inserting data into an identity column
An identity column contains a unique numeric value for each row in the table. Whether you can insert data into an identity column and how that data gets inserted depends on how the column is defined.

Related tasks

Selecting values while inserting data
When you insert rows into a table, you can also select values from the inserted rows at the same time.

Related reference

IDENTITY_VAL_LOCAL (Db2 SQL)

Creating tables for data integrity

To ensure that only valid data is added to your tables, you can use constraints, triggers, and unique indexes. For example, you might need to ensure that all items in your inventory table have valid item numbers and to prevent items without valid item numbers from being added.

About this task

Introductory concepts

Creation of relationships with referential constraints (Introduction to Db2 for z/OS)

Related concepts

Creation of relationships with referential constraints (Introduction to Db2 for z/OS)
Related tasks
Altering a table for referential integrity (Db2 Administration Guide)
Creating indexes to improve referential integrity performance for foreign keys (Db2 Performance)
Creating tables for data integrity
To ensure that only valid data is added to your tables, you can use constraints, triggers, and unique indexes. For example, you might need to ensure that all items in your inventory table have valid item numbers and to prevent items without valid item numbers from being added.
Using referential integrity for data consistency (Managing Security)

Ways to maintain data integrity
When you add or modify data in a Db2 table, you need to ensure that the data is valid. Two techniques that you can use to ensure valid data are constraints and triggers.

**Constraints** are rules that limit the values that you can insert, delete, or update in a table. There are two types of constraints:

- Check constraints determine the values that a column can contain. Check constraints are discussed in "Check constraints" on page 127.
- Referential constraints preserve relationships between tables. Referential constraints are discussed in "Referential constraints" on page 128. A specific type of referential constraints, the informational referential constraint, is discussed in "Informational referential constraints" on page 130.

To maintain data integrity Db2 enforces check constraints and referential constraints on data in a table. When these types of constraints are violated or might be violated, Db2 places the table space or partition that contains the table in CHECK-pending status.

**Triggers** are a series of actions that are invoked when a table is updated. Triggers are discussed in “Creating a trigger” on page 149.

Related reference
CHECK-pending status (Db2 Utilities)

**Check constraints**

A check constraint is a rule that specifies the values that are allowed in one or more columns of every row of a base table. For example, you can define a check constraint to ensure that all values in a column that contains ages are positive numbers.

Check constraints designate the values that specific columns of a base table can contain, providing you a method of controlling the integrity of data entered into tables. You can create tables with check constraints using the CREATE TABLE statement, or you can add the constraints with the ALTER TABLE statement. However, if the check integrity is compromised or cannot be guaranteed for a table, the table space or partition that contains the table is placed in a check pending state. Check integrity is the condition that exists when each row of a table conforms to the check constraints defined on that table.

For example, you might want to make sure that no salary can be below 15000 dollars. To do this, you can create the following check constraint:

```sql
CREATE TABLE EMPSAL
(ID          INTEGER      NOT NULL,
 SALARY      INTEGER      CHECK (SALARY >= 15000));
```

Using check constraints makes your programming task easier, because you do not need to enforce those constraints within application programs or with a validation routine. Define check constraints on one or more columns in a table when that table is created or altered.

**Check constraint considerations**

The syntax of a check constraint is checked when the constraint is defined, but the meaning of the constraint is not checked. The following examples show mistakes that are not caught. Column C1 is defined as INTEGER NOT NULL.
Allowable but mistaken check constraints:

- A self-contradictory check constraint:
  
  ```sql
  CHECK (C1 > 5 AND C1 < 2)
  ```

- Two check constraints that contradict each other:
  
  ```sql
  CHECK (C1 > 5)
  CHECK (C1 < 2)
  ```

- Two check constraints, one of which is redundant:
  
  ```sql
  CHECK (C1 > 0)
  CHECK (C1 >= 1)
  ```

- A check constraint that contradicts the column definition:
  
  ```sql
  CHECK (C1 IS NULL)
  ```

- A check constraint that repeats the column definition:
  
  ```sql
  CHECK (C1 IS NOT NULL)
  ```

A check constraint is not checked for consistency with other types of constraints. For example, a column in a dependent table can have a referential constraint with a delete rule of SET NULL. You can also define a check constraint that prohibits nulls in the column. As a result, an attempt to delete a parent row fails, because setting the dependent row to null violates the check constraint.

Similarly, a check constraint is not checked for consistency with a validation routine, which is applied to a table before a check constraint. If the routine requires a column to be greater than or equal to 10 and a check constraint requires the same column to be less than 10, table inserts are not possible. Plans and packages do not need to be rebound after check constraints are defined on or removed from a table.

When check constraints are enforced

After check constraints are defined on a table, any change must satisfy those constraints if it is made by:

- The LOAD utility with the option ENFORCE CONSTRAINT
- An SQL insert operation
- An SQL update operation

A row satisfies a check constraint if its condition evaluates either to true or to unknown. A condition can evaluate to unknown for a row if one of the named columns contains the null value for that row.

Any constraint defined on columns of a base table applies to the views defined on that base table.

When you use ALTER TABLE to add a check constraint to already populated tables, the enforcement of the check constraint is determined by the value of the CURRENT RULES special register as follows:

- If the value is STD, the check constraint is enforced immediately when it is defined. If a row does not conform, the check constraint is not added to the table and an error occurs.
- If the value is Db2, the check constraint is added to the table description but its enforcement is deferred. Because there might be rows in the table that violate the check constraint, the table is placed in CHECK-pending status.

Referential constraints

A referential constraint is a rule that specifies that the only valid values for a particular column are those values that exist in another specified table column. For example, a referential constraint can ensure that all customer IDs in a transaction table exist in the ID column of a customer table.

A table can serve as the "master list" of all occurrences of an entity. In the sample application, the employee table serves that purpose for employees; the numbers that appear in that table are the only
valid employee numbers. Likewise, the department table provides a master list of all valid department numbers; the project activity table provides a master list of activities performed for projects; and so on.

The following figure shows the relationships that exist among the tables in the sample application. Arrows point from parent tables to dependent tables.

![Diagram showing relationships among tables in the sample application](image)

**Figure 4. Relationships among tables in the sample application**

When a table refers to an entity for which there is a master list, it should identify an occurrence of the entity that actually appears in the master list; otherwise, either the reference is invalid or the master list is incomplete. Referential constraints enforce the relationship between a table and a master list.

**Restrictions on cycles of dependent tables**

A cycle is a set of two or more tables. The tables are ordered so that each is a dependent of the one before it, and the first is a dependent of the last. Every table in the cycle is a descendent of itself. Db2 restricts certain operations on cycles.

In the sample application, the employee and department tables are a cycle; each is a dependent of the other.

Db2 does not allow you to create a cycle in which a delete operation on a table involves that same table. Enforcing that principle creates rules about adding a foreign key to a table:

- In a cycle of two tables, neither delete rule can be CASCADE.
- In a cycle of more than two tables, two or more delete rules must not be CASCADE. For example, in a cycle with three tables, two of the delete rules must be other than CASCADE. This concept is illustrated in the following figure. The cycle on the left is valid because two or more of the delete rules are not CASCADE. The cycle on the right is invalid because it contains two cascading deletes.
Alternatively, a delete operation on a self-referencing table must involve the same table, and the delete rule there must be CASCADE or NO ACTION.

**Recommendation:** Avoid creating a cycle in which all the delete rules are RESTRICT and none of the foreign keys allows nulls. If you do this, no row of any of the tables can ever be deleted.

**Referential constraints on tables with multilevel security with row-level granularity**
You cannot use referential constraints on a security label column, which is used for multilevel security with row-level granularity. However, you can use referential constraints on other columns in the row.

Db2 does not enforce multilevel security with row-level granularity when it is already enforcing referential constraints. Referential constraints are enforced when the following situations occur:

- An insert operation is applied to a dependent table.
- An update operation is applied to a foreign key of a dependent table, or to the parent key of a parent table.
- A delete operation is applied to a parent table. In addition to all referential constraints being enforced, the Db2 system enforces all delete rules for all dependent rows that are affected by the delete operation. If all referential constraints and delete rules are not satisfied, the delete operation will not succeed.
- The LOAD utility with the ENFORCE CONSTRAINTS option is run on a dependent table.
- The CHECK DATA utility is run.

**Related concepts**

Multilevel security (Managing Security)

**Informational referential constraints**
An informational referential constraint is a referential constraint that Db2 does not enforce during normal operations. Use these constraints only when referential integrity can be enforced by another means, such as when retrieving data from other sources. These constraints might improve performance by enabling the query to qualify for automatic query rewrite.

Db2 ignores informational referential constraints during insert, update, and delete operations. Some utilities ignore these constraints; other utilities recognize them. For example, CHECK DATA and LOAD ignore these constraints. QUIESCE TABLESPACES recognizes these constraints by quiescing all table spaces related to the specified table space.

You should use this type of referential constraint only when an application process verifies the data in a referential integrity relationship. For example, when inserting a row in a dependent table, the application should verify that a foreign key exists as a primary or unique key in the parent table. To define an informational referential constraint, use the NOT ENFORCED option of the referential constraint definition in a CREATE TABLE or ALTER TABLE statement.

Informational referential constraints are often useful, especially in a data warehouse environment, for several reasons:

- To avoid the overhead of enforcement by Db2.
Typically, data in a data warehouse has been extracted and cleansed from other sources. Referential integrity might already be guaranteed. In this situation, enforcement by Db2 is unnecessary.

- To allow more queries to qualify for automatic query rewrite.

Automatic query rewrite is a process that examines a submitted query that references source tables and, if appropriate, rewrites the query so that it executes against a materialized query table that has been derived from those source tables. This process uses informational referential constraints to determine whether the query can use a materialized query table. Automatic query rewrite results in a significant reduction in query run time, especially for decision-support queries that operate over huge amounts of data.

**Related tasks**
Using materialized query tables to improve SQL performance (Db2 Performance)

**Related reference**
CREATE TABLE (Db2 SQL)

**Defining a parent key and unique index**
A *parent key* is either a primary key or a unique key in the parent table of a referential constraint. The values of a parent key determine the valid values of the foreign key in the constraint. You must create a unique index on a parent key.

**About this task**
The primary key of a table, if one exists, uniquely identifies each occurrence of an entity in the table. The PRIMARY KEY clause of the CREATE TABLE or ALTER TABLE statements identifies the column or columns of the primary key. Each identified column must be defined as NOT NULL.

Another way to allow only unique values in a column is to specify the UNIQUE clause when you create or alter a table.

A table that is to be a parent of dependent tables must have a primary or a unique key; the foreign keys of the dependent tables refer to the primary or unique key. Otherwise, a primary key is optional. Consider defining a primary key if each row of your table does pertain to a unique occurrence of some entity. If you define a primary key, an index must be created (the *primary index*) on the same set of columns, in the same order as those columns. If you are defining referential constraints for Db2 to enforce, takes steps to maintain data integrity read before creating or altering any of the tables involved.

A table can have no more than one primary key. A primary key has the same restrictions as index keys:

- The key can include no more than 64 columns.
- You cannot specify a column name twice.
- The sum of the column length attributes cannot be greater than 2000.

You define a list of columns as the primary key of a table with the PRIMARY KEY clause in the CREATE TABLE statement.

**Procedure**
Use the PRIMARY KEY clause in an ALTER TABLE statement. In this case, a unique index must already exist.

Consider the following items when you plan for primary keys:

- The theoretical model of a relational database suggests that every table should have a primary key to uniquely identify the entities it describes. However, you must weigh that model against the potential cost of index maintenance overhead. Db2 does not require you to define a primary key for tables with no dependents.
- Choose a primary key whose values will not change over time. Choosing a primary key with persistent values enforces the good practice of having unique identifiers that remain the same for the lifetime of the entity occurrence.
• A primary key column should not have default values unless the primary key is a single TIMESTAMP column.
• Choose the minimum number of columns to ensure uniqueness of the primary key.
• A view that can be updated that is defined on a table with a primary key should include all columns of the key. Although this is necessary only if the view is used for inserts, the unique identification of rows can be useful if the view is used for updates, deletes, or selects.
• Drop a primary key later if you change your database or application using SQL.

Related concepts
Ways to maintain data integrity
When you add or modify data in a Db2 table, you need to ensure that the data is valid. Two techniques that you can use to ensure valid data are constraints and triggers.

Related reference
ALTER TABLE (Db2 SQL)
CREATE TABLE (Db2 SQL)

Parent key columns
A parent key is either a primary key or a unique key in the parent table of a referential constraint. This key consists of a column or set of columns. The values of a parent key determine the valid values of the foreign key in the constraint.

If every row in a table represents relationships for a unique entity, the table should have one column or a set of columns that provides a unique identifier for the rows of the table. This column (or set of columns) is called the parent key of the table. To ensure that the parent key does not contain duplicate values, you must create a unique index on the column or columns that constitute the parent key. Defining the parent key is called entity integrity, because it requires each entity to have a unique key.

In some cases, using a timestamp as part of the key can be helpful, for example when a table does not have a "natural" unique key or if arrival sequence is the key.

Primary keys for some of the sample tables are:

<table>
<thead>
<tr>
<th>Table</th>
<th>Key Column</th>
</tr>
</thead>
<tbody>
<tr>
<td>Employee table</td>
<td>EMPNO</td>
</tr>
<tr>
<td>Department table</td>
<td>DEPTNO</td>
</tr>
<tr>
<td>Project table</td>
<td>PROJNO</td>
</tr>
</tbody>
</table>

Table 35 on page 132 shows part of the project table which has the primary key column, PROJNO.

<table>
<thead>
<tr>
<th>PROJNO</th>
<th>PROJNAME</th>
<th>DEPTNO</th>
</tr>
</thead>
<tbody>
<tr>
<td>MA2100</td>
<td>WELD LINE AUTOMATION</td>
<td>D01</td>
</tr>
<tr>
<td>MA2110</td>
<td>W L PROGRAMMING</td>
<td>D11</td>
</tr>
</tbody>
</table>

Table 36 on page 132 shows part of the project activity table, which has a primary key that contains more than one column. The primary key is a composite key, which consists of the PRONNO, ACTNO, and ACSTDATE columns.

<table>
<thead>
<tr>
<th>PROJNO</th>
<th>ACTNO</th>
<th>ACSTAFF</th>
<th>ACSTDATE</th>
<th>ACENDATE</th>
</tr>
</thead>
<tbody>
<tr>
<td>AD3100</td>
<td>10</td>
<td>.50</td>
<td>1982-01-01</td>
<td>1982-07-01</td>
</tr>
</tbody>
</table>
### Table 36. Part of the Project activities table with a composite primary key (continued)

<table>
<thead>
<tr>
<th>PROJNO</th>
<th>ACTNO</th>
<th>ACSTAFF</th>
<th>ACSTDATE</th>
<th>ACENDATE</th>
</tr>
</thead>
<tbody>
<tr>
<td>AD3110</td>
<td>10</td>
<td>1.00</td>
<td>1982-01-01</td>
<td>1983-01-01</td>
</tr>
<tr>
<td>AD3111</td>
<td>60</td>
<td>.50</td>
<td>1982-03-15</td>
<td>1982-04-15</td>
</tr>
</tbody>
</table>

### Defining a foreign key

Use foreign keys to enforce referential relationships between tables. A foreign key is a column or set of columns that references the parent key in the parent table.

### Before you begin

The following prerequisites are met:

- The privilege set must include the ALTER or the REFERENCES privilege on the columns of the parent key.
- A unique index exists on the parent key columns of the parent table.

### Procedure

To define a foreign key, use one of the following approaches:

- Issue a CREATE TABLE statement and specify a FOREIGN KEY clause.
  
  a) Choose a constraint name for the relationship that is defined by a foreign key.

  If you do not choose a name, Db2 generates one from the name of the first column of the foreign key, in the same way that it generates the name of an implicitly created table space.

  For example, the names of the relationships in which the employee-to-project activity table is a dependent would, by default, be recorded (in column RELNAME of SYSIBM.SYSFOREIGNKEYS) as EMPNO and PROJNO.

  The name is used in error messages, queries to the catalog, and DROP FOREIGN KEY statements. Hence, you might want to choose one if you are experimenting with your database design and have more than one foreign key that begins with the same column (otherwise Db2 generates the name).

  b) Specify column names that identify the columns of the parent key.

  A foreign key can refer to either a unique or a primary key of the parent table. If the foreign key refers to a non-primary unique key, you must specify the column names of the key explicitly. If the column names of the key are not specified explicitly, the default is to refer to the column names of the primary key of the parent table.

- Issue an ALTER TABLE statement and specify the FOREIGN KEY clause.

  You can add a foreign key to an existing table; in fact, that is sometimes the only way to proceed. To make a table self-referencing, you must add a foreign key after creating it. When a foreign key is added to a populated table, the table space is put into CHECK-pending status.

### Example

The following example shows a CREATE TABLE statement that specifies constraint names REPAPA and REPABE for the foreign keys in the employee-to-project activity table.

```sql
CREATE TABLE DSN8C10.EMPPROJACT
(EMPNO     CHAR(6)          NOT NULL,
 PROJNO    CHAR(6)          NOT NULL,
 ACTNO     SMALLINT         NOT NULL,
 CONSTRAINT REPAPA FOREIGN KEY (PROJNO, ACTNO)
 REFERENCES DSN8C10.PROJACT ON DELETE RESTRICT,
 CONSTRAINT REPABE FOREIGN KEY (EMPNO)
 REFERENCES DSN8C10.EMP ON DELETE RESTRICT)
 IN DATABASE DSN8D12A;
```
What to do next

If rows of the parent table are often deleted, it is best to create an index on the foreign key.

Related tasks
Adding parent keys and foreign keys (Db2 Administration Guide)

Related reference
CREATE TABLE (Db2 SQL)
ALTER TABLE (Db2 SQL)
SYSFOREIGNKEYS catalog table (Db2 SQL)

Maintaining referential integrity when using data encryption

If you use encrypted data in a referential constraint, the primary key of the parent table and the foreign key of the dependent table must have the same encrypted value.

About this task

The encrypted value should be extracted from the parent table (the primary key) and used for the dependent table (the foreign key). You can do this in one of the following two ways:

- Use the FINAL TABLE clause on a SELECT from UPDATE, SELECT from INSERT, or SELECT from MERGE statement.
- Use the ENCRYPT_TDES function to encrypt the foreign key using the same password as the primary key. The encrypted value of the foreign key will be the same as the encrypted value of the primary key.

The SET ENCRYPTION PASSWORD statement sets the password that will be used for the ENCRYPT_TDES function.

Related reference
ENCRIPT_TDES or ENCRYPT (Db2 SQL)
ENCRYPTION PASSWORD (Db2 SQL)

Creating work tables for the EMP and DEPT sample tables

Before testing SQL statements that insert, update, and delete rows in the DSN8C10.EMP and DSN8C10.DEPT sample tables, you should create duplicates of these tables. Create duplicates so that the original sample tables remain intact. These duplicate tables are called work tables.

About this task

This topic shows how to create the department and employee work tables and how to fill a work table with the contents of another table:

Each of these topics assumes that you logged on by using your own authorization ID. The authorization ID qualifies the name of each object that you create. For example, if your authorization ID is SMITH, and you create table YDEPT, the name of the table is SMITH.YDEPT. If you want to access table DSN8C10.DEPT, you must refer to it by its complete name. If you want to access your own table YDEPT, you need only to refer to it as YDEPT.

Use the following statements to create a new department table called YDEPT, modeled after the existing table, DSN8C10.DEPT, and an index for YDEPT:

```
CREATE TABLE YDEPT
LIKE DSN8C10.DEPT;

CREATE UNIQUE INDEX YDEPTX
ON YDEPT (DEPTNO);
```
If you want DEPTNO to be a primary key, as in the sample table, explicitly define the key. Use an ALTER TABLE statement, as in the following example:

```
ALTER TABLE YDEPT
  PRIMARY KEY (DEPTNO);
```

You can use an INSERT statement to copy the rows of the result table of a fullselect from one table to another. The following statement copies all of the rows from DSN8C10.DEPT to your own YDEPT work table:

```
INSERT INTO YDEPT
  SELECT *
  FROM DSN8C10.DEPT;
```

For information about using the INSERT statement, see “Inserting rows by using the INSERT statement” on page 335.

You can use the following statements to create a new employee table called YEMP:

```
CREATE TABLE YEMP
  (EMPNO     CHAR(6)        PRIMARY KEY NOT NULL,
   FIRSTNME  VARCHAR(12)    NOT NULL,
   MIDINIT   CHAR(1)        NOT NULL,
   LASTNAME  VARCHAR(15)    NOT NULL,
   WORKDEPT  CHAR(3)        REFERENCES YDEPT
     ON DELETE SET NULL,
   PHONENO   CHAR(4)        UNIQUE NOT NULL,
   HIREDATE  DATE                   ,
   JOB       CHAR(8)                ,
   EDLEVEL   SMALLINT               ,
   SEX       CHAR(1)                ,
   BIRTHDATE DATE                   ,
   SALARY    DECIMAL(9, 2)          ,
   BONUS     DECIMAL(9, 2)          ,
   COMM      DECIMAL(9, 2)          );
```

This statement also creates a referential constraint between the foreign key in YEMP (WORKDEPT) and the primary key in YDEPT (DEPTNO). It also restricts all phone numbers to unique numbers.

If you want to change a table definition after you create it, use the ALTER TABLE statement with a RENAME clause. If you want to change a table name after you create it, use the RENAME statement.

You can change a table definition by using the ALTER TABLE statement only in certain ways. For example, you can add and drop constraints on columns in a table. You can also change the data type of a column within character data types, within numeric data types, and within graphic data types. You can add a column to a table. However, you cannot use the ALTER TABLE statement to drop a column from a table.

Related tasks
Altering Db2 tables (Db2 Administration Guide)
Related reference
ALTER TABLE (Db2 SQL)
RENAME (Db2 SQL)

Creating created temporary tables

Use created temporary tables when you need to store data for only the life of an application process, but you want to share the table definition.

About this task

Db2 does not perform logging and locking operations for created temporary tables. Therefore, SQL statements that use these tables can execute queries efficiently.

Each application process has its own instance of the created temporary table.

An instance of a created temporary table exists at the current server until one of the following actions occurs:
• The application process ends.
• The remote server connection through which the instance was created terminates.
• The unit of work in which the instance was created completes.

When you run a ROLLBACK statement, Db2 deletes the instance of the created temporary table. When you run a COMMIT statement, Db2 deletes the instance of the created temporary table unless a cursor for accessing the created temporary table is defined with the WITH HOLD clause and is open.

You create the definition of a created temporary table using the SQL CREATE GLOBAL TEMPORARY TABLE statement.

Procedure

To create a created temporary table:

1. Define the table by issuing CREATE GLOBAL TEMPORARY TABLE statement.

   For example, the following statement creates the definition of a table called TEMPPROD:

   ```sql
   CREATE GLOBAL TEMPORARY TABLE TEMPPROD
   (SERIAL     CHAR(8)     NOT NULL,  
    DESCRIPTION VARCHAR(60) NOT NULL,  
    MFGCOST     DECIMAL(8,2),  
    MFGDEPT     CHAR(3),  
    MARKUP      SMALLINT,  
    SALESDEPT   CHAR(3),  
    CURDATE     DATE        NOT NULL);
   ```

   You can also create this same definition by copying the definition of a base table (named PROD) by using the LIKE clause:

   ```sql
   CREATE GLOBAL TEMPORARY TABLE TEMPPROD LIKE PROD;
   ```

   **Restriction:** You cannot use the MERGE statement with created temporary tables.

   The SQL statements in the examples create identical definitions for the TEMPPROD table, but these tables differ slightly from the PROD sample table PROD. The PROD sample table contains two columns, DESCRIPTION and CURDATE, that are defined as NOT NULL WITH DEFAULT. Because created temporary tables do not support non-null default values, the DESCRIPTION and CURDATE columns in the TEMPPROD table are defined as NOT NULL and do not have defaults.

   After you run one of the two CREATE statements, the definition of TEMPPROD exists, but no instances of the table exist.

2. Create an instance of the created temporary table by using it in an application.

   Db2 creates an instance of the table when it is specified in one of the following SQL statements:

   • OPEN
   • SELECT
   • INSERT
   • DELETE

   For example, suppose that you defined TEMPPROD as described the previous step and then run an application that contains the following statements:

   ```sql
   EXEC SQL DECLARE C1 CURSOR FOR SELECT * FROM TEMPPROD;
   EXEC SQL INSERT INTO TEMPPROD SELECT * FROM PROD;
   EXEC SQL OPEN C1;
   ...
   EXEC SQL COMMIT;
   ...
   EXEC SQL CLOSE C1;
   ```

   When you run the INSERT statement, Db2 creates an instance of TEMPPROD and populates that instance with rows from table PROD. When the COMMIT statement runs, Db2 deletes all rows
from TEMPPROD. However, assume that you change the declaration of cursor C1 to the following declaration:

```sql
EXEC SQL DECLARE C1 CURSOR WITH HOLD
   FOR SELECT * FROM TEMPPROD;
```

In this case, Db2 does not delete the contents of TEMPPROD until the application ends because C1, a cursor that is defined with the WITH HOLD clause, is open when the COMMIT statement runs. In either case, Db2 drops the instance of TEMPPROD when the application ends.

3. When the table is no longer needed, issue a DROP statement.
   For example, to drop the definition of TEMPPROD, you must run the following statement:

```sql
DROP TABLE TEMPPROD;
```

Related reference

- CREATE GLOBAL TEMPORARY TABLE (Db2 SQL)
- DROP (Db2 SQL)

**Temporary tables**

Use temporary tables when you need to store data for only the duration of an application process. Depending on whether you want to share the table definition, you can create a created temporary table or a declared temporary table.

The two kinds of temporary tables are:

- Created temporary tables, which you define using a CREATE GLOBAL TEMPORARY TABLE statement
- Declared temporary tables, which you define using a DECLARE GLOBAL TEMPORARY TABLE statement

SQL statements that use temporary tables can run faster because of the following reasons:

- For created temporary tables, Db2 provides no logging. For declared temporary tables, Db2 provides limited logging that can be further limited by the NOT LOGGED option of the DECLARE GLOBAL TEMPORARY TABLE statement.
- For created temporary tables, Db2 provides no locking. For declared temporary tables, Db2 provides limited locking.

Temporary tables are especially useful when you need to sort or query intermediate result tables that contain a large number of rows, but you want to store only a small subset of those rows permanently.

Temporary tables can also return result sets from stored procedures. The following topics provide more details about created temporary tables and declared temporary tables:

- “Creating created temporary tables” on page 135
- “Creating declared temporary tables” on page 137

For more information, see “Writing an external procedure to return result sets to a distributed client” on page 279.

**Creating declared temporary tables**

Use declared temporary tables when you need to store data for only the life of an application process and do not need to share the table definition. The definition of this table exists only while the application process runs. Db2 performs limited logging and locking operations for declared temporary tables.

**Before you begin**

Before you can define declared temporary tables, you must have a WORKFILE database that has at least one table space with a 32-KB page size.
About this task

You create an instance of a declared temporary table by using the SQL DECLARE GLOBAL TEMPORARY TABLE statement. That instance is known only to the application process in which the table is declared, so you can declare temporary tables with the same name in different applications. The qualifier for a declared temporary table is SESSION.

To create a declared temporary table, specify the DECLARE GLOBAL TEMPORARY TABLE statement. In that statement, specify the columns that the table is to contain by performing one of the following actions:

Procedure

To create a declared temporary table:

1. Issue a DECLARE GLOBAL TEMPORARY TABLE statement.

In that statement, you an specify the columns that the table is to contain by performing one of the following actions:

- Specify all the columns in the table. For example, the following statement defines a declared temporary table called TEMPPROD by explicitly specifying the columns.

  ```sql
  DECLARE GLOBAL TEMPORARY TABLE TEMPPROD
  (SERIAL       CHAR(8)     NOT NULL WITH DEFAULT '99999999',
   DESCRIPTION  VARCHAR(60) NOT NULL,
   PRODCOUNT    INTEGER GENERATED ALWAYS AS IDENTITY,
   MFGCOST      DECIMAL(8,2),
   MFGDEPT      CHAR(3),
   MARKUP       SMALLINT,
   SALESDEPT    CHAR(3),
   CURDATE      DATE        NOT NULL);
  ```

  Unlike columns of created temporary tables, columns of declared temporary tables can include the WITH DEFAULT clause.

- Use a LIKE clause to copy the definition of a base table, created temporary table, or view. For example, the following statement defines a declared temporary table called TEMPPROD by copying the definition of a base table. The base table has an identity column that the declared temporary table also uses as an identity column.

  ```sql
  DECLARE GLOBAL TEMPORARY TABLE TEMPPROD LIKE BASEPROD INCLUDING IDENTITY COLUMN ATTRIBUTES;
  ```

- If the base table, created temporary table, or view from which you select columns has identity columns, you can specify that the corresponding columns in the declared temporary table are also identity columns. To include these identity columns, specify the INCLUDING IDENTITY COLUMN ATTRIBUTES clause when you define the declared temporary table.

  If the source table has a row change timestamp column, you can specify that those column attributes are inherited in the declared temporary table by specifying INCLUDING ROW CHANGE TIMESTAMP COLUMN ATTRIBUTES.

- Use a fullselect to choose specific columns from a base table, created temporary table, or view. For example, the following statement defines a declared temporary table called TEMPPROD by selecting columns from a view. The view has an identity column that the declared temporary table also uses as an identity column. The declared temporary table inherits its default column values from the default column values of a base table on which the view is based.

  ```sql
  DECLARE GLOBAL TEMPORARY TABLE TEMPPROD
  AS (SELECT * FROM PRODVIEW)
  DEFINITION ONLY
  INCLUDING IDENTITY COLUMN ATTRIBUTES
  INCLUDING COLUMN DEFAULTS;
  ```

  If you want the declared temporary table columns to inherit the defaults for columns of the table or view that is named in the fullselect, specify the INCLUDING COLUMN DEFAULTS clause. If you want
the declared temporary table columns to have default values that correspond to their data types, specify the USING TYPE DEFAULTS clause.

Db2 creates an empty instance of a declared temporary table.

2. Complete one of the following actions:
   - Populate the declared temporary table by using INSERT statements
   - Modify the table using searched or positioned UPDATE or DELETE statements
   - Query the table using SELECT statements
   - Create indexes on the declared temporary table

3. After you run a DECLARE GLOBAL TEMPORARY TABLE statement, the definition of the declared temporary table exists as long as the application process runs. If you need to delete the definition before the application process completes, issue a DROP TABLE statement. For example, to drop the definition of TEMPPROD, run the following statement:

   ```sql
   DROP TABLE SESSION.TEMPPROD;
   ```

**Example**

The ON COMMIT clause that you specify in the DECLARE GLOBAL TEMPORARY TABLE statement determines whether Db2 keeps or deletes all the rows from the table when you run a COMMIT statement in an application with a declared temporary table. ON COMMIT DELETE ROWS, which is the default, causes all rows to be deleted from the table at a commit point, unless a held cursor is open on the table at the commit point. ON COMMIT PRESERVE ROWS causes the rows to remain past the commit point.

For example, Suppose that you run the following statement in an application program:

```sql
EXEC SQL DECLARE GLOBAL TEMPORARY TABLE TEMPPROD
   AS (SELECT * FROM BASEPROD)
   DEFINITION ONLY
   INCLUDING IDENTITY COLUMN ATTRIBUTES
   INCLUDING COLUMN DEFAULTS
   ON COMMIT PRESERVE ROWS;
EXEC SQL INSERT INTO SESSION.TEMPPROD SELECT * FROM BASEPROD;
⋮
EXEC SQL COMMIT;
⋮
```

When Db2 runs the preceding DECLARE GLOBAL TEMPORARY TABLE statement, Db2 creates an empty instance of TEMPPROD. The INSERT statement populates that instance with rows from table BASEPROD. The qualifier, SESSION, must be specified in any statement that references TEMPPROD. When Db2 executes the COMMIT statement, Db2 keeps all rows in TEMPPROD because TEMPPROD is defined with ON COMMIT PRESERVE ROWS. When the program ends, Db2 drops TEMPPROD.

**Related reference**

DECLARE GLOBAL TEMPORARY TABLE (Db2 SQL)

**Providing a unique key for a table**

Use ROWID columns or identity columns to store unique values for each row in a table.

**About this task**

**Question:** How can I provide a unique identifier for a table that has no unique column?

**Answer:** Add a column with the data type ROWID or an identity column. ROWID columns and identity columns contain a unique value for each row in the table. You can define the column as GENERATED ALWAYS, which means that you cannot insert values into the column, or GENERATED BY DEFAULT, which means that Db2 generates a value if you do not specify one. If you define the ROWID or identity column as GENERATED BY DEFAULT, you need to define a unique index that includes only that column to guarantee uniqueness.
Fixing tables with incomplete definitions

If a table has an incomplete definition, you cannot load the table, insert data, retrieve data, update data, or delete data. You can however drop the table, create the primary index, and drop or create other indexes.

Before you begin

To check if a table has an incomplete definition, look at the STATUS column in SYSIBM.SYSTABLES. The value I indicates that the definition is incomplete.

About this task

A table definition is incomplete in any of the following circumstances:

• **If the table is defined with a primary or unique key** and all of the following conditions are true:
  – The table space for the table was explicitly created.
  – The statement is not being run with schema processor.
  – The table does not have a primary or unique index for the defined primary or unique key.

• **If the table has a ROWID column that is defined as generated by default** and all of the following conditions are true:
  – The table space for the table was explicitly created.
  – The SET CURRENT RULES special register is not set to STD.
  – No unique index is defined on the ROWID column.

• **If the table has a LOB column** and all of the following conditions are true:
  – The table space for the table was explicitly created.
  – The SET CURRENT RULES special register is not set to STD.
  – No all auxiliary LOB objects are defined for the LOB column.

Procedure

To complete the definition of a table, use one of the following actions:

• Create a primary index or alter the table to drop the primary key.
• Create a unique index on the unique key or alter the table to drop the unique key.
• Defining a unique index on the ROWID column.
• Create the necessary LOB objects.

Example

To create the primary index for the project activity table, issue the following SQL statement:

```
CREATE UNIQUE INDEX XPROJAC1
ON DSN8C10.PROJACT (PROJNO, ACTNO, ACSTDATE);
```

RENAME TABLE in a table maintenance scenario

The RENAME TABLE statement is useful when you need to temporarily take a table offline for maintenance that involves structural changes to the table. Applications can continue to run against another copy of the table until maintenance is complete.

One way of accomplishing this is refer to the name of the table as an unqualified name in all applications. The unqualified table name is implicitly qualified by the content of the CURRENT SCHEMA special register. You set CURRENT SCHEMA to the schema of the real table to cause applications to access the real table. Before you take the real table offline, you change the CURRENT SCHEMA special register to the name of
the schema for the alternate copy of the table. When all applications are running with the alternate copy of the table, the real table can be modified. An example of such a modification is adding a column to the table.

Later, when table maintenance is complete, you can set the CURRENT SCHEMA special register to the name of the schema for the real table to cause all applications to switch back to using the real table.

**Related reference**
RENAMe (Db2 SQL)
CURRENT SCHEMA (Db2 SQL)

**Dropping tables**
When you drop a table, you delete the data and the table definition. You also delete all synonyms, views, indexes, referential constraints, and check constraints that are associated with that table.

**About this task**
The following SQL statement drops the YEMP table:

```
DROP TABLE YEMP;
```

**Use the DROP TABLE statement with care:** Dropping a table is **not** equivalent to deleting all its rows. When you drop a table, you lose more than its data and its definition. You lose all synonyms, views, indexes, and referential and check constraints that are associated with that table. You also lose all authorities that are granted on the table.

**Related reference**
DROP (Db2 SQL)

**Defining a view**
A **view** is a named specification of a result table. Use views to control which users have access to certain data or to simplify writing SQL statements.

**About this task**
Use the CREATE VIEW statement to define a view and give the view a name, just as you do for a table. The view that is created with the following statement shows each department manager's name with the department data in the DSN8C10.DEPT table.

```
CREATE VIEW VDEPTM AS
  SELECT DEPTNO, MGRNO, LASTNAME, ADMRDEPT
  FROM DSN8C10.DEPT, DSN8C10.EMP
  WHERE DSN8C10.EMP.EMPNO = DSN8C10.DEPT.MGRNO;
```

When a program accesses the data that is defined by a view, Db2 uses the view definition to return a set of rows that the program can access with SQL statements.

To see the departments that are administered by department D01 and the managers of those departments, run the following statement, which returns information from the VDEPTM view:

```
SELECT DEPTNO, LASTNAME
FROM VDEPTM
WHERE ADMRDEPT = 'D01';
```

When you create a view, you can reference the SESSION_USER and CURRENT SQLID special registers in the CREATE VIEW statement. When referencing the view, Db2 uses the value of the SESSION_USER or CURRENT SQLID special register that belongs to the user of the SQL statement (SELECT, UPDATE, INSERT, or DELETE) rather than the creator of the view. In other words, a reference to a special register in a view definition refers to its run time value.
You can specify a period specification for a view, subject to certain restrictions. Also, for a view that references an application-period temporal table or a bitemporal table, you can specify a period clause for an update or delete operation on the view.

A column in a view might be based on a column in a base table that is an identity column. The column in the view is also an identity column, except under any of the following circumstances:

• The column appears more than once in the view.
• The view is based on a join of two or more tables.
• The view is based on the union of two or more tables.
• Any column in the view is derived from an expression that refers to an identity column.

You can use views to limit access to certain kinds of data, such as salary information. Alternatively, you can use the IMPLICITLY HIDDEN clause of a CREATE TABLE statement to define a column of a table to be hidden from some operations.

You can also use views for the following actions:

• Make a subset of a table’s data available to an application. For example, a view based on the employee table might contain rows only for a particular department.
• Combine columns from two or more tables and make the combined data available to an application. By using a SELECT statement that matches values in one table with those in another table, you can create a view that presents data from both tables. However, you can only select data from this type of view. You cannot update, delete, or insert data using a view that joins two or more tables.
• Combine rows from two or more tables and make the combined data available to an application. By using two or more subselects that are connected by a set operator such as UNION, you can create a view that presents data from several tables. However, you can only select data from this type of view. You cannot update, delete, or insert data using a view that contains UNION operations.
• Present computed data, and make the resulting data available to an application. You can compute such data using any function or operation that you can use in a SELECT statement.

Related tasks
Changing data by using views that reference temporal tables (Db2 Administration Guide)

Related reference
CREATE VIEW (Db2 SQL)

Related information
Implementing Db2 views (Db2 Administration Guide)

Views

A view does not contain data; it is a stored definition of a set of rows and columns. A view can present any or all of the data in one or more tables.

Although you cannot modify an existing view, you can drop it and create a new one if your base tables change in a way that affects the view. Dropping and creating views does not affect the base tables or their data.

Restrictions when changing data through a view

Some views are read-only and thus cannot be used to update the table data. For those views that are updatable, several restrictions apply.

Consider the following restrictions when changing data through a view:

• You must have the appropriate authorization to insert, update, or delete rows using the view.
• When you use a view to insert a row into a table, the view definition must specify all the columns in the base table that do not have a default value. The row that is being inserted must contain a value for each of those columns.
• Views that you can use to update data are subject to the same referential constraints and check
costs as the tables that you used to define the views.

You can use the WITH CHECK option of the CREATE VIEW statement to specify the constraint that every
row that is inserted or updated through the view must conform to the definition of the view. You can
select every row that is inserted or updated through a view that is created with the WITH CHECK option.

• For an update operation on a view that references an application-period temporal table or a bitemporal
  table, the result table of the outer fullselect of the view definition, explicitly or implicitly, must include
  the start and end columns of the BUSINESS_TIME period.

• For an update or delete operation on a view that references an application-period temporal table or a
  bitemporal table, the view must not be defined with an INSTEAD OF trigger.

For complex views, you can make insert, update and delete operations possible by defining INSTEAD OF
triggers.

**Related tasks**

Inserting, updating, and deleting data in views by using INSTEAD OF triggers

INSTEAD OF triggers are triggers that execute instead of the INSERT, UPDATE, or DELETE statement that
activates the trigger. You can define these triggers on views only. Use INSTEAD OF triggers to insert,
update, and delete data in complex views.

**Changing data by using views that reference temporal tables (Db2 Administration Guide)**

**Related reference**

CREATE VIEW (Db2 SQL)

**Dropping a view**

When you drop a view, you also drop all views that are defined on that view. The base table is not
affected.

**Example**

The following SQL statement drops the VDEPTM view:

```sql
DROP VIEW VDEPTM;
```

**Creating a common table expression**

Creating a common table expression saves you the overhead of creating and dropping a regular view that
you need to use only once. Also, during statement preparation, Db2 does not need to access the catalog
for the view, which saves you additional overhead.

**About this task**

Use the WITH clause to create a common table expression.

**Procedure**

To created a common table expression use one of the following approaches:

• Specify a WITH clause at the beginning of a SELECT statement.

For example, the following statement finds the department with the highest total pay. The query
involves two levels of aggregation. First, you need to determine the total pay for each department by
using the SUM function and order the results by using the GROUP BY clause. You then need to find the
department with highest total pay based on the total pay for each department.

```sql
WITH DTOTAL (workdept, totalpay) AS
  (SELECT deptno, sum(salary+bonus)
   FROM  DSN8810.EMP
   GROUP BY workdept)
SELECT workdept
FROM DTOTAL
```
WHERE totalpay = (SELECT max(totalpay)
                    FROM DTOTAL);

The result table for the common table expression, DTOTAL, contains the department number and total pay for each department in the employee table. The fullselect in the previous example uses the result table for DTOTAL to find the department with the highest total pay. The result table for the entire statement looks similar to the following results:

<table>
<thead>
<tr>
<th>WORKDEPT</th>
</tr>
</thead>
<tbody>
<tr>
<td>D11</td>
</tr>
</tbody>
</table>

- Use common table expressions by specifying WITH before a fullselect in a CREATE VIEW statement. This technique is useful if you need to use the results of a common table expression in more than one query. For example, the following statement finds the departments that have a greater-than-average total pay and saves the results as the view RICH_DEPT:

```
CREATE VIEW RICH_DEPT (workdept) AS
    WITH DTOTAL (workdept, totalpay) AS
        (SELECT workdept, sum(salary+bonus)
         FROM DSN8C10.EMP
         GROUP BY workdept)
    SELECT workdept
    FROM DTOTAL
    WHERE totalpay > (SELECT AVG(totalpay)
                      FROM DTOTAL);
```

The fullselect in the previous example uses the result table for DTOTAL to find the departments that have a greater-than-average total pay. The result table is saved as the RICH_DEPT view and looks similar to the following results:

<table>
<thead>
<tr>
<th>WORKDEPT</th>
</tr>
</thead>
<tbody>
<tr>
<td>A00</td>
</tr>
<tr>
<td>D11</td>
</tr>
<tr>
<td>D21</td>
</tr>
</tbody>
</table>

- Use common table expressions by specifying WITH before a fullselect in an INSERT statement. For example, the following statement uses the result table for VITALDEPT to find the manager's number for each department that has a greater-than-average number of senior engineers. Each manager's number is then inserted into the vital_mgr table.

```
INSERT INTO vital_mgr (mgrno)
    WITH VITALDEPT (workdept, se_count) AS
        (SELECT workdept, count(*)
         FROM DSN8C10.EMP
         WHERE job = 'senior engineer'
         GROUP BY workdept)
    SELECT d.manager
    FROM DSN8C10.DEPT d, VITALDEPT s
    WHERE d.workdept = s.workdept
    AND s.se_count  >  (SELECT  AVG(se_count)
                         FROM VITALDEPT);
```

**Related reference**

common-table-expression (Db2 SQL)

**Common table expressions**

A common table expression is like a temporary view that is defined and used for the duration of an SQL statement.

You can define a common table expression wherever you can have a fullselect statement. For example, you can include a common table expression in a SELECT, INSERT, SELECT INTO, or CREATE VIEW statement.
Each common table expression must have a unique name and be defined only once. However, you can reference a common table expression many times in the same SQL statement. Unlike regular views or nested table expressions, which derive their result tables for each reference, all references to common table expressions in a given statement share the same result table.

You can use a common table expression in the following situations:

- When you want to avoid creating a view (when general use of the view is not required, and positioned updates or deletes are not used)
- When the result table is based on host variables
- When the same result table needs to be shared in a fullselect
- When the results need to be derived using recursion

**Related reference**
common-table-expression (Db2 SQL)

### Examples of recursive common table expressions

Recursive SQL is very useful in bill of materials (BOM) applications.

Consider a table of parts with associated subparts and the quantity of subparts required by each part. For more information about recursive SQL, refer to “Creating recursive SQL by using common table expressions” on page 388.

For the examples in this topic, create the following table:

```
CREATE TABLE PARTLIST
(PART VARCHAR(8),
SUBPART VARCHAR(8),
QUANTITY INTEGER);
```

Assume that the PARTLIST table is populated with the values that are in the following table:

<table>
<thead>
<tr>
<th>PART</th>
<th>SUBPART</th>
<th>QUANTITY</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>01</td>
<td>5</td>
</tr>
<tr>
<td>00</td>
<td>05</td>
<td>3</td>
</tr>
<tr>
<td>01</td>
<td>02</td>
<td>2</td>
</tr>
<tr>
<td>01</td>
<td>03</td>
<td>3</td>
</tr>
<tr>
<td>01</td>
<td>04</td>
<td>4</td>
</tr>
<tr>
<td>01</td>
<td>06</td>
<td>3</td>
</tr>
<tr>
<td>02</td>
<td>05</td>
<td>7</td>
</tr>
<tr>
<td>02</td>
<td>06</td>
<td>6</td>
</tr>
<tr>
<td>03</td>
<td>07</td>
<td>6</td>
</tr>
<tr>
<td>04</td>
<td>08</td>
<td>10</td>
</tr>
<tr>
<td>04</td>
<td>09</td>
<td>11</td>
</tr>
<tr>
<td>05</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>05</td>
<td>11</td>
<td>10</td>
</tr>
<tr>
<td>06</td>
<td>12</td>
<td>10</td>
</tr>
<tr>
<td>06</td>
<td>13</td>
<td>10</td>
</tr>
</tbody>
</table>
Example 1: Single level explosion:

Single level explosion answers the question, "What parts are needed to build the part identified by '01'?". The list will include the direct subparts, subparts of the subparts and so on. However, if a part is used multiple times, its subparts are only listed once.

```
WITH RPL (PART, SUBPART, QUANTITY) AS
  (SELECT ROOT.PART, ROOT.SUBPART, ROOT.QUANTITY
    FROM PARTLIST ROOT
    WHERE ROOT.PART = '01'
    UNION ALL
    SELECT CHILD.PART, CHILD.SUBPART, CHILD.QUANTITY
    FROM RPL PARENT, PARTLIST CHILD
    WHERE PARENT.SUBPART = CHILD.PART)
SELECT DISTINCT PART, SUBPART, QUANTITY
FROM RPL
ORDER BY PART, SUBPART, QUANTITY;
```

The preceding query includes a common table expression, identified by the name RPL, that expresses the recursive part of this query. It illustrates the basic elements of a recursive common table expression.

The first operand (fullselect) of the UNION, referred to as the initialization fullselect, gets the direct subparts of part '01'. The FROM clause of this fullselect refers to the source table and will never refer to itself (RPL in this case). The result of this first fullselect goes into the common table expression RPL. As in this example, the UNION must always be a UNION ALL.

The second operand (fullselect) of the UNION uses RPL to compute subparts of subparts by using the FROM clause to refer to the common table expression RPL and the source table PARTLIST with a join of a part from the source table (child) to a subpart of the current result contained in RPL (parent). The result goes then back to RPL again. The second operand of UNION is used repeatedly until no more subparts exist.

The SELECT DISTINCT in the main fullselect of this query ensures the same part/subpart is not listed more than once.

The result of the query is shown in the following table:

<table>
<thead>
<tr>
<th>PART</th>
<th>SUBPART</th>
<th>QUANTITY</th>
</tr>
</thead>
<tbody>
<tr>
<td>01</td>
<td>02</td>
<td>2</td>
</tr>
<tr>
<td>01</td>
<td>03</td>
<td>3</td>
</tr>
<tr>
<td>01</td>
<td>04</td>
<td>4</td>
</tr>
<tr>
<td>01</td>
<td>06</td>
<td>3</td>
</tr>
<tr>
<td>02</td>
<td>05</td>
<td>7</td>
</tr>
<tr>
<td>02</td>
<td>06</td>
<td>6</td>
</tr>
<tr>
<td>03</td>
<td>07</td>
<td>6</td>
</tr>
<tr>
<td>04</td>
<td>08</td>
<td>10</td>
</tr>
<tr>
<td>04</td>
<td>09</td>
<td>11</td>
</tr>
<tr>
<td>05</td>
<td>10</td>
<td>10</td>
</tr>
</tbody>
</table>
Observe in the result that part '01' contains subpart '02' which contains subpart '06' and so on. Further, notice that part '06' is reached twice, once through part '01' directly and another time through part '02'. In the output, however, the subparts of part '06' are listed only once (this is the result of using a SELECT DISTINCT).

Remember that with recursive common table expressions it is possible to introduce an infinite loop. In this example, an infinite loop would be created if the search condition of the second operand that joins the parent and child tables was coded as follows:

```sql
WHERE PARENT.SUBPART = CHILD.SUBPART
```

This infinite loop is created by not coding what is intended. You should carefully determining what to code so that there is a definite end of the recursion cycle.

The result produced by this example could be produced in an application program without using a recursive common table expression. However, such an application would require coding a different query for every level of recursion. Furthermore, the application would need to put all of the results back in the database to order the final result. This approach complicates the application logic and does not perform well. The application logic becomes more difficult and inefficient for other bill of material queries, such as summarized and indented explosion queries.

**Example 2: Summarized explosion:**

A summarized explosion answers the question, "What is the total quantity of each part required to build part '01'?" The main difference from a single level explosion is the need to aggregate the quantities. A single level explosion indicates the quantity of subparts required for the part whenever it is required. It does not indicate how many of each subpart is needed to build part '01'.

```sql
WITH RPL (PART, SUBPART, QUANTITY) AS
  (SELECT ROOT.PART, ROOT.SUBPART, ROOT.QUANTITY
   FROM PARTLIST ROOT
   WHERE ROOT.PART = '01'
   UNION ALL
   SELECT PARENT.PART, CHILD.SUBPART, PARENT.QUANTITY*CHILD.QUANTITY
   FROM RPL PARENT, PARTLIST CHILD
   WHERE PARENT.SUBPART = CHILD.PART)
SELECT PART, SUBPART, SUM(QUANTITY) AS "Total QTY Used"
FROM RPL
GROUP BY PART, SUBPART
ORDER BY PART, SUBPART;
```

In the preceding query, the select list of the second operand of the UNION in the recursive common table expression, identified by the name RPL, shows the aggregation of the quantity. To determine how many of each subpart is used, the quantity of the parent is multiplied by the quantity per parent of a child. If a part is used multiple times in different places, it requires another final aggregation. This is done by the grouping the parts and subparts in the common table expression RPL and using the SUM column function in the select list of the main fullselect.

The result of the query is shown in the following table:
Table 39. Result table for example 2

<table>
<thead>
<tr>
<th>PART</th>
<th>SUBPART</th>
<th>Total QTY Used</th>
</tr>
</thead>
<tbody>
<tr>
<td>01</td>
<td>02</td>
<td>2</td>
</tr>
<tr>
<td>01</td>
<td>03</td>
<td>3</td>
</tr>
<tr>
<td>01</td>
<td>04</td>
<td>4</td>
</tr>
<tr>
<td>01</td>
<td>05</td>
<td>14</td>
</tr>
<tr>
<td>01</td>
<td>06</td>
<td>15</td>
</tr>
<tr>
<td>01</td>
<td>07</td>
<td>18</td>
</tr>
<tr>
<td>01</td>
<td>08</td>
<td>40</td>
</tr>
<tr>
<td>01</td>
<td>09</td>
<td>44</td>
</tr>
<tr>
<td>01</td>
<td>10</td>
<td>140</td>
</tr>
<tr>
<td>01</td>
<td>11</td>
<td>140</td>
</tr>
<tr>
<td>01</td>
<td>12</td>
<td>294</td>
</tr>
<tr>
<td>01</td>
<td>13</td>
<td>150</td>
</tr>
<tr>
<td>01</td>
<td>14</td>
<td>144</td>
</tr>
</tbody>
</table>

Consider the total quantity for subpart '06'. The value of 15 is derived from a quantity of 3 directly for part '01' and a quantity of 6 for part '02' which is needed two times by part '01'.

Example 3: Controlling depth:

You can control the depth of a recursive query to answer the question, "What are the first two levels of parts that are needed to build part '01'?" For the sake of clarity in this example, the level of each part is included in the result table.

```sql
WITH RPL (LEVEL, PART, SUBPART, QUANTITY) AS
    (
        SELECT 1, ROOT.PART, ROOT.SUBPART, ROOT.QUANTITY
        FROM PARTLIST ROOT
        WHERE ROOT.PART = '01'
        UNION ALL
        SELECT PARENT.LEVEL+1, CHILD.PART, CHILD.SUBPART, CHILD.QUANTITY
        FROM RPL PARENT, PARTLIST CHILD
        WHERE PARENT.SUBPART = CHILD.PART
        AND PARENT.LEVEL < 2
    )
SELECT PART, LEVEL, SUBPART, QUANTITY
FROM RPL;
```

This query is similar to the query in example 1. The column LEVEL is introduced to count the level each subpart is from the original part. In the initialization fullselect, the value for the LEVEL column is initialized to 1. In the subsequent fullselect, the level from the parent table increments by 1. To control the number of levels in the result, the second fullselect includes the condition that the level of the parent must be less than 2. This ensures that the second fullselect only processes children to the second level.

The result of the query is shown in the following table:

Table 40. Result table for example 3

<table>
<thead>
<tr>
<th>PART</th>
<th>LEVEL</th>
<th>SUBPART</th>
<th>QUANTITY</th>
</tr>
</thead>
<tbody>
<tr>
<td>01</td>
<td>1</td>
<td>02</td>
<td>2</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>03</td>
<td>3</td>
</tr>
</tbody>
</table>
Creating a trigger

A trigger is a set of SQL statements that execute when a certain event occurs in a table or view. Use triggers to control changes in Db2 databases. Triggers are more powerful than constraints because they can monitor a broader range of changes and perform a broader range of actions. This topic describes support for advanced triggers.

About this task

Using triggers for active data:

For example, a constraint can disallow an update to the salary column of the employee table if the new value is over a certain amount. A trigger can monitor the amount by which the salary changes, as well as the salary value. If the change is above a certain amount, the trigger might substitute a valid value and call a user-defined function to send a notice to an administrator about the invalid update.

Triggers also move application logic into Db2, which can result in faster application development and easier maintenance. For example, you can write applications to control salary changes in the employee table, but each application program that changes the salary column must include logic to check those changes. A better method is to define a trigger that controls changes to the salary column. Then Db2 does the checking for any application that modifies salaries.

Example of creating and using a trigger:

Triggers automatically execute a set of SQL statements whenever a specified event occurs. These SQL statements can perform tasks such as validation and editing of table changes, reading and modifying tables, or invoking functions or stored procedures that perform operations both inside and outside Db2.

You create triggers using the CREATE TRIGGER statement. The following figure shows an example of a CREATE TRIGGER statement.

```
CREATE TRIGGER REORDER
AFTER UPDATE OF ON_HAND, MAX_STOCKED ON PARTS
REFERENCING NEW AS N_ROW
FOR EACH ROW
WHEN (N_ROW.ON_HAND < 0.10 * N_ROW.MAX_STOCKED)
BEGIN ATOMIC
CALL ISSUE_SHIP_REQUEST(N_ROW.MAX_STOCKED - N_ROW.ON_HAND,
```

### Table 40. Result table for example 3 (continued)

<table>
<thead>
<tr>
<th>PART</th>
<th>LEVEL</th>
<th>SUBPART</th>
<th>QUANTITY</th>
</tr>
</thead>
<tbody>
<tr>
<td>01</td>
<td>1</td>
<td>04</td>
<td>4</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>06</td>
<td>3</td>
</tr>
<tr>
<td>02</td>
<td>2</td>
<td>05</td>
<td>7</td>
</tr>
<tr>
<td>02</td>
<td>2</td>
<td>06</td>
<td>6</td>
</tr>
<tr>
<td>03</td>
<td>2</td>
<td>07</td>
<td>6</td>
</tr>
<tr>
<td>04</td>
<td>2</td>
<td>08</td>
<td>10</td>
</tr>
<tr>
<td>04</td>
<td>2</td>
<td>09</td>
<td>11</td>
</tr>
<tr>
<td>06</td>
<td>2</td>
<td>12</td>
<td>10</td>
</tr>
<tr>
<td>06</td>
<td>2</td>
<td>13</td>
<td>10</td>
</tr>
</tbody>
</table>
The parts of this trigger are:

1. Trigger name (REORDER)
2. Trigger activation time (AFTER)
3. Triggering event (UPDATE)
4. Subject table name (PARTS)
5. New transition variable correlation name (N_ROW)
6. Granularity (FOR EACH ROW)
7. Trigger condition (WHEN...)
8. Trigger body (BEGIN ATOMIC...END;)

When you execute this CREATE TRIGGER statement, Db2 creates a trigger package called REORDER and associates the trigger package with table PARTS. Db2 records the timestamp when it creates the trigger. If you define other triggers on the PARTS table, Db2 uses this timestamp to determine which trigger to activate first when the triggering event occurs. The trigger is now ready to use.

After Db2 updates columns ON_HAND or MAX_STOCKED in any row of table PARTS, trigger REORDER is activated. The trigger calls a stored procedure called ISSUE_SHIP_REQUEST if, after a row is updated, the quantity of parts on hand is less than 10% of the maximum quantity stocked. In the trigger condition, the qualifier N_ROW represents a value in a modified row after the triggering event.

When you no longer want to use trigger REORDER, you can delete the trigger by executing the statement:

```
DROP TRIGGER REORDER;
```

Executing this statement drops trigger REORDER and its associated trigger package named REORDER. If you drop table PARTS, Db2 also drops trigger REORDER and its trigger package.

**Parts of a trigger:**

A trigger contains the following parts:

- trigger name
- subject table
- trigger activation time
- triggering event
- granularity
- correlation names for transition variables and transition tables
- triggered action that consists of an optional search condition and a trigger body

**Trigger name:**

Specify a name for your trigger. You can use a qualifier or let Db2 determine the qualifier. When Db2 creates a trigger package for the trigger, it uses the same qualifier as the collection ID of the trigger package.

**Subject table or view:**
When you perform an insert, update, or delete operation on this table or view, the trigger is activated. You must name a local table or view in the CREATE TRIGGER statement. You cannot define a trigger on a catalog table.

**Trigger activation time:**

The choices for trigger activation time are BEFORE, AFTER, and INSTEAD OF. BEFORE and AFTER triggers can be defined for a table. INSTEAD OF triggers can be defined for a view.

BEFORE means that the trigger is activated before Db2 makes any changes to the subject table, and that the triggered action does not activate any other triggers. AFTER means that the trigger is activated after Db2 makes changes to the subject table and can activate other triggers. INSTEAD OF means that the trigger is activated when there is an attempt to change the subject view. Triggers with an activation time of BEFORE are known as before triggers. Triggers with an activation time of AFTER are known as after triggers. Triggers with an activation time of INSTEAD OF are known as instead of triggers.

**Triggering event:**

Every trigger is associated with an event. A trigger is activated when the triggering event occurs in the subject table or view. The triggering event is one of the following SQL operations:

- insert
- update
- delete

A triggering event can also be an update or delete operation that occurs as the result of a referential constraint with ON DELETE SET NULL or ON DELETE CASCADE.

A trigger can be activated by a MERGE statement for delete, insert, and update operations.

Triggers are not activated as the result of updates made to tables by Db2 utilities, with the exception of the LOAD utility when it is specified with the RESUME YES and SHRLEVEL CHANGE options.

When the triggering event for a trigger is an update operation, the trigger is called an update trigger. Similarly, triggers for insert operations are called insert triggers, and triggers for delete operations are called delete triggers.

The SQL statement that performs the triggering SQL operation is called the triggering SQL statement. Each triggering event is associated with one subject table or view and one SQL operation.

The following trigger is defined with an insert triggering event:

```sql
CREATE TRIGGER NEW_HIRE
AFTER INSERT ON EMP
FOR EACH ROW
BEGIN ATOMIC
    UPDATE COMPANY_STATS SET NBEMP = NBEMP + 1;
END
```

If the triggering SQL operation is an update operation, the event can be associated with specific columns of the subject table. In this case, the trigger is activated only if the update operation updates any of the specified columns.

The following trigger, PAYROLL1, which invokes user-defined function named PAYROLL_LOG, is activated only if an update operation is performed on the SALARY or BONUS column of table PAYROLL:

```sql
CREATE TRIGGER PAYROLL1
AFTER UPDATE OF SALARY, BONUS ON PAYROLL
FOR EACH STATEMENT
BEGIN ATOMIC
    VALUES(PAYROLL_LOG(USER, 'UPDATE', CURRENT TIME, CURRENT DATE));
END
```

**Granularity:**

The triggering SQL statement might modify multiple rows in the table. The granularity of the trigger determines whether the trigger is activated only once for the triggering SQL statement or once for every row that the SQL statement modifies. The granularity values are:
• FOR EACH ROW

The trigger is activated once for each row that Db2 modifies in the subject table or view. If the triggering SQL statement modifies no rows, the trigger is not activated. However, if the triggering SQL statement updates a value in a row to the same value, the trigger is activated. For example, if an UPDATE trigger is defined on table COMPANY_STATS, the following SQL statement will activate the trigger.

```sql
UPDATE COMPANY_STATS SET NBEMP = NBEMP;
```

• FOR EACH STATEMENT

The trigger is activated once when the triggering SQL statement executes. The trigger is activated even if the triggering SQL statement modifies no rows.

Triggers with a granularity of FOR EACH ROW are known as row triggers. Triggers with a granularity of FOR EACH STATEMENT are known as statement triggers. Statement triggers can only be after triggers.

The following statement is an example of a row trigger:

```sql
CREATE TRIGGER NEW_HIRE
AFTER INSERT ON EMP
FOR EACH ROW
BEGIN ATOMIC
    UPDATE COMPANY_STATS SET NBEMP = NBEMP + 1;
END
```

Trigger NEW_HIRE is activated once for every row inserted into the employee table.

**Transition variables:**

When you code a row trigger, you might need to refer to the values of columns in each updated row of the subject table or view. To do this, specify a correlation name (to use when referencing transition variables) in the REFERENCING clause of your CREATE TRIGGER statement. The two types of transition variables are:

• Old transition variables capture the values of columns before the triggering SQL statement updates them. You can use the REFERENCING OLD clause to define a correlation name for referencing old transition variables for update and delete triggers.

• New transition variables capture the values of columns after the triggering SQL statement updates them. You can use the REFERENCING NEW clause to define a correlation name for referencing new transition variables for update and insert triggers.

Transition variables can be referenced anywhere in an SQL statement where an expression or variable can be specified in triggers. See References to SQL parameters and variables (Db2 SQL) for more information.

The following example uses transition variables and invocations of the IDENTITY_VAL_LOCAL function to access values that are assigned to identity columns.

Suppose that you have created tables T and S, with the following definitions:

```sql
CREATE TABLE T
(ID SMALLINT GENERATED BY DEFAULT AS IDENTITY (START WITH 100),
 C2 SMALLINT,
 C3 SMALLINT,
 C4 SMALLINT);
```

```sql
CREATE TABLE S
(ID SMALLINT GENERATED ALWAYS AS IDENTITY,
 C1 SMALLINT);
```

Define a before insert trigger on T that uses the IDENTITY_VAL_LOCAL built-in function to retrieve the current value of identity column ID, and uses transition variables to update the other columns of T with the identity column value.

```sql
CREATE TRIGGER TR1
NO CASCADE BEFORE INSERT
ON T REFERENCING NEW AS N
FOR EACH ROW MODE DB2SQL
```
BEGIN ATOMIC
SET N.C3 =N.ID;
SET N.C4 =IDENTITY_VAL_LOCAL();
SET N.ID =N.C2 *10;
SET N.C2 =IDENTITY_VAL_LOCAL();
END

Now suppose that you execute the following INSERT statement:

```
INSERT INTO S (C1) VALUES (5);
```

This statement inserts a row into S with a value of 5 for column C1 and a value of 1 for identity column ID. Next, suppose that you execute the following SQL statement, which activates trigger TR1:

```
INSERT INTO T (C2)
VALUES (IDENTITY_VAL_LOCAL());
```

This insert statement, and the subsequent activation of trigger TR1, have the following results:

- The INSERT statement obtains the most recent value that was assigned to an identity column (1), and inserts that value into column C2 of table T. 1 is the value that Db2 inserted into identity column ID of table S.
- When the INSERT statement executes, Db2 inserts the value 100 into identity column ID column of C2.
- The first statement in the body of trigger TR1 inserts the value of transition variable N.ID (100) into column C3. N.ID is the value that identity column ID contains after the INSERT statement executes.
- The second statement in the body of trigger TR1 inserts the null value into column C4. By definition, the result of the IDENTITY_VAL_LOCAL function in the triggered action of a before insert trigger is the null value.
- The third statement in the body of trigger TR1 inserts 10 times the value of transition variable N.C2 (10*1) into identity column ID of table T. N.C2 is the value that column C2 contains after the INSERT is executed.
- The fourth statement in the body of trigger TR1 inserts the null value into column C2. By definition, the result of the IDENTITY_VAL_LOCAL function in the triggered action of a before insert trigger is the null value.

**Transition tables:**

If you want to refer to the entire set of rows that a triggering SQL statement modifies, rather than to individual rows, use a transition table. Like transition variables, a correlation name (to refer to the columns of the transition table) can appear in the REFERENCING clause of a CREATE TRIGGER statement. The names for those columns are the same as the name of the column in the table or view that the trigger is defined for. Transition tables are valid for both row triggers and statement triggers. The two types of transition tables are:

- Old transition tables, specified with the OLD TABLE transition-table-name clause, capture the values of columns before the triggering SQL statement updates them. You can define old transition tables for update and delete triggers.
- New transition tables, specified with the NEW TABLE transition-table-name clause, capture the values of columns after the triggering SQL statement updates them. You can define new transition variables for update and insert triggers.

The scope of old and new transition table names is the trigger body. If correlation names are specified for both old and new transition variables in the trigger, a reference to a transition variable must be qualified with the associated correlation name. The name of a transition variable can also be the same as the name of an SQL variable or global variable, or the name of a column in a table or view that is referenced in the trigger. Names that are the same should be explicitly qualified. Qualifying a name can clarify whether the name refers to a column, global variable, SQL variable, SQL parameter, or transition variable. To avoid ambiguity, qualify a transition variable with the correlation name specified in the REFERENCING clause in the CREATE TRIGGER or ALTER TRIGGER statement that defined the trigger.

The following example accesses a new transition table to capture the set of rows that are inserted into the INVOICE table:
CREATE TRIGGER LRG_ORDR
AFTER INSERT ON INVOICE
REFERENCING NEW TABLE AS N_TABLE
FOR EACH STATEMENT
BEGIN ATOMIC
    SELECT LARGE_ORDER_ALERT(CUST_NO,
                             TOTAL_PRICE, DELIVERY_DATE)
    FROM N_TABLE WHERE TOTAL_PRICE > 10000;
END

The SELECT statement in LRG_ORDER causes user-defined function LARGE_ORDER_ALERT to execute for each row in transition table N_TABLE that satisfies the WHERE clause (TOTAL_PRICE > 10000).

**Triggered action:**

When a trigger is activated, a triggered action occurs. Every trigger has one triggered action, which consists of an optional trigger condition and a trigger body.

**Trigger condition:**

If you want the triggered action to occur only when certain conditions are true, code a trigger condition. A trigger condition is similar to a predicate in a SELECT, except that the trigger condition begins with WHEN, rather than WHERE. If you do not include a trigger condition in your triggered action, the trigger body executes every time the trigger is activated.

For a row trigger, Db2 evaluates the trigger condition once for each modified row of the subject table.

For a statement trigger, Db2 evaluates the trigger condition once for each execution of the triggering SQL statement.

If the trigger condition of a before trigger has a fullselect, the fullselect cannot reference the subject table.

The following example shows a trigger condition that causes the trigger body to execute only when the number of ordered items is greater than the number of available items:

CREATE TRIGGER CK_AVAIL
BEFORE INSERT ON ORDERS
REFERENCING NEW AS NEW_ORDER
FOR EACH ROW
WHEN (NEW_ORDER.QUANTITY >
    (SELECT ON_HAND FROM PARTS
     WHERE NEW_ORDER.PARTNO=PARTS.PARTNO))
BEGIN ATOMIC
    VALUES(ORDER_ERROR(NEW_ORDER.PARTNO,
                        NEW_ORDER.QUANTITY));
END

**Trigger body:**

In the trigger body, you code the SQL statements that you want to execute whenever the trigger condition is true. The trigger body can include a single SQL-control-statement, including a compound statement, or triggered-SQL-statement that is to be executed for the triggered-action. The statements that you can use in a trigger body depend on the activation time of the trigger. See CREATE TRIGGER (advanced) (Db2 SQL) and SQL procedural language (SQL PL) (Db2 SQL) for more information about defining SQL triggers. Use control statements to develop triggers that contain logic.

Because you can include INSERT, DELETE, UPDATE, and MERGE statements in your trigger body, execution of the trigger body might cause activation of other triggers. See “Trigger cascading” on page 160 for more information.

**Examples**

**Example 1**

Define a trigger to increment the count of employees when a new employee is hired. The following example also explains how to determine why an SQL statement is allowed in the trigger.
The UPDATE statement is an SQL statement that is allowed because it is listed in the syntax diagram for triggered-SQL-statement.

Example 2
Define a trigger to return an error condition and back out any changes that are made by the trigger, as well as actions that result from referential constraints on the subject table. Use the SIGNAL statement to indicate the error information to be returned. When Db2 executes the SIGNAL statement, it returns an SQLCA to the application with SQLCODE -438. The SQLCA also includes the following values, which you supply in the SIGNAL statement:

- A 5-character value that Db2 uses as the SQLSTATE
- An error message that Db2 places in the SQLERRMC field

In the following example, the SIGNAL statement causes Db2 to return an SQLCA with SQLSTATE 75001 and terminate the salary update operation if an employee's salary increase is over 20%:

```
CREATE TRIGGER SAL_ADJ
BEFORE UPDATE OF SALARY ON EMP
REFERENCING OLD AS OLD_EMP
NEW AS NEW_EMP
FOR EACH ROW
WHEN (NEW_EMP.SALARY > (OLD_EMP.SALARY * 1.20))
BEGIN ATOMIC
  SIGNAL SQLSTATE '75001'
  ('Invalid Salary Increase - Exceeds 20%');
END
```

Example 3
Define a trigger to assign the current date to the HIRE_DATE column when a row is inserted into the EMP table. Because before triggers operate on rows of a table before those rows are modified, you cannot perform operations in the body of a before trigger that directly modify the subject table. You can, however, use the SET assignment-statement to modify the values in a row before those values go into the table. For example, this trigger uses a new transition variable (NEW_VAR.HIRE_DATE) to assign today's date for the new employee's hire date:

```
CREATE TRIGGER HIREDATE
NO CASCADE BEFORE INSERT ON EMP
REFERENCING NEW AS NEW_VAR
FOR EACH ROW
BEGIN ATOMIC
  SET NEW_VAR.HIRE_DATE = CURRENT_DATE;
END
```

Example 4
In the following example, table CLASS_SCHED contains a row for the class schedule of each class at a school. When a class schedule row is added to the table, trigger VALIDATE_SCHED is activated. In the trigger, SQL control statements are used to check for and respond to the following errors in the class start and end times:

<table>
<thead>
<tr>
<th>Type of error</th>
<th>Response</th>
</tr>
</thead>
<tbody>
<tr>
<td>End time is null</td>
<td>Make the ending time one hour after the starting time</td>
</tr>
<tr>
<td>End time is later than 9:00 p.m.</td>
<td>Issue an error message</td>
</tr>
<tr>
<td>Start day is on a weekend</td>
<td>Issue an error message</td>
</tr>
</tbody>
</table>

```
CREATE TRIGGER VALIDATE_SCHED
BEFORE INSERT ON CLASS_SCHED
REFERENCING NEW AS N
FOR EACH ROW
VS: BEGIN
```

Chapter 3. Db2 SQL programming 155
The SQL trigger has the following statements:

- The IF statements (1) and the SIGNAL statements (2) are SQL control statements.
- The SET assignment statement (3) is an SQL control statement that assigns values to variables.

Related tasks
Obfuscating source code of SQL procedures, SQL functions, and triggers (Db2 Administration Guide)

Related reference
CREATE TRIGGER (advanced) (Db2 SQL)
CREATE TRIGGER (basic) (Db2 SQL)
LOAD (Db2 Utilities)

Invoking a stored procedure or user-defined function from a trigger

A trigger body can include only SQL statements. To perform actions or use logic that is not available in SQL statements, create user-defined functions or stored procedures. Then invoke them from within the trigger body.

About this task
Introductory concepts
Triggers (Introduction to Db2 for z/OS)

Restriction: You cannot include INSERT, UPDATE, DELETE, or MERGE statements in stored procedures or user-defined functions that are invoked by a BEFORE TRIGGER. These actions are not allowed, because BEFORE triggers must not modify any table.

Procedure
To invoke a stored procedure or user-defined function from a trigger:
1. Ensure that the stored procedure or user-defined function is defined before the trigger is defined.

   - Define procedures by using the CREATE PROCEDURE statement.
   - Define triggers by using the CREATE FUNCTION statement.

2. Invoke the user-defined function or stored procedure by performing one of the following actions:

   - To invoke a user-defined function, include the user-defined function in one of the following statements in the trigger:

     SELECT statement
     Use a SELECT statement to execute the function conditionally. The number of times that the user-defined function executes depends on the number of rows in the result table of the SELECT statement. For example, in the following trigger, the SELECT statement invokes user-defined function LARGE_ORDER_ALERT. This function executes once for each row in transition table N_TABLE with an order price of more than 10000:

     ```sql
     CREATE TRIGGER LRG_ORDR
     AFTER INSERT ON INVOICE
     REFERENCING NEW TABLE AS N_TABLE
     FOR EACH STATEMENT MODE DB2SQL
     BEGIN ATOMIC
     ```
VALUES statement

Use the VALUES statement to execute a function unconditionally. The function executes once for each execution of a statement trigger or once for each row in a row trigger. In the following example, user-defined function PAYROLL_LOG executes every time the trigger PAYROLL1 is activated. This trigger is activated when an update operation occurs.

CREATE TRIGGER PAYROLL1
AFTER UPDATE ON PAYROLL
FOR EACH STATEMENT MODE DB2SQL
BEGIN ATOMIC
    VALUES(PAYROLL_LOG(USER, 'UPDATE',
                CURRENT TIME, CURRENT DATE));
END

To invoke a stored procedure, include a CALL statement in the trigger. The parameters of this stored procedure call must be constants, transition variables, table locators, or expressions.

If the parameter is a transition variable or table locator, and the CALL statement is in a BEFORE or AFTER trigger, Db2 returns a warning.

To pass transition tables from the trigger to the user-defined function or stored procedure, use table locators.

When you call a user-defined function or stored procedure from a trigger, you might want to give the function or procedure access to the entire set of modified rows. In this case, use table locators to pass a pointer to the old or new transition table.

Most of the code for using a table locator is in the function or stored procedure that receives the locator.

To pass the transition table from a trigger, specify the parameter TABLE transition-table-name when you invoke the function or stored procedure. This parameter causes Db2 to pass a table locator for the transition table to the user-defined function or stored procedure. For example, the following trigger passes a table locator for a transition table NEWEMPS to stored procedure CHECKEMP:

CREATE TRIGGER EMPRAISE
AFTER UPDATE ON EMP
REFERENCING NEW TABLE AS NEWEMPS
FOR EACH STATEMENT MODE DB2SQL
BEGIN ATOMIC
    CALL CHECKEMP(TABLE NEWEMPS);
END

Related concepts
Steps to creating and using a user-defined function
A user-defined function is similar to a host language subprogram or function. However, a user-defined function is often the better choice for an SQL application because you can invoke it in an SQL statement.

Related tasks
Accessing transition tables in a user-defined function or stored procedure
If you want to refer to the entire set of rows that a triggering SQL statement modifies, rather than to individual rows, use a transition table. You can reference a transition table in user-defined functions and procedures that are invoked from a trigger.

Creating stored procedures
A stored procedure is executable code that can be called by other programs. The process for creating one depends on the type of procedure.

Creating a user-defined function
You can extend the SQL functionality of Db2 by adding your own or third party vendor function definitions.

**Related reference**
- CALL (Db2 SQL)
- CREATE FUNCTION (Db2 SQL)
- CREATE PROCEDURE (Db2 SQL)
- select-statement (Db2 SQL)
- VALUES (Db2 SQL)

**Inserting, updating, and deleting data in views by using INSTEAD OF triggers**

INSTEAD OF triggers are triggers that execute instead of the INSERT, UPDATE, or DELETE statement that activates the trigger. You can define these triggers on views only. Use INSTEAD OF triggers to insert, update, and delete data in complex views.

**About this task**

Complex views are those views that are defined on expressions or multiple tables. In some cases, those views are read only. In these cases, INSTEAD OF triggers make the insert, update and delete operations possible. If the complex view is not read only, you can request an insert, update, or delete operation. However, Db2 automatically decides how to perform that operation on the base tables that are referenced in the view. With INSTEAD OF triggers, you can define exactly how Db2 is to execute an insert, update, or delete operation on the view. You no longer leave the decision to Db2.

**Procedure**

To insert, update, or delete data in a view by using INSTEAD OF triggers:

1. Define one or more INSTEAD OF triggers on the view by using a CREATE TRIGGER statement.
   
   You can create one trigger for each of the following operations: INSERT, UPDATE, and DELETE. These triggers define the action that Db2 is to take for each of these operations.

2. Submit a INSERT, UPDATE, or DELETE statement on the view.
   
   Db2 executes the appropriate INSTEAD OF trigger.

**Example**

Suppose that you create the following view on the sample tables DSN8C10.EMP and DSN8C10.DEPT:

```sql
CREATE VIEW EMPV (EMPNO, FIRSTNME, MIDINIT, LASTNAME, PHONENO, HIREDATE, DEPTNAME)
AS SELECT EMPNO, FIRSTNME, MIDINIT, LASTNAME, PHONENO, HIREDATE, DEPTNAME
FROM DSN8C10.EMP, DSN8C10.DEPT WHERE DSN8C10.EMP.WORKDEPT = DSN8C10.DEPT.DEPTNO
```

Suppose that you also define the following three INSTEAD OF triggers:

```sql
CREATE TRIGGER EMPV_INSERT INSTEAD OF INSERT ON EMPV
REFERENCING NEW AS NEWEMP
FOR EACH ROW MODE DB2SQL
INSERT INTO DSN8C10.EMP (EMPNO, FIRSTNME, MIDINIT, LASTNAME, WORKDEPT, PHONENO, HIREDATE)
VALUES(NEWEMP.EMPNO, NEWEMP.FIRSTNME, NEWEMP.MIDINIT, NEWEMP.LASTNAME, 
COALESCE((SELECT D.DEPTNO FROM DSN8C10.DEPT AS D
WHERE D.DEPTNAME = NEWEMP.DEPTNAME), 
RAISE_ERROR('70001', 'Unknown department name')),
NEWEMP.PHONENO, NEWEMP.HIREDATE)
```

```sql
CREATE TRIGGER EMPV_UPDATE INSTEAD OF UPDATE ON EMPV
REFERENCING NEW AS NEWEMP OLD AS OLDEMP
FOR EACH ROW MODE DB2SQL
BEGIN ATOMIC
UPDATE DSN8C10.EMP AS E
SET (E.FIRSTNME, E.MIDINIT, E.LASTNAME, E.WORKDEPT, E.PHONENO, 
E.HIREDATE) = (NEWEMP.FIRSTNME, NEWEMP.MIDINIT, NEWEMP.LASTNAME, 
```

158 Db2 12 for z/OS: Application Programming and SQL Guide
Because the view is on a query with an inner join, the view is read only. However, the INSTEAD OF triggers makes insert, update, and delete operations possible.

The following table describes what happens for various insert, update, and delete operations on the EMPV view.

<table>
<thead>
<tr>
<th>SQL statement</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>INSERT INTO EMPV VALUES (...)</td>
<td>The EMPV_INSERT trigger is activated. This trigger inserts the row into the base table DSN8C10.EMP if the department name matches a value in the WORKDEPT column in the DSN8C10.DEPT table. Otherwise, an error is returned. If a query had been used instead of a VALUES clause on the INSERT statement, the trigger body would be processed for each row from the query.</td>
</tr>
<tr>
<td>UPDATE EMPV</td>
<td>The EMPV_UPDATE trigger is activated. This trigger updates the DEPTNAME column in the DSN8C10.DEPT for the any qualifying rows.</td>
</tr>
<tr>
<td>DELETE FROM EMPV</td>
<td>The EMPV_DELETE trigger is activated. This trigger deletes the qualifying rows from the DSN8C10.EMP table.</td>
</tr>
</tbody>
</table>

**Related reference**

CREATE TRIGGER (basic) (Db2 SQL)

**Errors encountered in a trigger**

An SQL statement in a trigger body may fail during trigger execution, causing an error to occur.

Assuming that no handlers are defined in the trigger, if any SQL statement in the trigger body fails during trigger execution, Db2 rolls back all changes that are made by the triggering SQL statement and the triggered SQL statements. However, if the trigger body executes actions that are outside of the control of Db2, or are not under the same commit coordination as the Db2 subsystem in which the trigger executes, Db2 cannot undo those actions. Examples of external actions that are not under the control of Db2 are:

- Performing updates that are not under RRS commit control
- Sending an electronic mail message

If the trigger executes external actions that are under the same commit coordination as the Db2 subsystem under which the trigger executes, and an error occurs during trigger execution, Db2 places the application process that issued the triggering statement in a must-rollback state. The application must then execute a rollback operation to roll back those external actions. Examples of external actions that are under the same commit coordination as the triggering SQL operation are:

- Executing a distributed update operation
• From a user-defined function or stored procedure, executing an external action that affects an external resource manager that is under RRS commit control.

**Trigger packages**

A *trigger package* is a special type of package that is created only when you execute a CREATE TRIGGER statement. A trigger package executes only when the associated trigger is activated.

As with any other package, Db2 marks a trigger package invalid when you drop a table, index, or view on which the trigger package depends. Db2 executes an automatic rebind the next time the trigger is activated. However, if the automatic rebind fails, Db2 does not mark the trigger package as inoperative.

Unlike other packages, a trigger package is freed if you drop the table on which the trigger is defined, so you can re-create the trigger package only by recreating the table and the trigger.

Db2 supports basic and advanced triggers. You use a different REBIND subcommand for each type.

• For basic trigger packages, use the **REBIND TRIGGER PACKAGE** subcommand. You can also use **REBIND TRIGGER PACKAGE** to change a limited subset of the default bind options that Db2 used when creating the package. You can identify basic triggers by querying the SYSIBM.SYSTRIGGERS catalog table. Blank values in the SQLPL column identify basic triggers.

• For advanced trigger packages, use the **REBIND PACKAGE** subcommand. You can use the **ALTER TRIGGER** statement to change the option values with which Db2 originally bound the trigger package. You can identify advanced triggers by querying the SYSIBM.SYSTRIGGERS catalog table. 'Y' values in the SQLPL column identify advanced triggers.

Related reference

- [REBIND TRIGGER PACKAGE (DSN) (Db2 Commands)](https://www.ibm.com/support/docview.wss?uid=ssg1S0001666)
- [REBIND PACKAGE (DSN) (Db2 Commands)](https://www.ibm.com/support/docview.wss?uid=ssg1S0001666)

**Trigger cascading**

When a trigger performs an SQL operation, it might modify the subject table or other tables with triggers, therefore Db2 also activates those triggers. This situation is called trigger cascading.

A trigger that is activated as the result of another trigger can be activated at the same level as the original trigger or at a different level. Two triggers, A and B, are activated at different levels if trigger B is activated after trigger A is activated and completes before trigger A completes. If trigger B is activated after trigger A is activated and completes after trigger A completes, then the triggers are at the same level.

For example, in these cases, trigger A and trigger B are activated at the same level:

• Table X has two triggers that are defined on it, A and B. A is a before trigger and B is an after trigger. An update to table X causes both trigger A and trigger B to activate.

• Trigger A updates table X, which has a referential constraint with table Y, which has trigger B defined on it. The referential constraint causes table Y to be updated, which activates trigger B.

In these cases, trigger A and trigger B are activated at different levels:

• Trigger A is defined on table X, and trigger B is defined on table Y. Trigger B is an update trigger. An update to table X activates trigger A, which contains an UPDATE statement on table Y in its trigger body. This UPDATE statement activates trigger B.

• Trigger A calls a stored procedure. The stored procedure contains an INSERT statement for table X, which has insert trigger B defined on it. When the INSERT statement on table X executes, trigger B is activated.

When triggers are activated at different levels, it is called **trigger cascading**. Trigger cascading can occur only for after triggers because Db2 does not support cascading of before triggers.

To prevent the possibility of endless trigger cascading, Db2 supports only 16 levels of cascading of triggers, stored procedures, and user-defined functions. If a trigger, user-defined function, or stored procedure at the 17th level is activated, Db2 returns SQLCODE -724 and backs out all SQL changes in the
16 levels of cascading. However, as with any other SQL error that occurs during trigger execution, if any action occurs that is outside the control of Db2, that action is not backed out.

You can write a monitor program that issues IFI READS requests to collect Db2 trace information about the levels of cascading of triggers, user-defined functions, and stored procedures in your programs.

**Related tasks**
*Invoking IFI from a monitor program (Db2 Performance)*

### Activation order of multiple triggers

You can create multiple triggers for the same subject table, event, and activation time. The order in which those triggers are activated is the order in which the triggers were created.

Db2 records the timestamp when each CREATE TRIGGER statement executes. When an event occurs in a table that activates more than one trigger, Db2 uses the stored timestamps to determine which trigger to activate first.

Db2 always activates all before triggers that are defined on a table before the after triggers that are defined on that table, but within the set of before triggers, the activation order is by timestamp, and within the set of after triggers, the activation order is by timestamp.

In this example, triggers NEWHIRE1 and NEWHIRE2 have the same triggering event (INSERT), the same subject table (EMP), and the same activation time (AFTER). Suppose that the CREATE TRIGGER statement for NEWHIRE1 is run before the CREATE TRIGGER statement for NEWHIRE2:

```sql
CREATE TRIGGER NEWHIRE1
AFTER INSERT ON EMP
FOR EACH ROW MODE DB2SQL
BEGIN ATOMIC
  UPDATE COMPANY_STATS SET NBEMP = NBEMP + 1;
END

CREATE TRIGGER NEWHIRE2
AFTER INSERT ON EMP
REFERENCING NEW AS N_EMP
FOR EACH ROW MODE DB2SQL
BEGIN ATOMIC
  UPDATE DEPTS SET NBEMP = NBEMP + 1
  WHERE DEPT_ID = N_EMP.DEPT_ID;
END
```

When an insert operation occurs on table EMP, Db2 activates NEWHIRE1 first because NEWHIRE1 was created first. Now suppose that someone drops and re-creates NEWHIRE1. NEWHIRE1 now has a later timestamp than NEWHIRE2, so the next time an insert operation occurs on EMP, NEWHIRE2 is activated before NEWHIRE1.

If two row triggers are defined for the same action, the trigger that was created earlier is activated first for all affected rows. Then the second trigger is activated for all affected rows. In the previous example, suppose that an INSERT statement with a fullselect inserts 10 rows into table EMP. NEWHIRE1 is activated for all 10 rows, then NEWHIRE2 is activated for all 10 rows.

**Related reference**
*CREATE TRIGGER (advanced) (Db2 SQL)*
*CREATE TRIGGER (basic) (Db2 SQL)*

### Interactions between triggers and referential constraints

When you create triggers, you need to understand the interactions among the triggers and constraints on your tables. You also need to understand the effect that the order of processing of those constraints and triggers can have on the results.

In general, the following steps occur when triggering SQL statement S1 performs an insert, update, or delete operation on table T1:

1. Db2 determines the rows of T1 to modify. Call that set of rows M1. The contents of M1 depend on the SQL operation:
• For a delete operation, all rows that satisfy the search condition of the statement for a searched delete operation, or the current row for a positioned delete operation

• For an insert operation, the row identified by the VALUES statement, or the rows identified by the result table of a SELECT clause within the INSERT statement

• For an update operation, all rows that satisfy the search condition of the statement for a searched update operation, or the current row for a positioned update operation

2. Db2 processes all before triggers that are defined on T1, in order of creation.

Each before trigger executes the triggered action once for each row in M1. If M1 is empty, the triggered action does not execute.

If an error occurs when the triggered action executes, Db2 rolls back all changes that are made by S1.

3. Db2 makes the changes that are specified in statement S1 to table T1, unless an INSTEAD OF trigger is defined for that action. If an appropriate INSTEAD OF trigger is defined, Db2 executes the trigger instead of the statement and skips the remaining steps in this list.

If an error occurs, Db2 rolls back all changes that are made by S1.

4. If M1 is not empty, Db2 applies all the following constraints and checks that are defined on table T1:

• Referential constraints
• Check constraints
• Checks that are due to updates of the table through views defined WITH CHECK OPTION

Application of referential constraints with rules of DELETE CASCADE or DELETE SET NULL are activated before delete triggers or before update triggers on the dependent tables.

If any constraint is violated, Db2 rolls back all changes that are made by constraint actions or by statement S1.

5. Db2 processes all after triggers that are defined on T1, and all after triggers on tables that are modified as the result of referential constraint actions, in order of creation.

Each after row trigger executes the triggered action once for each row in M1. If M1 is empty, the triggered action does not execute.

Each after statement trigger executes the triggered action once for each execution of S1, even if M1 is empty.

If any triggered actions contain SQL insert, update, or delete operations, Db2 repeats steps 1 through 5 for each operation.

If an error occurs when the triggered action executes, or if a triggered action is at the 17th level of trigger cascading, Db2 rolls back all changes that are made in step 5 and all previous steps.

For example, table DEPT is a parent table of EMP, with these conditions:

• The DEPTNO column of DEPT is the primary key.
• The WORKDEPT column of EMP is the foreign key.
• The constraint is ON DELETE SET NULL.

Suppose the following trigger is defined on EMP:

```
CREATE TRIGGER EMPRAISE
AFTER UPDATE ON EMP
REFERENCING NEW TABLE AS NEWEMPS
FOR EACH STATEMENT MODE DB2SQL
BEGIN ATOMIC
  VALUES(CHECKEMP(TABLE NEWEMPS));
END
```

Also suppose that an SQL statement deletes the row with department number E21 from DEPT. Because of the constraint, Db2 finds the rows in EMP with a WORKDEPT value of E21 and sets WORKDEPT in those rows to null. This is equivalent to an update operation on EMP, which has update trigger EMPRAISE.
Therefore, because EMPRAISE is an after trigger, EMPRAISE is activated after the constraint action sets WORKDEPT values to null.

**Interactions between triggers and tables that have multilevel security with row-level granularity**

A BEFORE trigger affects the value of the transition variable that is associated with a security label column.

If a subject table has a security label column, the column in the transition table or transition variable that corresponds to the security label column in the subject table does not inherit the security label attribute. This means that the multilevel security check with row-level granularity is not enforced for the transition table or the transition variable. If you add a security label column to a subject table using the ALTER TABLE statement, the rules are the same as when you add any column to a subject table because the column in the transition table or the transition variable that corresponds to the security label column does not inherit the security label attribute.

If the ID you are using does not have write-down privilege and you execute an insert or update operation, the security label value of your ID is assigned to the security label column for the rows that you are inserting or updating.

When a BEFORE trigger is activated, the value of the transition variable that corresponds to the security label column is the security label of the ID if either of the following conditions is true:

- The user does not have write-down privilege
- The value for the security label column is not specified

If the user does not have write-down privilege, and the trigger changes the transition variable that corresponds to the security label column, the value of the security label column is changed back to the security label value of the user before the row is written to the page.

**Related concepts**

Multilevel security (Managing Security)

**Triggers that return inconsistent results**

When you create triggers and write SQL statements that activate those triggers, you need to ensure that executing those statements always produces the same results.

Two common reasons that you can get inconsistent results are:

- Positioned UPDATE or DELETE statements that use uncorrelated subqueries cause triggers to operate on a larger result table than you intended.
- Db2 does not always process rows in the same order, so triggers that propagate rows of a table can generate different result tables at different times.

The following examples demonstrate these situations.

**Example: Effect of an uncorrelated subquery on a triggered action:** Suppose that tables T1 and T2 look like this:

<table>
<thead>
<tr>
<th>Table T1</th>
<th>Table T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>A1</td>
<td>B1</td>
</tr>
<tr>
<td>==</td>
<td>==</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>

The following trigger is defined on T1:

```sql
CREATE TRIGGER TR1
AFTER UPDATE OF T1
FOR EACH ROW
MODE DB2SQL
BEGIN ATOMIC
```

Chapter 3. Db2 SQL programming 163
Now suppose that an application executes the following statements to perform a positioned update operation:

```sql
EXEC SQL BEGIN DECLARE SECTION;
  long hv1;
EXEC SQL END DECLARE SECTION;

EXEC SQL DECLARE C1 CURSOR FOR
  SELECT A1 FROM T1
  WHERE A1 IN (SELECT B1 FROM T2)
  FOR UPDATE OF A1;

EXEC SQL OPEN C1;

while(SQLCODE>=0 && SQLCODE!=100)
{
  EXEC SQL FETCH C1 INTO :hv1;
  UPDATE T1 SET A1=5 WHERE CURRENT OF C1;
}
```

When Db2 executes the FETCH statement that positions cursor C1 for the first time, Db2 evaluates the subselect, `SELECT B1 FROM T2`, to produce a result table that contains the two rows of column T2:

```
1
2
```

When Db2 executes the positioned UPDATE statement for the first time, trigger TR1 is activated. When the body of trigger TR1 executes, the row with value 2 is deleted from T2. However, because `SELECT B1 FROM T2` is evaluated only once, when the FETCH statement is executed again, Db2 finds the second row of T1, even though the second row of T2 was deleted. The FETCH statement positions the cursor to the second row of T1, and the second row of T1 is updated. The update operation causes the trigger to be activated again, which causes Db2 to attempt to delete the second row of T2, even though that row was already deleted.

To avoid processing of the second row after it should have been deleted, use a correlated subquery in the cursor declaration:

```sql
DCL C1 CURSOR FOR
  SELECT A1 FROM T1 X
  WHERE EXISTS (SELECT B1 FROM T2 WHERE X.A1 = B1)
  FOR UPDATE OF A1;
```

In this case, the subquery, `SELECT B1 FROM T2 WHERE X.A1 = B1`, is evaluated for each FETCH statement. The first time that the FETCH statement executes, it positions the cursor to the first row of T1. The positioned UPDATE operation activates the trigger, which deletes the second row of T2. Therefore, when the FETCH statement executes again, no row is selected, so no update operation or triggered action occurs.

**Example: Effect of row processing order on a triggered action**: The following example shows how the order of processing rows can change the outcome of an after row trigger.

Suppose that tables T1, T2, and T3 look like this:

```
Table T1     Table T2     Table T3
A1           B1           C1
==           ==           ==
1         (empty)      (empty)
2
```

The following trigger is defined on T1:

```sql
CREATE TRIGGER TR1
AFTER UPDATE ON T1
REFERENCING NEW AS N
FOR EACH ROW
MODE DB2SQL
BEGIN ATOMIC
```

164 📖 Db2 12 for z/OS: Application Programming and SQL Guide
Now suppose that a program executes the following UPDATE statement:

```
UPDATE T1 SET A1 = A1 + 1;
```

The contents of tables T2 and T3 after the UPDATE statement executes depend on the order in which Db2 updates the rows of T1.

If Db2 updates the first row of T1 first, after the UPDATE statement and the trigger execute for the first time, the values in the three tables are:

<table>
<thead>
<tr>
<th>Table T1</th>
<th>Table T2</th>
<th>Table T3</th>
</tr>
</thead>
<tbody>
<tr>
<td>A1</td>
<td>B1</td>
<td>C1</td>
</tr>
<tr>
<td>==</td>
<td>==</td>
<td>==</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>

After the second row of T1 is updated, the values in the three tables are:

<table>
<thead>
<tr>
<th>Table T1</th>
<th>Table T2</th>
<th>Table T3</th>
</tr>
</thead>
<tbody>
<tr>
<td>A1</td>
<td>B1</td>
<td>C1</td>
</tr>
<tr>
<td>==</td>
<td>==</td>
<td>==</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>2</td>
</tr>
</tbody>
</table>

However, if Db2 updates the second row of T1 first, after the UPDATE statement and the trigger execute for the first time, the values in the three tables are:

<table>
<thead>
<tr>
<th>Table T1</th>
<th>Table T2</th>
<th>Table T3</th>
</tr>
</thead>
<tbody>
<tr>
<td>A1</td>
<td>B1</td>
<td>C1</td>
</tr>
<tr>
<td>==</td>
<td>==</td>
<td>==</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>3</td>
</tr>
</tbody>
</table>

After the first row of T1 is updated, the values in the three tables are:

<table>
<thead>
<tr>
<th>Table T1</th>
<th>Table T2</th>
<th>Table T3</th>
</tr>
</thead>
<tbody>
<tr>
<td>A1</td>
<td>B1</td>
<td>C1</td>
</tr>
<tr>
<td>==</td>
<td>==</td>
<td>==</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>

### Converting existing triggers to support advanced capabilities

You can convert existing basic triggers to take advantage of advanced capabilities, including support for more SQL statements, including SQL PL in the trigger body, support for more variable types, and other advantages.

**Before you begin**

Advanced triggers are supported at application compatibility level V12R1M500 or higher.

You can identify basic triggers by querying the SYSIBM.SYSTRIGGERS catalog table. Blank values in the SQLPL column identify basic triggers.

**About this task**

*Advanced triggers* offer the following advantages over basic triggers:

- In the trigger definition, an advanced trigger can:
  - Include more types of SQL statements, including SQL PL control statements, dynamic SQL statements, and SQL comments.
– Define and reference more types of variables, including SQL variables and global variables.
– Explicitly specify bind options.
– Define multiple versions of the trigger.
  • All transition variables are nullable.
  • ALTER TRIGGER statements can change options, and change or regenerate the trigger body.
  • The OR REPLACE clause can be used in CREATE TRIGGER (advanced) statements. It enables the use of a single CREATE statement to either define a new trigger or trigger version, or update an existing trigger or trigger version if it already exists.

For more information about the differences between basic and advanced triggers, see Triggers (Introduction to Db2 for z/OS).

Procedure
To change an existing basic trigger into an advanced trigger, complete the following steps:
1. Modify the original CREATE TRIGGER statement into a CREATE TRIGGER (advanced) statement by removing any of the following items:
  • The MODE DB2SQL clause. Db2 attempts to creates a basic trigger if that clause is included.
  • Stand-alone fullselect or VALUES statements. You can use SELECT INTO statement or VALUES INTO statements instead.
2. Use one of the following approaches to convert to the new advanced trigger definition:
  • Issue the modified CREATE TRIGGER (advanced) statement with the OR REPLACE clause.
  • Issue a DROP statement for the original trigger and then issue the new CREATE TRIGGER statement.

The existing trigger is effectively dropped, and an advanced trigger is defined. If multiple triggers are defined on the associated table, the trigger activation order changes.

What to do next
If multiple triggers are defined on the associated table, you might need to restore the original the activation order of the triggers. To do that, you must drop and re-create any triggers that were created after the converted trigger was originally created, in the same order that they were originally created. For more information about the activation order of multiple triggers, see “Activation order of multiple triggers” on page 161.

Related reference
DROP (Db2 SQL)
CREATE TRIGGER (advanced) (Db2 SQL)
SYSTRIGGERS catalog table (Db2 SQL)

Sequence objects
A sequence is a user-defined object that generates a sequence of numeric values according to the specification with which the sequence was created. Sequences, unlike identity columns, are not associated with tables. Applications refer to a sequence object to get its current or next value.

The sequence of numeric values is generated in a monotonically ascending or descending order. The relationship between sequences and tables is controlled by the application, not by Db2.

Your application can reference a sequence object and coordinate the value as keys across multiple rows and tables. However, a table column that gets its values from a sequence object does not necessarily have unique values in that column. Even if the sequence object has been defined with the NO CYCLE clause, some other application might insert values into that table column other than values you obtain by referencing that sequence object.
Db2 always generates sequence numbers in order of request. However, in a data sharing group where the sequence values are cached by multiple Db2 members simultaneously, the sequence value assignments might not be in numeric order. Additionally, you might have gaps in sequence number values for the following reasons:

- If Db2 terminates abnormally before it assigns all the cached values
- If your application rolls back a transaction that increments the sequence
- If the statement containing NEXT VALUE fails after it increments the sequence

You create a sequence object with the CREATE SEQUENCE statement, alter it with the ALTER SEQUENCE statement, and drop it with the DROP SEQUENCE statement. You grant access to a sequence with the GRANT (privilege) ON SEQUENCE statement, and revoke access to the sequence with the REVOKE (privilege) ON SEQUENCE statement.

The values that Db2 generates for a sequence depend on how the sequence is created. The START WITH option determines the first value that Db2 generates. The values advance by the INCREMENT BY value in ascending or descending order.

The MINVALUE and MAXVALUE options determine the minimum and maximum values that Db2 generates. The CYCLE or NO CYCLE option determines whether Db2 wraps the generated values when it reaches the maximum value for an ascending sequence or the minimum value in a descending sequence.

**Keys across multiple tables:** You can use the same sequence number as a key value in two separate tables by first generating the sequence value with a NEXT VALUE expression to insert the first row in the first table. You can then reference this same sequence value with a PREVIOUS VALUE expression to insert the other rows in the second table.

**Example:** Suppose that an ORDERS table and an ORDER_ITEMS table are defined in the following way:

```sql
CREATE TABLE ORDERS
(ORDERNO INTEGER NOT NULL,
 ORDER_DATE DATE DEFAULT,
 CUSTNO SMALLINT
 PRIMARY KEY (ORDERNO));

CREATE TABLE ORDER_ITEMS
(ORDERNO INTEGER NOT NULL,
 PARTNO INTEGER NOT NULL,
 QUANTITY SMALLINT NOT NULL,
 PRIMARY KEY (ORDERNO,PARTNO),
 CONSTRAINT REF_ORDERNO FOREIGN KEY (ORDERNO)
 REFERENCES ORDERS (ORDERNO) ON DELETE CASCADE);
```

You create a sequence named ORDER_SEQ to use as key values for both the ORDERS and ORDER_ITEMS tables:

```sql
CREATE SEQUENCE ORDER_SEQ AS INTEGER
 START WITH 1
 INCREMENT BY 1
 NO MAXVALUE
 NO CYCLE
 CACHE 20;
```

You can then use the same sequence number as a primary key value for the ORDERS table and as part of the primary key value for the ORDER_ITEMS table:

```sql
INSERT INTO ORDERS (ORDERNO, CUSTNO)
VALUES (NEXT VALUE FOR ORDER_SEQ, 12345);

INSERT INTO ORDER_ITEMS (ORDERNO, PARTNO, QUANTITY)
VALUES (PREVIOUS VALUE FOR ORDER_SEQ, 987654, 2);
```

The NEXT VALUE expression in the first INSERT statement generates a sequence number value for the sequence object ORDER_SEQ. The PREVIOUS VALUE expression in the second INSERT statement retrieves that same value because it was the sequence number most recently generated for that sequence object within the current application process.
Db2 object relational extensions

With the object extensions of Db2, you can incorporate object-oriented concepts and methodologies into your relational database by extending Db2 with richer sets of data types and functions.

With those extensions, you can store instances of object-oriented data types in columns of tables and operate on them using functions in SQL statements. In addition, you can control the types of operations that users can perform on those data types.

The object extensions that Db2 provides are:

- Large objects (LOBs)
  The VARCHAR, VARGRAPHIC, and VARBINARY data types have a storage limit of 32 KB. Although this might be sufficient for small- to medium-size text data, applications often need to store large text documents. They might also need to store a wide variety of additional data types such as audio, video, drawings, mixed text and graphics, and images. Db2 provides three data types to store these data objects as strings of up to 2 GB - 1 in size. The three data types are binary large objects (BLOBs), character large objects (CLOBs), and double-byte character large objects (DBCLOBs).

  For a detailed discussion of LOBs, see “Storing LOB data in a table” on page 122 and Large objects (LOBs) (Db2 SQL).

- Distinct types
  A distinct type is a user-defined data type that shares its internal representation with a built-in data type but is considered to be a separate and incompatible type for semantic purposes. For example, you might want to define a picture type or an audio type, both of which have quite different semantics, but which use the built-in data type BLOB for their internal representation.

  For a detailed discussion of distinct types, see “Distinct types” on page 169.

- User-defined functions
  The built-in functions that are supplied with Db2 are a useful set of functions, but they might not satisfy all of your requirements. For those cases, you can use user-defined functions. For example, a built-in function might perform a calculation you need, but the function does not accept the distinct types you want to pass to it. You can then define a function based on a built-in function, called a sourced user-defined function, that accepts your distinct types. You might need to perform another calculation in your SQL statements for which no built-in function exists. In that situation, you can define and write an SQL function or an external function.

  For a detailed discussion of user-defined functions, see “Steps to creating and using a user-defined function” on page 183.

Creating a distinct type

Distinct types are useful when you want Db2 to handle certain data differently than other data of the same data type. For example, even though all currencies can be declared as type DECIMAL, you do not want euros to be compared to Japanese yen.

Procedure

Issue the CREATE DISTINCT TYPE statement.
For example, you can create distinct types for euros and yen by issuing the following SQL statements:

```
CREATE DISTINCT TYPE EURO AS DECIMAL(9,2);
CREATE DISTINCT TYPE JAPANESE_YEN AS DECIMAL(9,2);
```

Related reference

CREATE TYPE (distinct) (Db2 SQL)
Distinct types

A distinct type is a user-defined data type that shares its internal representation with a built-in data type (its source type), but is considered to be a separate and incompatible data type for most operations.

Each distinct type has the same internal representation as a built-in data type.

Suppose you want to define some audio and video data in a Db2 table. You can define columns for both types of data as BLOB, but you might want to use a data type that more specifically describes the data. To do that, define distinct types. You can then use those types when you define columns in a table or manipulate the data in those columns. For example, you can define distinct types for the audio and video data like this:

```sql
CREATE DISTINCT TYPE AUDIO AS BLOB (1M);
CREATE DISTINCT TYPE VIDEO AS BLOB (1M);
```

Then, your CREATE TABLE statement might look like this:

```sql
CREATE TABLE VIDEO_CATALOG;
(VIDEO_NUMBER CHAR(6) NOT NULL,
 VIDEO_SOUND AUDIO,
 VIDEO_PICS VIDEO,
 ROW_ID ROWID NOT NULL GENERATED ALWAYS);
```

For more information on LOB data, see “Storing LOB data in a table” on page 122 and Large objects (LOBs) (Db2 SQL).

After you define distinct types and columns of those types, you can use those data types in the same way you use built-in types. You can use the data types in assignments, comparisons, function invocations, and stored procedure calls. However, when you assign one column value to another or compare two column values, those values must be of the same distinct type. For example, you must assign a column value of type VIDEO to a column of type VIDEO, and you can compare a column value of type AUDIO only to a column of type AUDIO. When you assign a host variable value to a column with a distinct type, you can use any host data type that is compatible with the source data type of the distinct type. For example, to receive an AUDIO or VIDEO value, you can define a host variable like this:

```sql
SQL TYPE IS BLOB (1M) HVAV;
```

When you use a distinct type as an argument to a function, a version of that function that accepts that distinct type must exist. For example, if function SIZE takes a BLOB type as input, you cannot automatically use a value of type AUDIO as input. However, you can create a sourced user-defined function that takes the AUDIO type as input. For example:

```sql
CREATE FUNCTION SIZE(AUDIO)
RETURNS INTEGER
SOURCE SIZE(BLOB(1M));
```

Using distinct types in application programs: The main reason to use distinct types is because Db2 enforces strong typing for distinct types. Strong typing ensures that only functions, procedures, comparisons, and assignments that are defined for a data type can be used.

For example, if you have defined a user-defined function to convert U.S. dollars to euro currency, you do not want anyone to use this same user-defined function to convert Japanese yen to euros because the U.S. dollars to euros function returns the wrong amount. Suppose you define three distinct types:

```sql
CREATE DISTINCT TYPE US_DOLLAR AS DECIMAL(9,2);
CREATE DISTINCT TYPE EURO AS DECIMAL(9,2);
CREATE DISTINCT TYPE JAPANESE_YEN AS DECIMAL(9,2);
```

If a conversion function is defined that takes an input parameter of type US_DOLLAR as input, Db2 returns an error if you try to execute the function with an input parameter of type JAPANESE_YEN.
Example of distinct types, user-defined functions, and LOBs

You can create and use a distinct type based on a LOB data type.

The example in this topic demonstrates the following concepts:

• Creating a distinct type based on a LOB data type
• Defining a user-defined function with a distinct type as an argument
• Creating a table with a distinct type column that is based on a LOB type
• Defining a LOB table space, auxiliary table, and auxiliary index
• Inserting data from a host variable into a distinct type column based on a LOB column
• Executing a query that contains a user-defined function invocation
• Casting a LOB locator to the input data type of a user-defined function

Suppose that you keep electronic mail documents that are sent to your company in a Db2 table. The Db2 data type of an electronic mail document is a CLOB, but you define it as a distinct type so that you can control the types of operations that are performed on the electronic mail. The distinct type is defined like this:

```
CREATE DISTINCT TYPE E_MAIL AS CLOB(5M);
```

You have also defined and written user-defined functions to search for and return the following information about an electronic mail document:

• Subject
• Sender
• Date sent
• Message content
• Indicator of whether the document contains a user-specified string

The user-defined function definitions look like this:

```
CREATE FUNCTION SUBJECT(E_MAIL) 
RETURNS VARCHAR(200) 
EXTERNAL NAME 'SUBJECT' 
LANGUAGE C 
PARAMETER STYLE SQL 
NO SQL 
DETERMINISTIC 
NO EXTERNAL ACTION;
```

```
CREATE FUNCTION SENDER(E_MAIL) 
RETURNS VARCHAR(200) 
EXTERNAL NAME 'SENDER' 
LANGUAGE C 
PARAMETER STYLE SQL 
NO SQL 
DETERMINISTIC 
NO EXTERNAL ACTION;
```

```
CREATE FUNCTION SENDING_DATE(E_MAIL) 
RETURNS DATE 
EXTERNAL NAME 'SENDDATE' 
LANGUAGE C 
PARAMETER STYLE SQL 
NO SQL 
DETERMINISTIC 
NO EXTERNAL ACTION;
```

```
CREATE FUNCTION CONTENTS(E_MAIL) 
RETURNS CLOB(1M) 
EXTERNAL NAME 'CONTENTS' 
LANGUAGE C 
PARAMETER STYLE SQL 
NO SQL
```
The table that contains the electronic mail documents is defined like this:

```sql
CREATE TABLE DOCUMENTS
(LAST_UPDATE_TIME  TIMESTAMP,
 DOC_ROWID      ROWID NOT NULL GENERATED ALWAYS,
 A_DOCUMENT        E_MAIL);
```

Because the table contains a column with a source data type of CLOB, the table requires an associated LOB table space, auxiliary table, and index on the auxiliary table. Use statements like this to define the LOB table space, the auxiliary table, and the index:

```sql
CREATE LOB TABLESPACE DOCTSLOB
LOG YES
GBPCACHE SYSTEM;
CREATE AUX TABLE DOCAUX_TABLE
IN DOCTSLOB
STORES DOCUMENTS COLUMN A_DOCUMENT;
CREATE INDEX A_IX_DOC ON DOCAUX_TABLE;
```

To populate the document table, you write code that executes an INSERT statement to put the first part of a document in the table, and then executes multiple UPDATE statements to concatenate the remaining parts of the document. For example:

```sql
EXEC SQL BEGIN DECLARE SECTION;
    char hv_current_time[26];
    SQL TYPE IS CLOB (1M) hv_doc;
EXEC SQL END DECLARE SECTION;
/* Determine the current time and put this value */
/* into host variable hv_current_time.            */
/* Read up to 1 MB of document data from a file  */
/* into host variable hv_doc.                    */
⋮
/* Insert the time value and the first 1 MB of   */
/* document data into the table.                 */
EXEC SQL INSERT INTO DOCUMENTS
VALUES(:hv_current_time, DEFAULT, E_MAIL(:hv_doc));
/* Although there is more document data in the   */
/* file, read up to 1 MB more of data, and then */
/* use an UPDATE statement like this one to      */
/* concatenate the data in the host variable     */
/* to the existing data in the table.            */
EXEC SQL UPDATE DOCUMENTS
SET A_DOCUMENT = A_DOCUMENT || E_MAIL(:hv_doc)
WHERE LAST_UPDATE_TIME = :hv_current_time;
```

Now that the data is in the table, you can execute queries to learn more about the documents. For example, you can execute this query to determine which documents contain the word "performance":

```sql
SELECT SENDER(A_DOCUMENT), SENDING_DATE(A_DOCUMENT),
SUBJECT(A_DOCUMENT)
FROM DOCUMENTS
WHERE CONTAINS(A_DOCUMENT,'performance') = 1;
```

Because the electronic mail documents can be very large, you might want to use LOB locators to manipulate the document data instead of fetching all of a document into a host variable. You can use a LOB locator on any distinct type that is defined on one of the LOB types. The following example shows
how you can cast a LOB locator as a distinct type, and then use the result in a user-defined function that takes a distinct type as an argument:

```sql
EXEC SQL BEGIN DECLARE SECTION
    long hv_len;
    char hv_subject[200];
    SQL TYPE IS CLOB_LOCATOR hv_email_locator;
EXEC SQL END DECLARE SECTION

/* Select a document into a CLOB locator. */
EXEC SQL SELECT A_DOCUMENT, SUBJECT(A_DOCUMENT)
    INTO :hv_email_locator, :hv_subject
    FROM DOCUMENTS
    WHERE LAST_UPDATE_TIME = :hv_current_time;

/* Extract the subject from the document. The */
/* SUBJECT function takes an argument of type */
/* E_MAIL, so cast the CLOB locator as E_MAIL. */
EXEC SQL SET :hv_subject = SUBJECT(CAST(:hv_email_locator AS E_MAIL));
```

### Arrays in SQL statements

An array is an ordered set of elements of a single built-in data type. An array can have an associated user-defined array type, or it can be the result of an SQL operation that returns an array value without an associated user-defined array type.

Arrays can be **ordinary arrays** and **associative arrays**.

Ordinary arrays have a user-defined upper bound. Elements in the array can be accessed and modified by their index value. Array elements are referenced in SQL statements by using one-based indexing; for example, `MYARRAY[1]`, `MYARRAY[2]`, and so on.

Associative arrays have no upper bound. Associative arrays contain an ordered set of zero or more elements, where each element in the array is ordered by and can be referenced by an associated index value. The data type of the index values can be an integer or a character string, but all index values for the array have the same data type.

Arrays can be used only in the following contexts:

- Parameters to SQL functions
- RETURN data types from SQL functions
- Parameters to SQL procedures
- SQL variables that are declared in SQL functions
- SQL variables that are declared in SQL procedures

You can create an array by creating an array type, and then defining an array variable of that type. For example:

```sql
-- CREATE ORDINARY ARRAY TYPE INTARRAY
CREATE TYPE INTARRAY AS INTEGER ARRAY[100];
-- IN AN SQL PROCEDURE, DEFINE ARRAY INTA OF THE INTARRAY TYPE
DECLARE INTA INTARRAY;
-- CREATE ASSOCIATIVE ARRAY TYPE CHARARRAY
CREATE TYPE CHARARRAY AS CHAR(10) ARRAY[VARCHAR(10)];
-- IN AN SQL PROCEDURE, DEFINE ARRAY CHARA OF THE CHARARRAY TYPE
DECLARE CHARA CHARARRAY;
```

You cannot retrieve the contents of a column directly into an array. You need to use the `ARRAY_AGG` function to create an array that is the intermediate result of a `SELECT` statement, and then retrieve the contents of that array into an SQL array variable or parameter. For example:

```sql
-- INTB IS AN OUT PARAMETER OF ORDINARY ARRAY TYPE INTARRAY.
-- COL2 IS AN INTEGER COLUMN.
-- ARRAY_AGG RETRIEVES THE VALUES FROM COL2, AND PUTS THEM INTO AN ARRAY.
SELECT ARRAY_AGG(COL2) INTO INTB FROM TABLE1;
```
You can retrieve data from an array by using the UNNEST specification to assign array elements to an intermediate result table. For example:

```sql
-- IDS AND NAMES ARE ARRAYS OF TYPE INTARRAY.
INSERT INTO PERSONS(ID, NAME)
(SELECT T.I, T.N FROM UNNEST(IDS, NAMES) AS T(I, N));
```

To populate arrays, you use array constructors.

For example, this statement populates an ordinary array:

```sql
SET CHARA = ARRAY['1','2','3','4','5','6'];
```

For example, these statements populate an associative array, which must be populated one element at a time:

```sql
SET CANADACAPITALS['Alberta'] = 'Edmonton';
SET CANADACAPITALS['Manitoba'] = 'Winnipeg';
SET CANADACAPITALS['Ontario'] = 'Toronto';
SET CANADACAPITALS['Nova Scotia'] = 'Halifax';
```

A number of built-in functions are available for manipulating arrays. They are:

**ARRAY_DELETE**
- Deletes elements from an array.

**ARRAY_FIRST**
- Returns the minimum array index value of an array.

**ARRAY_LAST**
- Returns the maximum array index value of an array.

**ARRAY_NEXT**
- Returns the next larger array index value, relative to a specified array index value.

**ARRAY_PRIOR**
- Returns the next smaller array index value, relative to a specified array index value.

**CARDINALITY**
- Returns the number of elements in an array.

**MAX_CARDINALITY**
- Returns the maximum number of elements that an array can contain.

**TRIM_ARRAY**
- Deletes elements from the end of an ordinary array.

Related concepts
- Array type comparisons (Db2 SQL)
- Array type assignments (Db2 SQL)
- Array types (Db2 SQL)

Related reference
- Array constructor (Db2 SQL)
- ARRAY_AGG (Db2 SQL)
- ARRAY_DELETE (Db2 SQL)
- ARRAY_FIRST (Db2 SQL)
- ARRAY_NEXT (Db2 SQL)
- ARRAY_PRIOR (Db2 SQL)
- CARDINALITY (Db2 SQL)
- MAX_CARDINALITY (Db2 SQL)
- TRIM_ARRAY (Db2 SQL)
Example of using arrays in an SQL procedure
An example demonstrates many of the ways that you can use arrays in a native SQL procedure.

The example demonstrates how to:

• Create an associative array type.
• Create an ordinary array type.
• Create a stored procedure with arrays as parameters.
• Define arrays as SQL variables.
• Use the ARRAY_AGG built-in function in a cursor declaration, to assign the rows of a single-column result table to elements of an array. Use the cursor to retrieve the array into an SQL out parameter.
• Use an array constructor to initialize an array.
• Assign a constant or an expression to an array element.
• Use the UNNEST specification to generate the intermediate result table from an array for a subselect within an INSERT statement.
• Use the ARRAY_AGG built-in function to assign the rows of a single column result table to elements of an array, and then assign that array to an array SQL OUT parameter.
• Use the CARDINALITY built-in function to determine how many times to execute a WHILE loop.
• Use a parameter marker for an array variable and an array index in the WHERE clause of a SELECT statement.
• Use the ARRAY_AGG built-in function in the SELECT list of a SELECT INTO statement, and assign the resulting array to an array SQL OUT parameter.
• Update column values with array elements.

In this example, the pound sign (#) is used as the SQL terminator character.

```sql
-- CREATE ASSOCIATIVE ARRAY TYPES
CREATE TYPE CHARARRAY AS CHAR(10) ARRAY[VARCHAR(3)]#
CREATE TYPE BIGINTARRAY AS BIGINT ARRAY[INTEGER]#

-- CREATE ORDINARY ARRAY TYPES
CREATE TYPE INTARRAY AS INTEGER ARRAY[100]#
CREATE TYPE STRINGARRAY AS VARCHAR(10) ARRAY[100]#

-- CREATE TABLES THAT ARE USED IN SQL PROCEDURE PROCESSPERSONS
CREATE TABLE PERSONS (ID INTEGER, NAME VARCHAR(10))#
CREATE TABLE ARRAYTEST (CHARCOL CHAR(10), INTCOL INT)#

-- SQL PROCEDURE PROCESSPERSONS HAS THREE ARRAY PARAMETERS:
-- OUTSETARRAY IS AN OUT PARAMETER OF ORDINARY ARRAY TYPE STRINGARRAY.
-- OUTSELECTWITHCURSOR IS AN OUT PARAMETER OF ORDINARY ARRAY TYPE STRINGARRAY.
-- OUTSELECTWITHARRAYAGG IS AN OUT PARAMETER OF ORDINARY ARRAY TYPE INTARRAY.
CREATE PROCEDURE PROCESSPERSONS(OUT OUTSETARRAY STRINGARRAY,
INOUT INT0 INT,
OUT OUTSELECTWITHCURSOR STRINGARRAY,
OUT OUTMAXCARDINALITY BIGINT,
OUT OUTSELECTWITHARRAYAGG INTARRAY)

ARRAYDEMO: BEGIN
-- DECLARE SQL VARIABLES OF ORDINARY ARRAY TYPES
DECLARE IDS_ORDARRAYVAR INTARRAY;
DECLARE INT_ORDARRAYVAR INTARRAY;
DECLARE NAMES_ORDARRAYVAR STRINGARRAY;
-- DECLARE SQL VARIABLES OF ASSOCIATIVE ARRAY TYPES
DECLARE CHAR_ASSOCARRAYVAR CHARARRAY;
DECLARE BIGINT_ASSOCARRAYVAR BIGINTARRAY;
-- DECLARE SCALAR SQL VARIABLES
DECLARE DECFLOAT_VAR DECFLOAT;
DECLARE BIGINT_VAR BIGINT;
DECLARE SMALLINT_VAR SMALLINT;
DECLARE INT_VAR INT DEFAULT 1;
DECLARE STMT_VAR CHAR(100);
-- DECLARE A CURSOR
DECLARE C2 CURSOR FOR S1;
```
-- THE RESULT TABLE OF CURSOR C1 IS AN ARRAY THAT IS POPULATED BY
-- RETRIEVING THE VALUES OF THE NAME COLUMN FROM TABLE PERSONS,
-- ORDERING THE VALUES BY ID, AND USING THE ARRAY_AGG FUNCTION
-- TO ASSIGN THE VALUES TO AN ARRAY.
--
DECLARE C1 CURSOR FOR SELECT ARRAY_AGG(NAME ORDER BY ID) FROM PERSONS
  WHERE NAME LIKE 'J%';
--
-- USE ARRAY CONSTRUCTORS TO INITIALIZE ARRAYS
--
SET IDS_ORDARRAYVAR = ARRAY[5,6,7];
SET NAMES_ORDARRAYVAR = ARRAY['BOB', 'ANN', 'SUE'];
SET CHAR_ASSOCARRAYVAR['001']='1';
SET CHAR_ASSOCARRAYVAR['002']='2';
SET CHAR_ASSOCARRAYVAR['003']='3';
SET CHAR_ASSOCARRAYVAR['004']='4';
SET CHAR_ASSOCARRAYVAR['005']='5';
SET CHAR_ASSOCARRAYVAR['006']='6';
SET INT_ORDARRAYVAR = ARRAY[1,INTEGER(2),3+0,4,5,6];
SET BIGINT_ASSOCARRAYVAR[1] = 9;
SET BIGINT_ASSOCARRAYVAR[3] = 10;
SET BIGINT_ASSOCARRAYVAR[5] = 11;
SET BIGINT_ASSOCARRAYVAR[7] = 12;
SET BIGINT_ASSOCARRAYVAR[9] = 13;
--
-- ASSIGN A CONSTANT TO AN ARRAY ELEMENT.
--
SET IDS_ORDARRAYVAR[4] = 8;
--
-- ASSIGN AN EXPRESSION TO AN ARRAY ELEMENT.
--
SET IDS_ORDARRAYVAR[5] = 8 * 4;
--
-- ASSIGN AN ARRAY ELEMENT TO ANOTHER ARRAY ELEMENT. USE AN EXPRESSION
-- TO IDENTIFY THE TARGET ARRAY ELEMENT.
--
SET NAMES_ORDARRAYVAR[1+INT_VAR] = NAMES_ORDARRAYVAR[5];
--
-- POPULATE THE PERSONS TABLE WITH AN INSERT STATEMENT WITH A SUBSELECT:
-- USE UNNEST TO RETRIEVE VALUES FROM AN ARRAY INTO AN INTERMEDIATE RESULT
-- TABLE.
-- INSERT THE VALUES FROM THE INTERMEDIATE RESULT TABLE INTO
-- THE PERSONS TABLE.
--
INSERT INTO PERSONS(ID, NAME)
  (SELECT T.I, T.N FROM UNNEST(IDS_ORDARRAYVAR, NAMES_ORDARRAYVAR) AS T(I, N));
--
-- USE THE ARRAY_AGG FUNCTION TO CREATE AN ARRAY FROM THE RESULT
-- TABLE OF A SELECT. THEN ASSIGN THAT ARRAY TO AN SQL OUT PARAMETER.
--
SET OUTSETARRAY = (SELECT ARRAY_AGG(NAME ORDER BY ID)
  FROM PERSONS
  WHERE NAME LIKE '%%%');
--
-- USE THE CARDINALITY FUNCTION TO CONTROL THE NUMBER OF TIMES THAT
-- AN INSERT STATEMENT IS EXECUTED TO POPULATE TABLE ARRAYTEST
-- WITH ARRAY ELEMENTS.
--
SET SMALLINT_VAR = 1;
WHILE SMALLINT_VAR <= CARDINALITY(INT_ORDARRAYVAR) DO
  INSERT INTO ARRAYTEST VALUES
    (CHAR_ASSOCARRAYVAR[SMALLINT_VAR],
    INT_ORDARRAYVAR[SMALLINT_VAR]);
  SET SMALLINT_VAR = SMALLINT_VAR+1;
END WHILE;
--
-- DYNAMICALLY EXECUTE AN SQL SELECT STATEMENT WITH A PARAMETER MARKER
-- FOR AN ARRAY, AND A PARAMETER MARKER FOR THE ARRAY INDEX.
--
SET INT_VAR = 3;
SET STMT_VAR =
  'SELECT INTCOL FROM ARRAYTEST WHERE INTCOL = ' ||
  'CAST(? AS INTARRAY)[?]';
PREPARE S1 FROM STMT_VAR;
OPEN C2 USING INT_ORDARRAYVAR, INT_VAR;
FETCH C2 INTO INT0;
CLOSE C2;
--
-- USE A CURSOR TO FETCH AN ARRAY THAT IS CREATED WITH THE ARRAY_AGG FUNCTION
-- INTO AN ARRAY SQL OUT PARAMETER.
--
OPEN C1;
FETCH C1 INTO OUTSELECTWITHCURSOR;
CLOSE C1;
-- RETURN THE MAXIMUM CARDINALITY OF AN ARRAY USING THE MAX_CARDINALITY
-- FUNCTION, AND STORE THE VALUE IN AN SQL VARIABLE.
--
SET OUTMAXCARDINALITY = MAX_CARDINALITY(INT_ORDARRAYVAR);
-- IN A SELECT INTO STATEMENT, USE THE ARRAY_AGG FUNCTION TO
-- ASSIGN THE VALUES OF COLUMN INTCOL TO ARRAY ELEMENTS, AND ASSIGN
-- THOSE ELEMENTS TO ARRAY OUT PARAMETER OUTSELECTWITHARRAYAGG.
--
SELECT ARRAY_AGG(INTCOL) INTO OUTSELECTWITHARRAYAGG FROM ARRAYTEST;
--
-- IN AN UPDATE STATEMENT, ASSIGN ARRAY ELEMENTS TO COLUMNS.
--
SET SMALLINT_VAR = 1;
WHILE SMALLINT_VAR <= CARDINALITY(INT_ORDARRAYVAR) DO
  UPDATE ARRAYTEST
  SET CHARCOL = CHAR_ASSOCARRAYVAR[SMALLINT_VAR], INTCOL = INT_ORDARRAYVAR[SMALLINT_VAR];
  SET SMALLINT_VAR = SMALLINT_VAR +1;
END WHILE;
END#
Each time that the function is invoked, the package executes one or more times. See CREATE FUNCTION (external scalar) (Db2 SQL).

**External table**

The function is written in a programming language. It returns a table to the subselect from which it was started by returning one row at a time, each time that the function is started. The external executable routine (package) is registered with a database server along with various attributes of the function. Each time that the function is invoked, the package executes one or more times. See CREATE FUNCTION (external table) (Db2 SQL).

**Sourced**

The function is implemented by invoking another function (either built-in, external, SQL, or sourced) that exists at the server. The function inherits the attributes of the underlying source function. A sourced function does not have an associated package. See CREATE FUNCTION (sourced) (Db2 SQL).

**SQL scalar**

The function is written exclusively in SQL statements and returns a scalar value. The body of an SQL scalar function is written in the SQL procedural language (SQL PL). The function is defined at the current server along with various attributes of the function.

Db2 supports two types of SQL scalar functions, inlined and compiled:

- *Inlined SQL scalar functions* contain a single RETURN statement, which returns the value of a simple expression. The function is not invoked as part of a query; instead, the expression in the RETURN statement of the function is copied (inlined) into the query itself. Therefore, a package is not generated for an inlined SQL scalar function.

- *Compiled SQL scalar functions* support a larger set of functionality, including all of the SQL PL statements. A package is generated for a compiled SQL scalar function. It contains the body of the function, including control statements. It might also contain statements generated by Db2. Each time that the function is invoked, the package executes one or more times.

When a CREATE FUNCTION statement for an SQL scalar function is processed, Db2 attempts to create an inlined SQL scalar function. If the function cannot be created as an inlined function, Db2 attempts to create a compiled SQL scalar function. For more information on the syntax and rules for these types of functions, see CREATE FUNCTION (inlined SQL scalar) (Db2 SQL) and CREATE FUNCTION (compiled SQL scalar) (Db2 SQL).

To determine what type of SQL scalar function is created, refer to the INLINE column of the SYSIBM.SYSROUTINES catalog table.

**SQL table**

The function is written exclusively as an SQL RETURN statement and returns a set of rows. The body of an SQL table function is written in the SQL procedural language. The function is defined at the current server along with various attributes. The function is not invoked as part of a query. Instead, the expression in the RETURN statement of the function is copied (inlined) into the query itself. Therefore, a package is not generated for an SQL table function. See CREATE FUNCTION (SQL table) (Db2 SQL).

The environment for user-defined functions includes application address space, from which a program invokes a user-defined function; a Db2 system, where the packages from the user-defined function are run; and a WLM-established address space, where the user-defined function may be executed; as shown in the following figure.
For information on Java user-defined functions, see Java stored procedures and user-defined functions (Db2 Application Programming for Java). For user-defined functions in other languages, see the following instructions.

**Procedure**

To create a user-defined function:

1. Write and prepare the user-defined function, as described in “Writing an external user-defined function” on page 184.
   This step is necessary only for an external user-defined function.
2. Define the user-defined function to Db2 by issuing a CREATE FUNCTION statement that specifies the type of function that you want to create.
   For more information, see CREATE FUNCTION (Db2 SQL).
3. Invoke the user-defined function from an SQL application, as described in “Invoking a user-defined function” on page 452.

**Definition for an SQL user-defined scalar function**

You can define an SQL user-defined function to calculate the tangent of a value by using the existing built-in SIN and COS functions:

```
CREATE FUNCTION TAN (X DOUBLE)
RETURNS DOUBLE
LANGUAGE SQL
CONTAINS SQL
DETERMINISTIC
RETURN SIN(X)/COS(X);
```

The logic of the function is contained in the function definition as the following statement:

```
RETURN SIN(X)/COS(X)
```
What to do next

If you discover after you define the function that you need to change a part of the definition, you can use an ALTER FUNCTION statement to change the definition. You cannot use ALTER FUNCTION to change some of the characteristics of a user-defined function definition.

Related concepts
Sample user-defined functions (Db2 SQL)

Related tasks
Controlling user-defined functions (Db2 Administration Guide)

Related reference
CREATE FUNCTION (Db2 SQL)

External functions

An external user-defined function is a function that is written in a programming language. An external function is defined to the database with a reference to an external program that contains the logic that is executed when the function is invoked.

An external user-defined function that returns a single value is a scalar function. An external user-defined function that returns a table is a table function.

You can write an external user-defined function in assembler, C, C++, COBOL, PL/I, or Java. User-defined functions that are written in COBOL can include object-oriented extensions, just as other Db2 COBOL programs can. User-defined functions that are written in Java follow coding guidelines and restrictions specific to Java. For information about writing Java user-defined functions, see Java stored procedures and user-defined functions (Db2 Application Programming for Java).

Examples

Example 1: Definition for an external user-defined scalar function
A programmer develops a user-defined function that searches for a string of maximum length 200 in a CLOB value whose maximum length is 500 KB. This CREATE FUNCTION statement defines the user-defined function:

```sql
CREATE FUNCTION FINDSTRING (CLOB(500K), VARCHAR(200))
RETURNS INTEGER
CAST FROM FLOAT
SPECIFIC FINDSTRINGCLOB
EXTERNAL NAME 'FINDSTR'
LANGUAGE C
PARAMETER STYLE SQL
NO SQL
DETERMINISTIC
NO EXTERNAL ACTION;
```

The function returns a status code as an integer. The CAST FROM clause is specified because the function operation results in a floating point value, and users are expecting an integer result for their SQL statements. The user-defined function is written in C and contains no SQL statements.

Suppose that you want a FINDSTRING user-defined function to work on BLOB data types, as well as CLOB types. You can define another instance of a FINDSTRING user-defined function that specifies a BLOB type as input:

```sql
CREATE FUNCTION FINDSTRING (BLOB(500K), VARCHAR(200))
RETURNS INTEGER
CAST FROM FLOAT
SPECIFIC FINDSTRINGBLOB
EXTERNAL NAME 'FNDBLOB'
LANGUAGE C
PARAMETER STYLE SQL
NO SQL
DETERMINISTIC;
```
Each instance of FINDSTRING uses a different application program to implement the logic for the user-defined function.

**Example 2: Definition for an external user-defined scalar function**

A programmer has written a user-defined function for division. That is, this user-defined function is invoked when an application program executes a statement using the division operator (/), such as the following statement:

```
UPDATE TABLE1 SET INTCOL1="/"(INTCOL2,INTCOL3);
```

The user-defined function takes two integer values as input. The output from the user-defined function is of type integer. The user-defined function is in the MATH schema, is written in assembler, and contains no SQL statements. This CREATE FUNCTION statement defines the user-defined function:

```
CREATE FUNCTION MATH."/" (INT, INT)
RETURNS INTEGER
SPECIFIC DIVIDE
EXTERNAL NAME 'DIVIDE'
LANGUAGE ASSEMBLE
PARAMETER STYLE SQL
NO SQL
DETERMINISTIC;
```

**Example 3: Definition for an external user-defined table function**

An application programmer develops a user-defined function that receives two input values and returns a table. The two input values are:

- A character string of maximum length 30 that describes a subject
- A character string of maximum length 255 that contains text to search for

The user-defined function scans documents on the subject for the search string and returns a list of documents that match the search criteria, with an abstract for each document. The list is in the form of a two-column table. The first column is a character column of length 16 that contains document IDs. The second column is a varying-character column of maximum length 5000 that contains document abstracts.

The user-defined function is written in COBOL, uses SQL only to perform queries, and always produces the same output for given input. The CARDINALITY option specifies that you should expect an invocation of the user-defined function to return about 20 rows.

The following CREATE FUNCTION statement defines the user-defined function:

```
CREATE FUNCTION DOCMATCH (VARCHAR(30), VARCHAR(255))
RETURNS TABLE (DOC_ID CHAR(16), DOC_ABSTRACT VARCHAR(5000))
EXTERNAL NAME 'DOCMTCH'
LANGUAGE COBOL
PARAMETER STYLE SQL
READS SQL DATA
DETERMINISTIC
CARDINALITY 20;
```

**SQL scalar functions**

An SQL scalar function is a user-defined function written in SQL and it returns a single value each time it is invoked. SQL scalar functions contain the source code for the user-defined function in the user-defined function definition. There are two kinds of SQL scalar functions, inlined and compiled.

All SQL scalar functions that were created prior to DB2 10 are inlined SQL scalar functions. Beginning with DB2 10, SQL scalar functions may be created as either inlined or compiled.

Db2 determines whether an SQL scalar function is inlined or compiled according to whether or not the CREATE FUNCTION statement that defines the function makes use of enhanced features. See CREATE FUNCTION (inlined SQL scalar) (Db2 SQL) and CREATE FUNCTION (compiled SQL scalar) (Db2 SQL) for more information.
An inlined SQL scalar function has a body with a single RETURN statement. The RETURN statement can return either a NULL value or a simple expression that does not reference a scalar fullselect. No package will be generated for an inlined SQL scalar function. During the preparation of an SQL statement that references the function (when the function is invoked), the expression specified in the RETURN statement of the function is simply inlined into that SQL statement.

A compiled SQL scalar function can have a body with logic written in SQL PL language. It can make use of any of the enhanced features for the CREATE FUNCTION statement including the support for TABLE LOCATOR data type for parameters, various options, and an enhanced RETURN statement that allows reference to a scalar fullselect. A package is created for a compiled SQL scalar function.

Compiled SQL scalar functions include support for versions and source code management. You can use compiled SQL scalar functions for the following tasks:

- Define multiple versions of an SQL scalar function, where one version is considered the "active" version.
- Activate a particular version of an SQL scalar function.
- Alter the routine options that are associated with a version of an SQL scalar function.
- Define a new version of an SQL scalar function by specifying the same function signature as the current version, and different routine options and function body.
- Replace the definition of an existing version by specifying the same function signature as the current version, and different routine options and function body.
- Drop a version of an SQL scalar function.
- Fall back to a previous version without requiring an explicit rebind or recompile, by activating the previous version.

You can deploy compiled SQL scalar functions to multiple servers to allow a wider community to use functions that have been thoroughly tested, without the risk of changing the logic in the routine body. Use the Unified Debugger to remotely debug compiled SQL scalar functions that execute on Db2 for z/OS servers.

To prepare an SQL scalar function for execution, you execute the CREATE FUNCTION statement, either statically or dynamically.

**Example: Definition for a compiled SQL scalar user-defined function**

The following example defines a scalar function that returns the text of an input string, in reverse order. The example also explains how to determine why various SQL statements are allowed in a compiled SQL scalar function.

A compiled SQL scalar CREATE FUNCTION statement contains an SQL-routine-body, as defined in CREATE FUNCTION (compiled SQL scalar) (Db2 SQL). The syntax diagram for SQL-routine-body defines the function body as a single SQL control statement. The syntax diagram for SQL-control-statement in SQL procedural language (SQL PL) (Db2 SQL) identifies the control statements that can be specified, including a RETURN statement.

An SQL function can contain multiple SQL statements if the outermost SQL statement is an SQL-control-statement that includes other SQL statements. These statements are defined as SQL procedure statements. The syntax diagram in SQL-procedure-statement (Db2 SQL) identifies the SQL statements that can be specified within a control statement. The syntax notes for SQL-procedure-statement clarify the SQL statements that are allowed in an SQL function.
The SQL function has the following keywords and statements:

- The BEGIN and END keywords *(A)* indicate the beginning and the end of a compound statement.
- The DECLARE statements *(B)* are components of a compound statement, and define SQL variables within the compound statement. For more information on compound statements, see compound-statement (Db2 SQL).
- The IF statement *(C)*, the RETURN statements *(D)*, and the WHILE statement *(F)* are SQL control statements.
- The SET assignment statements *(E)* are SQL control statements that assign values to SQL variables.

SQL variables can be referenced anywhere in the compound statement in which they are declared, including any SQL statement that is directly or indirectly nested within that compound statement. See References to SQL parameters and variables (Db2 SQL) for more information.

**Related tasks**

- Creating a user-defined function
  
  You can extend the SQL functionality of Db2 by adding your own or third party vendor function definitions.

**Related reference**

- CREATE FUNCTION (Db2 SQL)

**SQL table functions**

An SQL table function is a function that is written exclusively in SQL statements and returns a single result table.

An SQL table function can define a parameter as a distinct type, define a parameter for a transition table (for example, the TABLE LIKE ... AS LOCATOR syntax), and include a single SQL PL RETURN statement that returns a result table.

The CREATE statement for an SQL table function is an executable statement that can be dynamically prepared only if DYNAMICRULES run behavior is implicitly or explicitly specified.

The ALTER statement for an SQL table function can be embedded in an application program or issued interactively. The ALTER statement is an executable statement that can be dynamically prepared only if DYNAMICRULES run behavior is implicitly or explicitly specified.

**Sourced functions**

A sourced function is a function that invokes another function that already exists at the server. The function inherits the attributes of the underlying source function. The source function can be built-in, external, SQL, or sourced. Sourced functions can be used to extend built-in aggregate and scalar functions for use on distinct types.

You can use sourced functions to build upon existing built-in functions or other user-defined functions. Sourced functions are useful to extend built-in aggregate and scalar functions for use on distinct types.

To implement a sourced function, issue a CREATE FUNCTION statement and identify the function upon which you want to base the sourced function in the SOURCE clause.

**Example: Definition of a sourced user-defined function**

Suppose you need a user-defined function that finds a string in a value with a distinct type of BOAT. BOAT is a distinct type based on a BLOB data type. User-defined function FINDSTRINGBLOB has already been defined to take a BLOB data type as input and perform the required function, but it cannot be invoked with a value of the BOAT data type. The specific name for FINDSTRING is FINDSTRINGBLOB.
You can define a sourced user-defined function based on FINDSTRING to do the string search on values of type BOAT. Db2 implicitly casts the BOAT argument to a BLOB when the source function, FINDSTRING that accepts a BLOB value, is invoked. This CREATE FUNCTION statement defines the sourced user-defined function:

```
CREATE FUNCTION FINDSTRING (BOAT, VARCHAR(200))
RETURNS INTEGER
SPECIFIC FINDSTRINGBOAT
SOURCE SPECIFIC FINDSTRINGBLOB;
```

**Related reference**
CREATE FUNCTION (sourced) (Db2 SQL)

**Steps to creating and using a user-defined function**

A user-defined function is similar to a host language subprogram or function. However, a user-defined function is often the better choice for an SQL application because you can invoke it in an SQL statement.

This section contains information that applies to all user-defined functions and specific information about user-defined functions in languages other than Java.

Creating and using a user-defined function involves these steps:

- Setting up the environment for user-defined functions
  A systems administrator probably performs this step. The user-defined function environment is shown in the following figure.

**Figure 7. The user-defined function environment**

It contains an application address space, from which a program invokes a user-defined function; a Db2 system, where the packages from the user-defined function are run; and a WLM-established address space, where the user-defined function is executed. The steps for setting up and maintaining the user-defined function environment are the same as for setting up and maintaining the environment for stored procedures in WLM-established address spaces.

- Writing and preparing the user-defined function
  This step is necessary only for an external user-defined function.
  The person who performs this step is called the user-defined function implementer.

- Defining the user-defined function to Db2
The person who performs this step is called the user-defined function **definer**.

- Invoking the user-defined function from an SQL application

The person who performs this step is called the user-defined function **invoker**.

**Related concepts**
Java stored procedures and user-defined functions (Db2 Application Programming for Java)

**Writing an external user-defined function**

An **external user-defined function** is written in a programming language and is similar to other SQL programs. You can include static or dynamic SQL statements, IFI calls, and Db2 commands that are issued through IFI calls.

**Procedure**

You can write an external user-defined function in assembler, C, C++, COBOL, PL/I, or Java.

User-defined functions that are written in COBOL can include object-oriented extensions, just as other Db2 COBOL programs can. User-defined functions that are written in Java follow coding guidelines and restrictions specific to Java.

Your user-defined function can also access remote data by using DRDA access using CONNECT or SET CONNECTION statements.

**Restrictions on user-defined function programs**

Observe these restrictions when you write a user-defined function:

- Because Db2 uses the Resource Recovery Services attachment facility (RRSAF) as its interface with your user-defined function, you must not include RRSAF calls in your user-defined function. Db2 rejects any RRSAF calls that it finds in a user-defined function.

- If your user-defined function is not defined with parameters SCRATCHPAD or EXTERNAL ACTION, the user-defined function is not guaranteed to execute under the same task each time it is invoked.

- You cannot execute COMMIT or ROLLBACK statements in your user-defined function.

- You must close all cursors that were opened within a user-defined scalar function. Db2 returns an SQL error if a user-defined scalar function does not close all cursors that it opened before it completes.

- When you choose the language in which to write a user-defined function program, be aware of restrictions on the number of parameters that can be passed to a routine in that language. User-defined table functions in particular can require large numbers of parameters. Consult the programming guide for the language in which you plan to write the user-defined function for information about the number of parameters that can be passed.

- You cannot pass LOB file reference variables as parameters to user-defined functions.

- User-defined functions cannot return LOB file reference variables.

- You cannot pass parameters with the type XML to user-defined functions. You can specify tables or views that contain XML columns as table locator parameters. However, you cannot reference the XML columns in the body of the user-defined function.

**Coding your user-defined function as a main program or as a subprogram**

You can code your user-defined function as either a main program or a subprogram. The way that you code your program must agree with the way you defined the user-defined function: with the PROGRAM TYPE MAIN or PROGRAM TYPE SUB parameter. The main difference is that when a main program starts, Language Environment allocates the application program storage that the external user-defined function uses. When a main program ends, Language Environment closes files and releases dynamically allocated storage.

If you code your user-defined function as a subprogram and manage the storage and files yourself, you can get better performance. The user-defined function should always free any allocated storage before it exits. To keep data between invocations of the user-defined function, use a scratchpad.
You must code a user-defined table function that accesses external resources as a subprogram. Also ensure that the definer specifies the EXTERNAL ACTION parameter in the CREATE FUNCTION or ALTER FUNCTION statement. Program variables for a subprogram persist between invocations of the user-defined function, and use of the EXTERNAL ACTION parameter ensures that the user-defined function stays in the same address space from one invocation to another.

**Parallelism considerations**

If the definer specifies the parameter ALLOW PARALLEL in the definition of a user-defined scalar function, and the invoking SQL statement runs in parallel, the function can run under a parallel task. Db2 executes a separate instance of the user-defined function for each parallel task. When you write your function program, you need to understand how the following parameter values interact with ALLOW PARALLEL so that you can avoid unexpected results:

- **SCRATCHPAD**
  
  When an SQL statement invokes a user-defined function that is defined with the ALLOW PARALLEL parameter, Db2 allocates one scratchpad for each parallel task of each reference to the function. This can lead to unpredictable or incorrect results.
  
  For example, suppose that the user-defined function uses the scratchpad to count the number of times it is invoked. If a scratchpad is allocated for each parallel task, this count is the number of invocations done by the parallel task and not for the entire SQL statement, which is not the result that is wanted.

- **FINAL CALL**
  
  If a user-defined function performs an external action, such as sending a note, for each final call to the function, one note is sent for each parallel task instead of once for the function invocation.

- **EXTERNAL ACTION**
  
  Some user-defined functions with external actions can receive incorrect results if the function is executed by parallel tasks.
  
  For example, if the function sends a note for each initial call to the function, one note is sent for each parallel task instead of once for the function invocation.

- **NOT DETERMINISTIC**
  
  A user-defined function that is non-deterministic can generate incorrect results if it is run under a parallel task.
  
  For example, suppose that you execute the following query under parallel tasks:
  
  ```sql
  SELECT * FROM T1 WHERE C1 = COUNTER();
  ```
  
  COUNTER is a user-defined function that increments a variable in the scratchpad every time it is invoked. Counter is non-deterministic because the same input does not always produce the same output. Table T1 contains one column, C1, that has the following values:

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
</table>

  When the query is executed with no parallelism, Db2 invokes COUNTER once for each row of table T1, and there is one scratchpad for counter, which Db2 initializes the first time that COUNTER executes. COUNTER returns 1 the first time it executes, 2 the second time, and so on. The result table for the query has the following values:

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
</table>
Now suppose that the query is run with parallelism, and Db2 creates three parallel tasks. Db2 executes the predicate WHERE C1 = COUNTER() for each parallel task. This means that each parallel task invokes its own instance of the user-defined function and has its own scratchpad. Db2 initializes the scratchpad to zero on the first call to the user-defined function for each parallel task.

If parallel task 1 processes rows 1 to 3, parallel task 2 processes rows 4 to 6, and parallel task 3 processes rows 7 to 10, the following results occur:

- When parallel task 1 executes, C1 has values 1, 2, and 3, and COUNTER returns values 1, 2, and 3, so the query returns values 1, 2, and 3.
- When parallel task 2 executes, C1 has values 4, 5, and 6, but COUNTER returns values 1, 2, and 3, so the query returns no rows.
- When parallel task 3, executes, C1 has values 7, 8, 9, and 10, but COUNTER returns values 1, 2, 3, and 4, so the query returns no rows.

Thus, instead of returning the 10 rows that you might expect from the query, Db2 returns only 3 rows.

**Related concepts**

Java stored procedures and user-defined functions (Db2 Application Programming for Java)

**Parameters for external user-defined functions**

To receive parameters from and pass parameters to an invoker of an external user-defined function, you must understand the structure of the parameter list. You must also understand the meaning of each parameter, and whether Db2 or your user-defined function sets the value of each parameter.

The following figure shows the structure of the parameter list that Db2 passes to a user-defined function. An explanation of each parameter follows.
Figure 8. Parameter conventions for a user-defined function

Input parameter values

Db2 obtains the input parameters from the invoker's parameter list, and your user-defined function receives those parameters according to the rules of the host language in which the user-defined function is written. The number of input parameters is the same as the number of parameters in the user-defined function invocation. If one of the parameters in the function invocation is an expression, Db2 evaluates the expression and assigns the result of the expression to the parameter.

For all data types except LOBs, ROWIDs, locators, and VARCHAR (with C language), see the tables listed in the following table for the host data types that are compatible with the data types in the user-defined function definition.
### Table 42. Listing of tables of compatible data types

<table>
<thead>
<tr>
<th>Language</th>
<th>Compatible data types table</th>
</tr>
</thead>
<tbody>
<tr>
<td>Assembler</td>
<td>“Compatibility of SQL and language data types” on page 484</td>
</tr>
<tr>
<td>C</td>
<td>“Compatibility of SQL and language data types” on page 484</td>
</tr>
<tr>
<td>COBOL</td>
<td>“Compatibility of SQL and language data types” on page 484</td>
</tr>
<tr>
<td>PL/I</td>
<td>“Compatibility of SQL and language data types” on page 484</td>
</tr>
</tbody>
</table>

For LOBs, ROWIDs, and locators, see the following table for the assembler data types that are compatible with the data types in the user-defined function definition.

### Table 43. Compatible assembler language declarations for LOBs, ROWIDs, and locators

<table>
<thead>
<tr>
<th>SQL data type in definition</th>
<th>Assembler declaration</th>
</tr>
</thead>
<tbody>
<tr>
<td>TABLE LOCATOR</td>
<td>DS FL4</td>
</tr>
<tr>
<td>BLOB LOCATOR</td>
<td>DS FL4</td>
</tr>
<tr>
<td>CLOB LOCATOR</td>
<td>DS FL4</td>
</tr>
<tr>
<td>DBCLOB LOCATOR</td>
<td>DS FL4</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>SQL data type in definition</th>
<th>Assembler declaration</th>
</tr>
</thead>
<tbody>
<tr>
<td>BLOB(n)</td>
<td>If ( n \leq 65535 ):</td>
</tr>
</tbody>
</table>
|                             | \[ \begin{align*} | var & DS 0FL4 \\
|                             | var_length & DS FL4 \\
|                             | var_data & DS CLn \end{align*} \] |
|                             | If \( n > 65535 \): |
|                             | \[ \begin{align*} | var & DS 0FL4 \\
|                             | var_length & DS FL4 \\
|                             | var_data & DS CL65535 \\
|                             | ORG var_data+(n-65535) \end{align*} \] |

<table>
<thead>
<tr>
<th>SQL data type in definition</th>
<th>Assembler declaration</th>
</tr>
</thead>
<tbody>
<tr>
<td>CLOB(n)</td>
<td>If ( n \leq 65535 ):</td>
</tr>
</tbody>
</table>
|                             | \[ \begin{align*} | var & DS 0FL4 \\
|                             | var_length & DS FL4 \\
|                             | var_data & DS CLn \end{align*} \] |
|                             | If \( n > 65535 \): |
|                             | \[ \begin{align*} | var & DS 0FL4 \\
|                             | var_length & DS FL4 \\
|                             | var_data & DS CL65535 \\
|                             | ORG var_data+(n-65535) \end{align*} \] |
Table 43. Compatible assembler language declarations for LOBs, ROWIDs, and locators (continued)

<table>
<thead>
<tr>
<th>SQL data type in definition</th>
<th>Assembler declaration</th>
</tr>
</thead>
<tbody>
<tr>
<td>DBCLOB(n)</td>
<td>If n (=2*n) &lt;= 65534:</td>
</tr>
<tr>
<td></td>
<td>var DS 0FL4</td>
</tr>
<tr>
<td></td>
<td>var_length DS FL4</td>
</tr>
<tr>
<td></td>
<td>var_data DS CLm</td>
</tr>
<tr>
<td></td>
<td>If n &gt; 65534:</td>
</tr>
<tr>
<td></td>
<td>var DS 0FL4</td>
</tr>
<tr>
<td></td>
<td>var_length DS FL4</td>
</tr>
<tr>
<td></td>
<td>var_data DS CL65534</td>
</tr>
<tr>
<td></td>
<td>ORG var_data+(m-65534)</td>
</tr>
<tr>
<td>ROWID</td>
<td>DS HL2,CL40</td>
</tr>
</tbody>
</table>

For LOBs, ROWIDs, VARCHARs, and locators see the following table for the C data types that are compatible with the data types in the user-defined function definition.

Table 44. Compatible C language declarations for LOBs, ROWIDs, VARCHARs, and locators

<table>
<thead>
<tr>
<th>SQL data type in definition</th>
<th>C declaration</th>
</tr>
</thead>
<tbody>
<tr>
<td>TABLE LOCATOR</td>
<td>unsigned long</td>
</tr>
<tr>
<td>BLOB LOCATOR</td>
<td></td>
</tr>
<tr>
<td>CLOB LOCATOR</td>
<td></td>
</tr>
<tr>
<td>DBCLOB LOCATOR</td>
<td></td>
</tr>
<tr>
<td>BLOB(n)</td>
<td>struct {</td>
</tr>
<tr>
<td></td>
<td>unsigned long</td>
</tr>
<tr>
<td></td>
<td>length;</td>
</tr>
<tr>
<td></td>
<td>char data[n];</td>
</tr>
<tr>
<td></td>
<td>}</td>
</tr>
<tr>
<td></td>
<td>var;</td>
</tr>
<tr>
<td>CLOB(n)</td>
<td>struct {</td>
</tr>
<tr>
<td></td>
<td>unsigned long</td>
</tr>
<tr>
<td></td>
<td>length;</td>
</tr>
<tr>
<td></td>
<td>char var_data[n];</td>
</tr>
<tr>
<td></td>
<td>}</td>
</tr>
<tr>
<td></td>
<td>var;</td>
</tr>
<tr>
<td>DBCLOB(n)</td>
<td>struct {</td>
</tr>
<tr>
<td></td>
<td>unsigned long</td>
</tr>
<tr>
<td></td>
<td>length;</td>
</tr>
<tr>
<td></td>
<td>sqldbchar</td>
</tr>
<tr>
<td></td>
<td>data[n];</td>
</tr>
<tr>
<td></td>
<td>}</td>
</tr>
<tr>
<td></td>
<td>var;</td>
</tr>
<tr>
<td>ROWID</td>
<td>struct {</td>
</tr>
<tr>
<td></td>
<td>short int length;</td>
</tr>
<tr>
<td></td>
<td>char data[40];</td>
</tr>
<tr>
<td></td>
<td>}</td>
</tr>
<tr>
<td></td>
<td>var;</td>
</tr>
</tbody>
</table>
### Table 44. Compatible C language declarations for LOBs, ROWIDs, VARCHARs, and locators (continued)

<table>
<thead>
<tr>
<th>SQL data type in definition</th>
<th>C declaration</th>
</tr>
</thead>
<tbody>
<tr>
<td>VARCHAR(n)²</td>
<td>If PARAMETER VARCHAR NULTERM is specified or implied:</td>
</tr>
<tr>
<td></td>
<td>char data[n+1];</td>
</tr>
<tr>
<td></td>
<td>If PARAMETER VARCHAR STRUCTURE is specified:</td>
</tr>
<tr>
<td></td>
<td>struct</td>
</tr>
<tr>
<td></td>
<td>{short len;</td>
</tr>
<tr>
<td></td>
<td>char data[n];</td>
</tr>
<tr>
<td></td>
<td>} var;</td>
</tr>
</tbody>
</table>

**Note:**

1. The SQLUDF file, which is in data set DSN1210.SDSNC.H, includes the typedef sqldbchar. Using sqldbchar lets you manipulate DBCS and Unicode UTF-16 data in the same format in which it is stored in Db2. sqldbchar also makes applications easier to port to other Db2 platforms.
2. This row does not apply to VARCHAR(n) FOR BIT DATA. BIT DATA is always passed in a structured representation.

For LOBs, ROWIDs, and locators, see the following table for the COBOL data types that are compatible with the data types in the user-defined function definition.

### Table 45. Compatible COBOL declarations for LOBs, ROWIDs, and locators

<table>
<thead>
<tr>
<th>SQL data type in definition</th>
<th>COBOL declaration</th>
</tr>
</thead>
<tbody>
<tr>
<td>TABLE LOCATOR</td>
<td>01 var PIC S9(9) COMP-5</td>
</tr>
<tr>
<td>BLOB LOCATOR</td>
<td>01 var.</td>
</tr>
<tr>
<td>CLOB LOCATOR</td>
<td>01 var.</td>
</tr>
<tr>
<td>DBCLOB LOCATOR</td>
<td>01 var.</td>
</tr>
<tr>
<td>BLOB(n)</td>
<td>49 var-LENGTH PIC S9(9) COMP-5.</td>
</tr>
<tr>
<td></td>
<td>49 var-DATA PIC X(n).</td>
</tr>
<tr>
<td>CLOB(n)</td>
<td>49 var-LENGTH PIC S9(9) COMP-5.</td>
</tr>
<tr>
<td></td>
<td>49 var-DATA PIC X(n).</td>
</tr>
<tr>
<td>DBCLOB(n)</td>
<td>49 var-LENGTH PIC S9(9) COMP-5.</td>
</tr>
<tr>
<td></td>
<td>49 var-DATA PIC G(n) DISPLAY-1.</td>
</tr>
<tr>
<td>ROWID</td>
<td>01 var.</td>
</tr>
<tr>
<td></td>
<td>49 var-LEN PIC S9(4) COMP-5.</td>
</tr>
<tr>
<td></td>
<td>49 var-TEXT PIC X(40).</td>
</tr>
</tbody>
</table>

For LOBs, ROWIDs, and locators, see the following table for the PL/I data types that are compatible with the data types in the user-defined function definition.
<table>
<thead>
<tr>
<th>SQL data type in definition</th>
<th>PL/I</th>
</tr>
</thead>
<tbody>
<tr>
<td>TABLE LOCATOR</td>
<td>BIN FIXED(31)</td>
</tr>
<tr>
<td>BLOB LOCATOR</td>
<td>If n &lt;= 32767:</td>
</tr>
<tr>
<td>CLOB LOCATOR</td>
<td>01 var, 03 var_LENGTH BIN FIXED(31), 03 var_DATA CHAR(n);</td>
</tr>
<tr>
<td>DBCLOB LOCATOR</td>
<td>01 var, 02 var_LENGTH BIN FIXED(31), 02 var_DATA, 03 var_DATA1(n) CHAR(32767), 03 var_DATA2 CHAR(mod(n,32767));</td>
</tr>
<tr>
<td>BLOB(n)</td>
<td>If n &gt; 32767:</td>
</tr>
<tr>
<td>CLOB(n)</td>
<td>01 var, 02 var_LENGTH BIN FIXED(31), 02 var_DATA, 03 var_DATA1(n) CHAR(32767), 03 var_DATA2 CHAR(mod(n,32767));</td>
</tr>
<tr>
<td>DBCLOB(n)</td>
<td>If n &lt;= 16383:</td>
</tr>
<tr>
<td></td>
<td>01 var, 03 var_LENGTH BIN FIXED(31), 03 var_DATA GRAPHIC(n);</td>
</tr>
<tr>
<td>ROWID</td>
<td>If n &gt; 16383:</td>
</tr>
<tr>
<td></td>
<td>01 var, 02 var_LENGTH BIN FIXED(31), 02 var_DATA, 03 var_DATA1(n) GRAPHIC(16383), 03 var_DATA2 GRAPHIC(mod(n,16383));</td>
</tr>
<tr>
<td>ROWID</td>
<td>CHAR(40) VAR;</td>
</tr>
</tbody>
</table>
Result parameters: Set these values in your user-defined function before exiting. For a user-defined scalar function, you return one result parameter. For a user-defined table function, you return the same number of parameters as columns in the RETURNS TABLE clause of the CREATE FUNCTION statement. Db2 allocates a buffer for each result parameter value and passes the buffer address to the user-defined function. Your user-defined function places each result parameter value in its buffer. You must ensure that the length of the value you place in each output buffer does not exceed the buffer length. Use the SQL data type and length in the CREATE FUNCTION statement to determine the buffer length.

See “Parameters for external user-defined functions” on page 186 to determine the host data type to use for each result parameter value. If the CREATE FUNCTION statement contains a CAST FROM clause, use a data type that corresponds to the SQL data type in the CAST FROM clause. Otherwise, use a data type that corresponds to the SQL data type in the RETURNS or RETURNS TABLE clause.

To improve performance for user-defined table functions that return many columns, you can pass values for a subset of columns to the invoker. For example, a user-defined table function might be defined to return 100 columns, but the invoker needs values for only two columns. Use the DBINFO parameter to indicate to Db2 the columns for which you will return values. Then return values for only those columns. See DBINFO for information about how to indicate the columns of interest.

Input parameter indicators: These are SMALLINT values, which Db2 sets before it passes control to the user-defined function. You use the indicators to determine whether the corresponding input parameters are null. The number and order of the indicators are the same as the number and order of the input parameters. On entry to the user-defined function, each indicator contains one of these values:

0 The input parameter value is not null.

negative The input parameter value is null.

Code the user-defined function to check all indicators for null values unless the user-defined function is defined with RETURNS NULL ON NULL INPUT. A user-defined function defined with RETURNS NULL ON NULL INPUT executes only if all input parameters are not null.

Result indicators: These are SMALLINT values, which you must set before the user-defined function ends to indicate to the invoking program whether each result parameter value is null. A user-defined scalar function has one result indicator. A user-defined table function has the same number of result indicators as the number of result parameters. The order of the result indicators is the same as the order of the result parameters. Set each result indicator to one of these values:

0 or positive The result parameter is not null.

negative The result parameter is null.

SQLSTATE value: This CHAR(5) value represents the SQLSTATE that is passed in to the program from the database manager. The initial value is set to ‘00000’. Although the SQLSTATE is usually not set by the program, it can be set as the result SQLSTATE that is used to return an error or a warning. Returned values that start with anything other than ‘00’, ‘01’, or ‘02’ are error conditions.

User-defined function name: Db2 sets this value in the parameter list before the user-defined function executes. This value is VARCHAR(257): 128 bytes for the schema name, 1 byte for a period, and 128 bytes for the user-defined function name. If you use the same code to implement multiple versions of a user-defined function, you can use this parameter to determine which version of the function the invoker wants to execute.

Specific name: Db2 sets this value in the parameter list before the user-defined function executes. This value is VARCHAR(128) and is either the specific name from the CREATE FUNCTION statement or a specific name that Db2 generated. If you use the same code to implement multiple versions of a user-defined function, you can use this parameter to determine which version of the function the invoker wants to execute.
**Diagnostic message:** Your user-defined function can set this CHAR or VARCHAR value to a character string of up to 1000 bytes before exiting. Use this area to pass descriptive information about an error or warning to the invoker.

Db2 allocates a buffer for this area and passes you the buffer address in the parameter list. At least the first 17 bytes of the value you put in the buffer appear in the SQLERRMC field of the SQLCA that is returned to the invoker. The exact number of bytes depends on the number of other tokens in SQLERRMC. Do not use X'FF' in your diagnostic message. Db2 uses this value to delimit tokens.

**Scratchpad:** If the definer specified SCRATCHPAD in the CREATE FUNCTION statement, Db2 allocates a buffer for the scratchpad area and passes its address to the user-defined function. Before the user-defined function is invoked for the first time in an SQL statement, Db2 sets the length of the scratchpad in the first 4 bytes of the buffer and then sets the scratchpad area to X'00'. Db2 does not reinitialize the scratchpad between invocations of a correlated subquery.

You must ensure that your user-defined function does not write more bytes to the scratchpad than the scratchpad length.

**Call type:** For a user-defined scalar function, if the definer specified FINAL CALL in the CREATE FUNCTION statement, Db2 passes this parameter to the user-defined function. For a user-defined table function, Db2 always passes this parameter to the user-defined function.

On entry to a user-defined scalar function, the call type parameter has one of the following values:

- `1`  
  This is the *first call* to the user-defined function for the SQL statement. For a first call, all input parameters are passed to the user-defined function. In addition, the scratchpad, if allocated, is set to binary zeros.

0  
This is a *normal call*. For a normal call, all the input parameters are passed to the user-defined function. If a scratchpad is also passed, Db2 does not modify it.

1  
This is a *final call*. For a final call, no input parameters are passed to the user-defined function. If a scratchpad is also passed, Db2 does not modify it.

This type of final call occurs when the invoking application explicitly closes a cursor. When a value of 1 is passed to a user-defined function, the user-defined function can execute SQL statements.

255  
This is a *final call*. For a final call, no input parameters are passed to the user-defined function. If a scratchpad is also passed, Db2 does not modify it.

This type of final call occurs when the invoking application executes a COMMIT or ROLLBACK statement, or when the invoking application abnormally terminates. When a value of 255 is passed to the user-defined function, the user-defined function cannot execute any SQL statements, except for CLOSE CURSOR. If the user-defined function executes any close cursor statements during this type of final call, the user-defined function should tolerate SQLCODE -501 because Db2 might have already closed cursors before the final call.

During the first call, your user-defined scalar function should acquire any system resources it needs. During the final call, the user-defined scalar function should release any resources it acquired during the first call. The user-defined scalar function should return a result value only during normal calls. Db2 ignores any results that are returned during a final call. However, the user-defined scalar function can set the SQLSTATE and diagnostic message area during the final call.

If an invoking SQL statement contains more than one user-defined scalar function, and one of those user-defined functions returns an error SQLSTATE, Db2 invokes all of the user-defined functions for a final call, and the invoking SQL statement receives the SQLSTATE of the first user-defined function with an error.

On entry to a user-defined table function, the call type parameter has one of the following values:
This is the first call to the user-defined function for the SQL statement. A first call occurs only if the FINAL CALL keyword is specified in the user-defined function definition. For a first call, all input parameters are passed to the user-defined function. In addition, the scratchpad, if allocated, is set to binary zeros.

This is the open call to the user-defined function by an SQL statement. If FINAL CALL is not specified in the user-defined function definition, all input parameters are passed to the user-defined function, and the scratchpad, if allocated, is set to binary zeros during the open call. If FINAL CALL is specified for the user-defined function, Db2 does not modify the scratchpad.

This is a fetch call to the user-defined function by an SQL statement. For a fetch call, all input parameters are passed to the user-defined function. If a scratchpad is also passed, Db2 does not modify it.

This is a close call. For a close call, no input parameters are passed to the user-defined function. If a scratchpad is also passed, Db2 does not modify it.

This is a final call. This type of final call occurs only if FINAL CALL is specified in the user-defined function definition. For a final call, no input parameters are passed to the user-defined function. If a scratchpad is also passed, Db2 does not modify it.

This type of final call occurs when the invoking application executes a CLOSE CURSOR statement.

This is a final call. For a final call, no input parameters are passed to the user-defined function. If a scratchpad is also passed, Db2 does not modify it.

This type of final call occurs when the invoking application executes a COMMIT or ROLLBACK statement, or when the invoking application abnormally terminates. When a value of 255 is passed to the user-defined function, the user-defined function cannot execute any SQL statements, except for CLOSE CURSOR. If the user-defined function executes any close cursor statements during this type of final call, the user-defined function should tolerate SQLCODE -501 because Db2 might have already closed cursors before the final call.

If a user-defined table function is defined with FINAL CALL, the user-defined function should allocate any resources it needs during the first call and release those resources during the final call that sets a value of 2.

If a user-defined table function is defined with NO FINAL CALL, the user-defined function should allocate any resources it needs during the open call and release those resources during the close call.

During a fetch call, the user-defined table function should return a row. If the user-defined function has no more rows to return, it should set the SQLSTATE to 02000.

During the close call, a user-defined table function can set the SQLSTATE and diagnostic message area.

If a user-defined table function is invoked from a subquery, the user-defined table function receives a CLOSE call for each invocation of the subquery within the higher level query, and a subsequent OPEN call for the next invocation of the subquery within the higher level query.

**DBINFO:** If the definer specified DBINFO in the CREATE FUNCTION statement, Db2 passes the DBINFO structure to the user-defined function. DBINFO contains information about the environment of the user-defined function caller. It contains the following fields, in the order shown:

**Location name length**
An unsigned 2-byte integer field. It contains the length of the location name in the next field.

**Location name**
A 128-byte character field. It contains the name of the location to which the invoker is currently connected.
Authorization ID length
An unsigned 2-byte integer field. It contains the length of the authorization ID in the next field.

Authorization ID
A 128-byte character field. It contains the authorization ID of the application from which the user-defined function is invoked, padded on the right with blanks. If this user-defined function is nested within other user-defined functions, this value is the authorization ID of the application that invoked the highest-level user-defined function.

Subsystem code page
A 48-byte structure that consists of 10 integer fields and an eight-byte reserved area. These fields provide information about the CCSIDs of the subsystem from which the user-defined function is invoked.

Table qualifier length
An unsigned 2-byte integer field. It contains the length of the table qualifier in the next field. If the table name field is not used, this field contains 0.

Table qualifier
A 128-byte character field. It contains the qualifier of the table that is specified in the table name field.

Table name length
An unsigned 2-byte integer field. It contains the length of the table name in the next field. If the table name field is not used, this field contains 0.

Table name
A 128-byte character field. This field contains the name of the table for the update or insert operation if the reference to the user-defined function in the invoking SQL statement is in one of the following places:
- The right side of a SET clause in an update operation
- In the VALUES list of an insert operation
Otherwise, this field is blank.

Column name length
An unsigned 2-byte integer field. It contains the length of the column name in the next field. If no column name is passed to the user-defined function, this field contains 0.

Column name
A 128-byte character field. This field contains the name of the column that the update or insert operation modifies if the reference to the user-defined function in the invoking SQL statement is in one of the following places:
- The right side of a SET clause in an update operation
- In the VALUES list of an insert operation
Otherwise, this field is blank.

Product information
An 8-byte character field that identifies the product on which the user-defined function executes.
The product identifier (PRDID) value is an 8-byte character value in pppvvrmm format, where: ppp is a 3-letter product code; vv is the version; rr is the release; and m is the modification level. In Db2 12 for z/OS, the modification level indicates a range of function levels:
- DSN12015 for V12R1M500 or higher.
- DSN12010 for V12R1M100.

For more information, see Product identifier (PRDID) values in Db2 for z/OS (Db2 Administration Guide).

Reserved area
2 bytes.
**Operating system**

A 4-byte integer field. It identifies the operating system on which the program that invokes the user-defined function runs. The value is one of these:

0  
Unknown

1  
OS/2

3  
Windows

4  
AIX®

5  
Windows NT

6  
HP-UX

7  
Solaris

8  
z/OS

13  
Siemens Nixdorf

15  
Windows 95

16  
SCO UNIX

18  
Linux®

19  
DYNIX/ptx®

24  
Linux for S/390®

25  
Linux on IBM Z

26  
Linux/IA64

27  
Linux/PPC

28  
Linux/PPC64

29  
Linux/AMD64

400®  
iSeries

**Number of entries in table function column list**

An unsigned 2-byte integer field.

**Reserved area**

26 bytes.

**Table function column list pointer**

If a table function is defined, this field is a pointer to an array that contains 1000 2-byte integers. Db2 dynamically allocates the array. If a table function is not defined, this pointer is null.
Only the first $n$ entries, where $n$ is the value in the field entitled number of entries in table function column list, are of interest. $n$ is greater than or equal to 0 and less than or equal to the number result columns defined for the user-defined function in the RETURNS TABLE clause of the CREATE FUNCTION statement. The values correspond to the numbers of the columns that the invoking statement needs from the table function. A value of 1 means the first defined result column, 2 means the second defined result column, and so on. The values can be in any order. If $n$ is equal to 0, the first array element is 0. This is the case for a statement like the following one, where the invoking statement needs no column values.

```
SELECT COUNT(*) FROM TABLE(TF(…)) AS QQ
```

This array represents an opportunity for optimization. The user-defined function does not need to return all values for all the result columns of the table function. Instead, the user-defined function can return only those columns that are needed in the particular context, which you identify by number in the array. However, if this optimization complicates the user-defined function logic enough to cancel the performance benefit, you might choose to return every defined column.

**Unique application identifier**

This field is a pointer to a string that uniquely identifies the application's connection to Db2. The string is regenerated for each connection to Db2.

The string is the LUWID, which consists of a fully-qualified LU network name followed by a period and an LUW instance number. The LU network name consists of a 1- to 8-character network ID, a period, and a 1- to 8-character network LU name. The LUW instance number consists of 12 hexadecimal characters that uniquely identify the unit of work.

**Reserved area**

20 bytes.

If you write your user-defined function in C or C++, you can use the declarations in member SQLUDF of DSN1210.SDSNC.H for many of the passed parameters. To include SQLUDF, make these changes to your program:

- Put this statement in your source code:
  ```
  #include <sqludf.h>
  ```

- Include the DSN1210.SDSNC.H data set in the SYSLIB concatenation for the compiler step of your program preparation job.

- Specify the NOMARGINS and NOSEQUENCE options in the compiler step of your program preparation job.

**Examples of receiving parameters in a user-defined function:**

The following examples show how a user-defined function that is written in each of the supported host languages receives the parameter list that is passed by Db2.

These examples assume that the user-defined function is defined with the SCRATCHPAD, FINAL CALL, and DBINFO parameters.

**Assembler:** The follow figure shows the parameter conventions for a user-defined scalar function that is written as a main program that receives two parameters and returns one result. For an assembler language user-defined function that is a subprogram, the conventions are the same. In either case, you must include the CEEENTRY and CEEEXIT macros.

```
MYMAIN   CEEENTRY AUTO=PROGSIZE,MAIN=YES,PLIST=OS
USING PROGAREA, R13
L     R7,0(R1)            GET POINTER TO PARM1
MVC   PARM1(4),0(R7)      MOVE VALUE INTO LOCAL COPY OF PARM1
L     R7,4(R1)            GET POINTER TO PARM2
MVC   PARM2(4),0(R7)      MOVE VALUE INTO LOCAL COPY OF PARM2
L     R7,12(R1)           GET POINTER TO INDICATOR 1
MVC   F_IND1(2),0(R7)  MOVE PARM1 INDICATOR TO LOCAL STORAGE
LH    R7,F_IND1        MOVE PARM1 INDICATOR INTO R7
LTR   R7,R7               CHECK IF IT IS NEGATIVE
```
C or C++: For C or C++ user-defined functions, the conventions for passing parameters are different for main programs and subprograms.

For subprograms, you pass the parameters directly. For main programs, you use the standard argc and argv variables to access the input and output parameters:

- The argv variable contains an array of pointers to the parameters that are passed to the user-defined function. All string parameters that are passed back to Db2 must be null terminated.
  - argv[0] contains the address of the load module name for the user-defined function.
  - argv[1] through argv[n] contain the addresses of parameters 1 through n.
- The argc variable contains the number of parameters that are passed to the external user-defined function, including argv[0].

The following figure shows the parameter conventions for a user-defined scalar function that is written as a main program that receives two parameters and returns one result.

```c
#include <stdlib.h>
#include <stdio.h>

main(argc,argv)
    int argc;
    char *argv[];
{
    /* Assume that the user-defined function invocation/ */
    /* included 2 input parameters in the parameter */
    /* list. Also assume that the definition includes */
    /* the SCRATCHPAD, FINAL CALL, and DBINFO options, */
    /* so DB2 passes the scratchpad, calltype, and */
    /* dbinfo parameters. */
    /* The argv vector contains these entries: */
    /* argv[0] 1 load module name */
    /* argv[1-2] 2 input parms */
    /* argv[3] 1 result parm */
    /* argv[4-5] 2 null indicators */
    /* argv[6] 1 result null indicator */
    /* argv[7] 1 SQLSTATE variable */
    /* argv[8] 1 qualified func name */
    /* argv[9] 1 specific func name */
    /* argv[10] 1 diagnostic string */

    // Code goes here... */
```
The following figure shows the parameter conventions for a user-defined scalar function written as a C subprogram that receives two parameters and returns one result.

```c
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <sqludf.h>

#pragma runopts(plist(os))

int main() {
    /* Assume the first parameter is an integer. */
    int parm1;
    parm1 = *(int *) argv[1];

    /* Access the null indicator for the first parameter. */
    short int ind1;
    ind1 = *(short int *) argv[4];

    strcpy(argv[7], "xxxxx");

    char f_func[28];
    strcpy(f_func, argv[8]);

    char f_spec[19];
    strcpy(f_spec, argv[9]);

    strcpy(argv[10], "yyyyyyyy");

    char l_result[11];
    strcpy(argv[3], l_result);

    return 0;
}
```

The following figure shows the parameter conventions for a user-defined scalar function written as a C subprogram that receives two parameters and returns one result.
void myfunc(long *parm1, char parm2[11], char result[11],
short *f_ind1, short *f_ind2, short *f_indr,
char udf_sqlstate[6], char udf_fname[138],
char udf_specname[129], char udf_msgtext[71],
struct sqludf_scratchpad *udf_scratchpad,
long *udf_call_type,
struct sql_dbinfo *udf_dbinfo);
{
    /***************************************************/
    /* Declare local copies of parameters             */
    /***************************************************/
    int _p1;
    char _p2[11];
    short int _ind1;
    short int _ind2;
    char _udf_sqlstate[6]; /* SQLSTATE */
    char _udf_fname[138]; /* function name */
    char _udf_specname[129]; /* specific function name */
    char _udf_msgtext[71] /* diagnostic message text*/
    long _udf_call_type; /* call type */
    sqludf_dbinfo *udf_dbinfo /* dbinfo */
    /***************************************************/
    /* Copy each of the parameters in the parameter   */
    /* list into a local variable to demonstrate      */
    /* how the parameters can be referenced.          */
    /***************************************************/
    _p1 = *parm1;
    strcpy(_p2, parm2);
    _ind1 = *f_ind1;
    _ind1 = *f_ind2;
    strcpy(_udf_sqlstate, udf_sqlstate);
    strcpy(_udf_fname, udf_fname);
    strcpy(_udf_specname, udf_specname);
    _udf_call_type = *udf_call_type;
    strcpy(_udf_msgtext, udf_msgtext);
    memcpy(&(udf_scratchpad), udf_scratchpad, sizeof(udf_scratchpad));
    memcpy(&(udf_dbinfo), udf_dbinfo, sizeof(udf_dbinfo));
}

The following figure shows the parameter conventions for a user-defined scalar function that is written as a C++ subprogram that receives two parameters and returns one result. This example demonstrates that you must use an extern "C" modifier to indicate that you want the C++ subprogram to receive parameters according to the C linkage convention. This modifier is necessary because the CEEPIPI CALL_SUB interface, which Db2 uses to call the user-defined function, passes parameters using the C linkage convention.

#pragma runopts(plist(os))
#include <stdlib.h>
#include <stdio.h>
#include <sqludf.h>

extern "C" void myfunc(long *parm1, char parm2[11],
char result[11], short *f_ind1, short *f_ind2, short *f_indr,
char udf_sqlstate[6], char udf_fname[138],
char udf_specname[129], char udf_msgtext[71],
struct sqludf_scratchpad *udf_scratchpad,
long *udf_call_type,
struct sql_dbinfo *udf_dbinfo);
{
    /***************************************************/
    /* Define local copies of parameters.              */
    /***************************************************/
    int _p1;
    char _p2[11];
    short int _ind1;
    short int _ind2;
    char _udf_sqlstate[6]; /* SQLSTATE */
    char _udf_fname[138]; /* function name */
    char _udf_specname[129]; /* specific function name */
    char _udf_msgtext[71] /* diagnostic message text*/
    sqludf_scratchpad *udf_scratchpad /* scratchpad */
    long _udf_call_type; /* call type */
    sqludf_dbinfo *udf_dbinfo /* dbinfo */
    /***************************************************/
}
/* Copy each of the parameters in the parameter list into a local variable to demonstrate how the parameters can be referenced. */
l_p1 = *parm1;
strcpy(l_p2,parm2);
l_ind1 = *f_ind1;
l_ind1 = *f_ind2;
strcpy(ludf_sqlstate,udf_sqlstate);
strcpy(ludf_fname,udf_fname);
strcpy(ludf_specname,udf_specname);
l_udf_call_type = *udf_call_type;
strcpy(ludf_msgtext,udf_msgtext);
memcpy(&ludf_scratchpad,udf_scratchpad,sizeof(ludf_scratchpad));
memcpy(&ludf_dbinfo,udf_dbinfo,sizeof(ludf_dbinfo));
}

**COBOL:** The following figure shows the parameter conventions for a user-defined table function that is written as a main program that receives two parameters and returns two results. For a COBOL user-defined function that is a subprogram, the conventions are the same.

```cobol
CBL APOST,RES,RENT
IDENTIFICATION DIVISION.
   :
DATA DIVISION.
   :
LINKAGE SECTION.
   ******************************************************
   * Declare each of the parameters *
   ******************************************************
   01 UDFPARM1 PIC S9(9) USAGE COMP.
   01 UDFPARM2 PIC X(10).
   :
   ******************************************************
   * Declare these variables for result parameters *
   ******************************************************
   01 UDFRESULT1 PIC X(10).
   01 UDFRESULT2 PIC X(10).
   :
   ******************************************************
   * Declare a null indicator for each parameter *
   ******************************************************
   01 UDF-IND1 PIC S9(4) USAGE COMP.
   01 UDF-IND2 PIC S9(4) USAGE COMP.
   :
   ******************************************************
   * Declare the SQLSTATE that can be set by the user-defined function *
   ******************************************************
   01 UDF-SQLSTATE PIC X(5).
   ******************************************************
   * Declare the qualified function name *
   ******************************************************
   01 UDF-FUNC.
      49 UDF-FUNC-LEN PIC 9(4) USAGE BINARY.
      49 UDF-FUNC-TEXT PIC X(137).
   ******************************************************
   * Declare the specific function name *
   ******************************************************
   01 UDF-SPEC.
      49 UDF-SPEC-LEN PIC 9(4) USAGE BINARY.
      49 UDF-SPEC-TEXT PIC X(128).
   ******************************************************
   * Declare SQL diagnostic message token *
   ******************************************************
   01 UDF-DIAG.
      49 UDF-DIAG-LEN PIC 9(4) USAGE BINARY.
      49 UDF-DIAG-TEXT PIC X(1000).
   ******************************************************
   * Declare the scratchpad *
```
UDF-SCRATCHPAD.
49 UDF-SPAD-LEN PIC 9(9) USAGE BINARY.
49 UDF-SPAD-TEXT PIC X(100).

Declare the call type

UDF-CALL-TYPE PIC 9(9) USAGE BINARY.

CONSTANTS FOR DB2-ENCODING SCHEME.

SQLUDF-ASCII PIC 9(9) VALUE 1.
SQLUDF-EBCDIC PIC 9(9) VALUE 2.
SQLUDF-UNICODE PIC 9(9) VALUE 3.

Structure used for DBINFO

SQLUDF-DBINFO.

  location name length
  05 DBNAMELEN PIC 9(4) USAGE BINARY.
  location name
  05 DBNAME PIC X(128).

  authorization ID length
  05 AUTHIDLEN PIC 9(4) USAGE BINARY.
  authorization ID
  05 AUTHID PIC X(128).

  environment CCSID information
  05 CODEPG PIC X(48).
  05 CDPG-DB2 REDEFINES CODEPG.
  10 DB2-CCSIDS OCCURS 3 TIMES.
    15 DB2-SBCS PIC 9(9) USAGE BINARY.
    15 DB2-DBCS PIC 9(9) USAGE BINARY.
    15 DB2-MIXED PIC 9(9) USAGE BINARY.
  10 ENCODING-SCHEME PIC 9(9) USAGE BINARY.
  10 RESERVED PIC X(8).

  other platform-specific deprecated CCSID structures not included here
  05 TBSCHMENAME LEN PIC 9(4) USAGE BINARY.
  schema name
  05 TBSCHMENAME PIC X(128).

  table name length
  05 TBNAMELEN PIC 9(4) USAGE BINARY.
  table name
  05 TBNAME PIC X(128).

  column name length
  05 COLNAMELEN PIC 9(4) USAGE BINARY.
  column name
  05 COLNAME PIC X(128).

  product information
  05 VER-REL PIC X(8).
  reserved for expansion
  05 RESD0 PIC X(2).

  platform type
  05 PLATFORM PIC 9(9) USAGE BINARY.

  number of entries in tfcolumn list array (tfcolumn, below)
  05 NUMTFCOL PIC 9(4) USAGE BINARY.

  reserved for expansion
  05 RESD1 PIC X(26).

  tfcolumn will be allocated dynamically if TF is defined
  otherwise this will be a null pointer
  05 TFCOLUMN USAGE IS POINTER.
  Application identifier
  05 APPL-ID USAGE IS POINTER.
  reserved for expansion
  05 RESD2 PIC X(20).

PROCEDURE DIVISION USING UDFPARM1, UDFPARM2, UDFRESULT1,
UDFRESULT2, UDF-IND1, UDF-IND2,
UDF-RIND1, UDF-RIND2,
UDF-SECRET, UDF-FUNC, UDF-SPEC,
UDF-DIAG, UDF-SCRATCHPAD,
UDF-CALL-TYPE, SQLUDF-DBINFO.
PL/I: The following figure shows the parameter conventions for a user-defined scalar function that is written as a main program that receives two parameters and returns one result. For a PL/I user-defined function that is a subprogram, the conventions are the same.

```pli
*PROCESS SYSTEM(MVS);
MYMAIN: PROC(UDF_PARM1, UDF_PARM2, UDF_RESULT,
UDF_IND1, UDF_IND2, UDF_INDR,
UDF_SQLSTATE, UDF_NAME, UDF_SPEC_NAME,
UDF_DIAG_MSG, UDF_SCRATCHPAD,
UDF_CALL_TYPE, UDF_DBINFO)
OPTIONS(MAIN NOEXECOPS REENTRANT);
DCL UDF_PARM1 BIN FIXED(31);    /* first parameter */
DCL UDF_PARM2 CHAR(10);         /* second parameter */
DCL UDF_RESULT CHAR(10);        /* result parameter */
DCL UDF_IND1 BIN FIXED(15);     /* indicator for 1st parm */
DCL UDF_IND2 BIN FIXED(15);     /* indicator for 2nd parm */
DCL UDF_INDR BIN FIXED(15);     /* indicator for result */
DCL UDF_SQLSTATE CHAR(5);       /* SQLSTATE returned to DB2 */
DCL UDF_NAME CHAR(137) VARYING; /* Qualified function name */
DCL UDF_SPEC_NAME CHAR(128) VARYING; /* Specific function name */
DCL UDF_DIAG_MSG CHAR(70) VARYING; /* Diagnostic string */
DCL UDF_SCRATCHPAD /* Scratchpad */
DCL 01 UDF_DBINFO BASED(DBINFO),              /* Dbinfo                */
03 UDF_DBINFO_LLEN BIN FIXED(15),    /* location length     */
03 UDF_DBINFO_LOC CHAR(128),        /* location name       */
03 UDF_DBINFO_ALEN BIN FIXED(15),   /* auth ID  length     */
03 UDF_DBINFO_AUTH CHAR(128),       /* authorization ID    */
03 UDF_DBINFO_CDPG,                  /* environment CCSID info */
05 DB2_CCSIDS(3),
07 R1          BIN FIXED(15),/* Reserved */
07 DB2_SBCS BIN FIXED(15),    /* SBCS CCSID          */
07 R2          BIN FIXED(15), /* Reserved */
07 DB2_DBCS BIN FIXED(15),    /* DBCS CCSID          */
07 R3          BIN FIXED(15), /* Reserved */
07 DB2_MIXED BIN FIXED(15),   /* MIXED CCSID         */
05 DB2_ENCODING_SCHEME BIN FIXED(31),
05 DB2_CCSID_RESERVED CHAR(8),
03 UDF_DBINFO_SLEN BIN FIXED(15), /* schema length */
03 UDF_DBINFO_SCHEMA CHAR(128),       /* schema name         */
03 UDF_DBINFO_TLEN BIN FIXED(15),  /* table length        */
03 UDF_DBINFO_TABLE CHAR(128),       /* table name          */
03 UDF_DBINFO_CLEN BIN FIXED(15),  /* column length       */
03 UDF_DBINFO_COLUMN CHAR(128),      /* column name         */
03 UDF_DBINFO_RELVER CHAR(8),       /* DB2 release level   */
03 UDF_DBINFO_RECORD CHAR(2),       /* reserved */
03 UDF_DBINFO_PLATFORM BIN FIXED(31), /* database platform */
03 UDF_DBINFO_NUMTFCOL BIN FIXED(15), /* # of TF columns used */
03 UDF_DBINFO_RESERV1 CHAR(26),     /* reserved */
03 UDF_DBINFO_TFCOLUMN PTR,  /* -> TFcolumn list */
03 UDF_DBINFO_APPLIC_PTR, /* -> application id */
03 UDF_DBINFO_RESERV2 CHAR(29);    /* reserved */
```

Related reference
CREATE FUNCTION (external scalar) (Db2 SQL)
Making a user-defined function reentrant

A reentrant user-defined function is a function for which a single copy of the function can be used concurrently by two or more processes.

**Procedure**

Compiling and link-editing your user-defined function as reentrant is recommended. (For an assembler program, you must also code the user-defined function to be reentrant.)

Reentrant user-defined functions have the following advantages:

- The operating system does not need to load the user-defined function into storage every time the user-defined function is called.
- Multiple tasks in a WLM-established stored procedures address space can share a single copy of the user-defined function. This decreases the amount of virtual storage that is needed for code in the address space.

If your user-defined function consists of several programs, you must bind each program that contains SQL statements into a separate package. The definer of the user-defined function must have EXECUTE authority for all packages that are part of the user-defined function.

When the primary program of a user-defined function calls another program, Db2 uses the CURRENT PACKAGE PATH special register to determine the list of collections to search for the called program's package. The primary program can change this collection ID by executing the statement SET CURRENT PACKAGE PATH.

If the value of CURRENT PACKAGE PATH is blank or an empty string, Db2 uses the CURRENT PACKAGESET special register to determine the collection to search for the called program's package. The primary program can change this value by executing the statement SET CURRENT PACKAGESET.

If both special registers CURRENT PACKAGE PATH and CURRENT PACKAGESET contain a blank value, Db2 uses the method described in “Binding an application plan” on page 869 to search for the package.

**Special registers in a user-defined function or a stored procedure**

You can use all special registers in a user-defined function or a stored procedure. However, you can modify only some of those special registers.

After a user-defined function or a stored procedure completes, Db2 restores all special registers to the values they had before invocation.

The following table shows information that you need when you use special registers in a user-defined function or stored procedure.

<table>
<thead>
<tr>
<th>Special register</th>
<th>Initial value when INHERIT SPECIAL REGISTERS option is specified</th>
<th>Initial value when DEFAULT SPECIAL REGISTERS option is specified</th>
<th>Routine can use SET statement to modify?</th>
</tr>
</thead>
<tbody>
<tr>
<td>CURRENT ACCELERATOR</td>
<td>Inherited from the invoking application; otherwise, no preferred accelerator is used and Db2 will determine the target accelerator</td>
<td>The ACCELERATOR bind option value if specified for the user-defined function or stored procedure package; otherwise, no preferred accelerator is used and Db2 will determine the target accelerator</td>
<td>Yes</td>
</tr>
<tr>
<td>Special register</td>
<td>Initial value when INHERIT SPECIAL REGISTERS option is specified</td>
<td>Initial value when DEFAULT SPECIAL REGISTERS option is specified</td>
<td>Routine can use SET statement to modify?</td>
</tr>
<tr>
<td>------------------------------------------------------</td>
<td>------------------------------------------------------------------</td>
<td>-----------------------------------------------------------------</td>
<td>------------------------------------------</td>
</tr>
<tr>
<td>CURRENT APPLICATION COMPATIBILITY</td>
<td>The value of bind option APPLCOMPAT for the user-defined function or stored procedure package</td>
<td>The value of bind option APPLCOMPAT for the user-defined function or stored procedure package</td>
<td>Yes</td>
</tr>
<tr>
<td>CURRENT APPLICATION ENCODING SCHEME</td>
<td>The value of bind option ENCODING for the user-defined function or stored procedure package</td>
<td>The value of bind option ENCODING for the user-defined function or stored procedure package</td>
<td>Yes</td>
</tr>
<tr>
<td>CURRENT CLIENT_ACCTNG</td>
<td>Inherited from the invoking application</td>
<td>Inherited from the invoking application</td>
<td>Not applicable⁵</td>
</tr>
<tr>
<td>CURRENT CLIENT_APPLNAME</td>
<td>Inherited from the invoking application</td>
<td>Inherited from the invoking application</td>
<td>Not applicable⁵</td>
</tr>
<tr>
<td>CURRENT CLIENT_USERID</td>
<td>Inherited from the invoking application</td>
<td>Inherited from the invoking application</td>
<td>Not applicable⁵</td>
</tr>
<tr>
<td>CURRENT_CLIENT_WRKSTNNAME</td>
<td>Inherited from the invoking application</td>
<td>Inherited from the invoking application</td>
<td>Not applicable⁵</td>
</tr>
<tr>
<td>CURRENT DATE</td>
<td>New value for each SQL statement in the user-defined function or stored procedure package¹</td>
<td>New value for each SQL statement in the user-defined function or stored procedure package¹</td>
<td>Not applicable⁵</td>
</tr>
<tr>
<td>CURRENT DEBUG MODE</td>
<td>Inherited from the invoking application</td>
<td>DISALLOW</td>
<td>Yes</td>
</tr>
<tr>
<td>CURRENT DECFLOAT ROUNCING MODE</td>
<td>Inherited from the invoking application</td>
<td>The value of bind option ROUNDING for the user-defined function or stored procedure package</td>
<td>Yes</td>
</tr>
<tr>
<td>CURRENT DEGREE</td>
<td>CURRENT DEGREE²</td>
<td>The value of field CURRENT DEGREE on installation panel DSNTIP8</td>
<td>Yes</td>
</tr>
<tr>
<td>CURRENT EXPLAIN MODE</td>
<td>Inherited from the invoking application</td>
<td>NO</td>
<td>Yes</td>
</tr>
<tr>
<td>CURRENT GET_ACCEL_ARCHIVE</td>
<td>Inherited from the invoking application⁶; otherwise, the subsystem parameter value will be used</td>
<td>The GETACCELARCHIVE bind option value if specified for the user-defined function or stored procedure package; otherwise, the subsystem parameter value will be used</td>
<td>Yes</td>
</tr>
<tr>
<td>CURRENT LOCALE LC_CTYPE</td>
<td>Inherited from the invoking application</td>
<td>The value of field CURRENT LC_CTYPE on installation panel DSNTIPF</td>
<td>Yes</td>
</tr>
<tr>
<td>CURRENT MAINTAINED TABLE TYPES FOR OPTIMIZATION</td>
<td>Inherited from the invoking application</td>
<td>System default value</td>
<td>Yes</td>
</tr>
<tr>
<td>Special register</td>
<td>Initial value when INHERIT SPECIAL REGISTERS option is specified</td>
<td>Initial value when DEFAULT SPECIAL REGISTERS option is specified</td>
<td>Routine can use SET statement to modify?</td>
</tr>
<tr>
<td>----------------------------------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------</td>
<td>-----------------------------------------</td>
</tr>
<tr>
<td>CURRENT MEMBER</td>
<td>New value for each SET host-variable=CURRENT MEMBER statement</td>
<td>New value for each SET host-variable=CURRENT MEMBER statement</td>
<td>Not applicable</td>
</tr>
<tr>
<td>CURRENT OPTIMIZATION HINT</td>
<td>The value of bind option OPTHINT for the user-defined function or stored procedure package or inherited from the invoking application</td>
<td>The value of bind option OPTHINT for the user-defined function or stored procedure package</td>
<td>Yes</td>
</tr>
<tr>
<td>CURRENT PACKAGE PATH</td>
<td>An empty string if the routine was defined with a COLLID value; otherwise, inherited from the invoking application</td>
<td>An empty string, regardless of whether a COLLID value was specified for the routine</td>
<td>Yes</td>
</tr>
<tr>
<td>CURRENT PACKAGESET</td>
<td>Inherited from the invoking application</td>
<td>Inherited from the invoking application</td>
<td>Yes</td>
</tr>
<tr>
<td>CURRENT PATH</td>
<td>The value of bind option PATH for the user-defined function or stored procedure package or inherited from the invoking application</td>
<td>The value of bind option PATH for the user-defined function or stored procedure package</td>
<td>Yes</td>
</tr>
<tr>
<td>CURRENT PRECISION</td>
<td>Inherited from the invoking application</td>
<td>The value of field DECIMAL ARITHMETIC on installation panel DSNTIP4</td>
<td>Yes</td>
</tr>
<tr>
<td>CURRENT QUERY ACCELERATION</td>
<td>Inherited from the invoking application; otherwise, the subsystem parameter value will be used</td>
<td>The QUERYACCELERATION bind option value if specified for the user-defined function or stored procedure package; otherwise, the subsystem parameter value will be used</td>
<td>Yes</td>
</tr>
<tr>
<td>CURRENT QUERY ACCELERATION WAITFORDATA</td>
<td>Inherited from the invoking application; otherwise, the subsystem parameter value will be used</td>
<td>The ACCELERATIONWAITFORDAT A bind option value if specified for the user-defined function or stored procedure package; otherwise, the subsystem parameter value will be used</td>
<td>Yes</td>
</tr>
<tr>
<td>CURRENT REFRESH AGE</td>
<td>Inherited from the invoking application</td>
<td>System default value</td>
<td>Yes</td>
</tr>
<tr>
<td>CURRENT ROUTINE VERSION</td>
<td>Inherited from the invoking application</td>
<td>The empty string</td>
<td>Yes</td>
</tr>
<tr>
<td>CURRENT RULES</td>
<td>Inherited from the invoking application</td>
<td>The value of bind option SQLRULES for the plan that invokes a user-defined function or stored procedure</td>
<td>Yes</td>
</tr>
</tbody>
</table>
Table 47. Characteristics of special registers in a user-defined function or a stored procedure (continued)

<table>
<thead>
<tr>
<th>Special register</th>
<th>Initial value when INHERIT SPECIAL REGISTERS option is specified</th>
<th>Initial value when DEFAULT SPECIAL REGISTERS option is specified</th>
<th>Routine can use SET statement to modify?</th>
</tr>
</thead>
<tbody>
<tr>
<td>CURRENT SCHEMA</td>
<td>Inherited from the invoking application</td>
<td>The value of CURRENT SCHEMA when the routine is entered</td>
<td>Yes</td>
</tr>
<tr>
<td>CURRENT SERVER</td>
<td>Inherited from the invoking application</td>
<td>Inherited from the invoking application</td>
<td>Yes</td>
</tr>
<tr>
<td>CURRENT SQLID</td>
<td>The primary authorization ID of the application process or inherited from the invoking application</td>
<td>The primary authorization ID of the application process</td>
<td>Yes</td>
</tr>
<tr>
<td>CURRENT TEMPORAL BUSINESS.TIME</td>
<td>Inherited from the invoking application</td>
<td>NULL</td>
<td>Yes</td>
</tr>
<tr>
<td>CURRENT TEMPORAL SYSTEM.TIME</td>
<td>Inherited from the invoking application</td>
<td>NULL</td>
<td>Yes</td>
</tr>
<tr>
<td>CURRENT TIME</td>
<td>New value for each SQL statement in the user-defined function or stored procedure package</td>
<td>New value for each SQL statement in the user-defined function or stored procedure package</td>
<td>Not applicable</td>
</tr>
<tr>
<td>CURRENT TIMESTAMP</td>
<td>New value for each SQL statement in the user-defined function or stored procedure package</td>
<td>New value for each SQL statement in the user-defined function or stored procedure package</td>
<td>Not applicable</td>
</tr>
<tr>
<td>CURRENT TIMESTAMP WITH TIME ZONE</td>
<td>New value for each SQL statement in the user-defined function or stored procedure package</td>
<td>New value for each SQL statement in the user-defined function or stored procedure package</td>
<td>Not applicable</td>
</tr>
<tr>
<td>CURRENT TIME ZONE</td>
<td>Inherited from the invoking application</td>
<td>Inherited from the invoking application</td>
<td>Not applicable</td>
</tr>
<tr>
<td>ENCRYPTION PASSWORD</td>
<td>Inherited from the invoking application</td>
<td>Inherited from the invoking application</td>
<td>Yes</td>
</tr>
<tr>
<td>SESSION TIME ZONE</td>
<td>Inherited from the invoking application</td>
<td>The value of CURRENT TIME ZONE when the routine is entered</td>
<td>Yes</td>
</tr>
<tr>
<td>SESSION_USER or USER</td>
<td>Primary authorization ID of the application process</td>
<td>Primary authorization ID of the application process</td>
<td>Not applicable</td>
</tr>
</tbody>
</table>

Chapter 3. Db2 SQL programming 207
Table 47. Characteristics of special registers in a user-defined function or a stored procedure (continued)

<table>
<thead>
<tr>
<th>Special register</th>
<th>Initial value when INHERIT SPECIAL REGISTERS option is specified</th>
<th>Initial value when DEFAULT SPECIAL REGISTERS option is specified</th>
<th>Routine can use SET statement to modify?</th>
</tr>
</thead>
</table>

Notes:

1. If the user-defined function or stored procedure is invoked within the scope of a trigger, Db2 uses the timestamp for the triggering SQL statement as the timestamp for all SQL statements in the package.
2. Db2 allows parallelism at only one level of a nested SQL statement. If you set the value of the CURRENT DEGREE special register to ANY, and parallelism is disabled, Db2 ignores the CURRENT DEGREE value.
3. If the routine definition includes a specification for COLLID, Db2 sets CURRENT PACKAGESET to the value of COLLID. If both CURRENT PACKAGE PATH and COLLID are specified, the CURRENT PACKAGE PATH value takes precedence and COLLID is ignored.
4. If the function definition includes a specification for PACKAGE PATH, Db2 sets CURRENT PACKAGE PATH to the value of PACKAGE PATH.
5. Not applicable because no SET statement exists for the special register.
6. If a program within the scope of the invoking program issues a SET statement for the special register before the user-defined function or stored procedure is invoked, the special register inherits the value from the SET statement. Otherwise, the special register contains the value that is set by the bind option for the user-defined function or stored procedure package.
7. If a program within the scope of the invoking program issues a SET CURRENT SQLID statement before the user-defined function or stored procedure is invoked, the special register inherits the value from the SET statement. Otherwise, CURRENT SQLID contains the authorization ID of the application process.
8. If the user-defined function or stored procedure package uses a value other than RUN for the DYNAMICRULES bind option, the SET CURRENT SQLID statement can be executed. However, it does not affect the authorization ID that is used for the dynamic SQL statements in the package. The DYNAMICRULES value determines the authorization ID that is used for dynamic SQL statements.

Related concepts

Dynamic rules options for dynamic SQL statements
The DYNAMICRULES bind option and the runtime environment determine the rules for the dynamic SQL attributes.

Related reference

BIND and REBIND options for packages, plans, and services (Db2 Commands)
Special registers (Db2 SQL)

Accessing transition tables in a user-defined function or stored procedure

If you want to refer to the entire set of rows that a triggering SQL statement modifies, rather than to individual rows, use a transition table. You can reference a transition table in user-defined functions and procedures that are invoked from a trigger.

About this task

This topic describes how to access transition variables in a user-defined function, but the same techniques apply to a stored procedure.

To access transition tables in a user-defined function, use table locators, which are pointers to the transition tables. You declare table locators as input parameters in the CREATE FUNCTION statement using the TABLE LIKE table-name AS LOCATOR clause.
Procedure

To access transition tables in a user-defined function or stored procedure:

1. Declare input parameters to receive table locators. You must define each parameter that receives a table locator as an unsigned 4-byte integer.
3. Declare a cursor to access the rows in each transition table.
4. Assign the input parameter values to the table locators.
5. Access rows from the transition tables using the cursors that are declared for the transition tables.

Results

The following examples show how a user-defined function that is written in C, C++, COBOL, or PL/I accesses a transition table for a trigger. The transition table, NEWEMP, contains modified rows of the employee sample table. The trigger is defined like this:

```
CREATE TRIGGER EMPRAISE
AFTER UPDATE ON EMP
REFERENCING NEW TABLE AS NEWEMPS
FOR EACH STATEMENT MODE DB2SQL
BEGIN ATOMIC
VALUES (CHECKEMP(TABLE NEWEMPS));
END;
```

The user-defined function definition looks like this:

```
CREATE FUNCTION CHECKEMP(TABLE LIKE EMP AS LOCATOR)
RETURNS INTEGER
EXTERNAL NAME 'CHECKEMP'
PARAMETER STYLE SQL
LANGUAGE language;
```

Assembler: The following example shows how an assembler program accesses rows of transition table NEWEMPS.

```
CHECKEMP CSECT
SAVE (14,12) ANY SAVE SEQUENCE
LR R12,R15 CODE ADDRESSABILITY
USING CHECKEMP,R12 TELL THE ASSEMBLER
LR R7,R1 SAVE THE PARM POINTER
USING PARMAREA,R7 SET ADDRESSABILITY FOR PARMS
USING SQLDSECT,R8 ESTABLISH ADDRESSABILITY TO SQLDSECT
L R6,PROGSIZE GET SPACE FOR USER PROGRAM
GETMAIN R,LV=(6) GET STORAGE FOR PROGRAM VARIABLES
LR R10,R1 POINT TO THE ACQUIRED STORAGE
LR R2,R10 POINT TO THE FIELD
LR R3,R6 GET ITS LENGTH
SR R4,R4 CLEAR THE INPUT ADDRESS
SR R5,R5 CLEAR THE INPUT LENGTH
MVCL R2,R4 CLEAR OUT THE FIELD
ST R13,FOUR(R10) CHAIN THE SAVEAREA PTRS
ST R10,EIGHT(R13) CHAIN SAVEAREA FORWARD
LR R13,R10 POINT TO THE SAVEAREA
USING PROGAREA,R13 SET ADDRESSABILITY
ST R6,GETLENTH SAVE THE LENGTH OF THE GETMAIN

EXEC SQL DECLARE C1 CURSOR FOR
   SELECT LASTNAME FROM TABLE(:TRIGTBL LIKE EMP)
   WHERE SALARY > 100000

* Declare table locator host variable TRIGTBL
* TRIGTBL SQL TYPE IS TABLE LIKE EMP AS LOCATOR
* Declare a cursor to retrieve rows from the transition table
  * table
* EXEC SQL DECLARE C1 CURSOR FOR
* SELECT LASTNAME FROM TABLE(:TRIGTBL LIKE EMP)
* WHERE SALARY > 100000
```

Chapter 3. Db2 SQL programming 209
L R2,TABLOC GET ADDRESS OF LOCATOR
L R2,0(0,R2) GET LOCATOR VALUE
ST R2,TRIGTB
EXEC SQL OPEN C1
EXEC SQL FETCH C1 INTO :NAME
ENDPROC

C or C++: The following example shows how a C or C++ program accesses rows of transition table NEWEMPS.

```c
int CHECK_EMP(int trig_tbl_id)
{
    /**********************************************************************/
    /* Declare table locator host variable trig_tbl_id */
    /**********************************************************************/
    EXEC SQL BEGIN DECLARE SECTION;
    SQL TYPE IS TABLE LIKE EMP AS LOCATOR trig_tbl_id;
    char name[25];
    EXEC SQL END DECLARE SECTION;
    /**********************************************************************/
    /* Declare a cursor to retrieve rows from the transition table */
    /**********************************************************************/
    EXEC SQL DECLARE C1 CURSOR FOR
    SELECT NAME FROM TABLE(:trig_tbl_id LIKE EMPLOYEE)
    WHERE SALARY > 100000;
    /**********************************************************************/
    /* Fetch a row from transition table */
    /**********************************************************************/
    EXEC SQL OPEN C1;
    EXEC SQL FETCH C1 INTO :name;
    EXEC SQL CLOSE C1;
}
```

COBOL: The following example shows how a COBOL program accesses rows of transition table NEWEMPS.

```cobol
IDENTIFICATION DIVISION.
PROGRAM-ID. CHECKEMP.
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
DATA DIVISION.
WORKING-STORAGE SECTION.
  01  NAME   PIC X(24).
  ...
LINKAGE SECTION.
**********************************************************************
* Declare table locator host variable TRIG-TBL-ID *
**********************************************************************
  01  TRIG-TBL-ID SQL TYPE IS TABLE LIKE EMP AS LOCATOR.
  ...
PROCEDURE DIVISION USING TRIG-TBL-ID.
  **********************************************************************
  * Declare a cursor to retrieve row from transition table *
  **********************************************************************
  EXEC SQL DECLARE C1 CURSOR FOR
  SELECT NAME FROM TABLE(:TRIG-TBL-ID LIKE EMPLOYEE)
  WHERE SALARY > 100000 END-EXEC.
  **********************************************************************
  * Fetch a row from transition table *
  **********************************************************************
...
PL/I: The following example shows how a PL/I program accesses rows of transition table NEWEMPS.

```pli
CHECK_EMP: PROC(TRIG_TBL_ID) RETURNS(BIN FIXED(31))
OPTIONS(MAIN NOEXECOPS REENTRANT);
/**************************************************************/
/* Declare table locator host variable TRIG_TBL_ID */
/**************************************************************/
DECLARE TRIG_TBL_ID SQL TYPE IS TABLE LIKE EMP AS LOCATOR;
DECLARE NAME CHAR(24);

/**************************************************************/
/* Declare a cursor to retrieve rows from the */
/* transition table */
/**************************************************************/
EXEC SQL DECLARE C1 CURSOR FOR
SELECT NAME FROM TABLE(:TRIG_TBL_ID LIKE EMP)
WHERE SALARY > 100000;

/**************************************************************/
/* Retrieve rows from the transition table */
/**************************************************************/
EXEC SQL OPEN C1;
EXEC SQL FETCH C1 INTO :NAME;

EXEC SQL CLOSE C1;
END CHECK_EMP;
```

Preparing an external user-defined function for execution

Because an external user-defined function is written in a programming language, preparing it is similar to the way that you prepare any other application program.

Procedure

To prepare an external user-defined function for execution:

1. Precompile the user-defined function program and bind the DBRM into a package. You need to do this only if your user-defined function contains SQL statements. You do not need to bind a plan for the user-defined function.

2. Compile the user-defined function program and link-edit it with Language Environment and RRSAF.

   You must compile the program with a compiler that supports Language Environment and link-edit the appropriate Language Environment components with the user-defined function. You must also link-edit the user-defined function with RRSAF.

   The program preparation JCL samples DSNHASM, DSNHC, DSNHCPP, DSNHICOB, and DSNHPLI show you how to precompile, compile, and link-edit assembler, C, C++, COBOL, and PL/I Db2 programs. For object-oriented programs in C++, see JCL sample DSNHCPP2 for program preparation hints.

3. For a user-defined function that contains SQL statements, grant EXECUTE authority on the user-defined function package to the function definer.

Abnormal termination of an external user-defined function

If an external user-defined function abnormally terminates, your program receives SQLCODE -430 for invoking the statement.

Db2 also performs the following actions:

- Places the unit of work that contains the invoking statement in a must-rollback state.
- Stops the user-defined function, and subsequent calls fail, in either of the following situations:
– The number of abnormal terminations equals the STOP AFTER \( n \) FAILURES value for the user-defined function.

– If the STOP AFTER \( n \) FAILURES option is not specified, the number of abnormal terminations equals the default MAX ABEND COUNT value for the subsystem.

You should include code in your program to check for a user-defined function abend and to roll back the unit of work that contains the user-defined function invocation.

### Saving information between invocations of a user-defined function by using a scratchpad

If you create a scratchpad for a reentrant user-defined function, Db2 can use it to preserve information between invocations of the function.

### About this task

You can use a scratchpad to save information between invocations of a user-defined function. To indicate that a scratchpad should be allocated when the user-defined function executes, the function definer specifies the SCRATCHPAD parameter in the CREATE FUNCTION statement.

The scratchpad consists of a 4-byte length field, followed by the scratchpad area. The definer can specify the length of the scratchpad area in the CREATE FUNCTION statement. The specified length does not include the length field. The default size is 100 bytes. Db2 initializes the scratchpad for each function to binary zeros at the beginning of execution for each subquery of an SQL statement and does not examine or change the content thereafter. On each invocation of the user-defined function, Db2 passes the scratchpad to the user-defined function. You can therefore use the scratchpad to preserve information between invocations of a reentrant user-defined function.

The following example demonstrates how to enter information in a scratchpad for a user-defined function defined like this:

```sql
CREATE FUNCTION COUNTER()
RETURNS INT
SCRATCHPAD
FENCED
NOT DETERMINISTIC
NO SQL
NO EXTERNAL ACTION
LANGUAGE C
PARAMETER STYLE SQL
EXTERNAL NAME 'UDFCTR';
```

The scratchpad length is not specified, so the scratchpad has the default length of 100 bytes, plus 4 bytes for the length field. The user-defined function increments an integer value and stores it in the scratchpad on each execution.

```c
#include <stdlib.h>
#include <stdio.h>

/* Structure scr defines the passed scratchpad for function ctr */
struct scr {
    long len;
    long countr;
    char not_used[96];
};

// Function ctr: Increments a counter and reports the value from the scratchpad.
void ctr(
    long *out,           /* Output answer (counter) */
    short *outnull,      /* Output null indicator */
    char *sqlstate,      /* SQLSTATE */
    char *funcname,      /* Function name */
    char *specname       /* Specific function name */
) {
    // Increment counter and store it in scratchpad
    countr = countr + 1;
    // Initialize scratchpad
    memset(scr, 0, sizeof(scr));
    // PassScratchpad to function
    // (implementation details)
}
```

The scratchpad length is not specified, so the scratchpad has the default length of 100 bytes, plus 4 bytes for the length field. The user-defined function increments an integer value and stores it in the scratchpad on each execution.
Example of creating and using a user-defined scalar function

You can create a user-defined scalar function that gets input from a table and puts the output in a table.

Suppose that your organization needs a user-defined scalar function that calculates the bonus that each employee receives. All employee data, including salaries, commissions, and bonuses, is kept in the employee table, EMP. The input fields for the bonus calculation function are the values of the SALARY and COMM columns. The output from the function goes into the BONUS column. Because this function gets its input from a Db2 table and puts the output in a Db2 table, a convenient way to manipulate the data is through a user-defined function.

The user-defined function's definer and invoker determine that this new user-defined function should have these characteristics:

- The user-defined function name is CALC_BONUS.
- The two input fields are of type DECIMAL(9,2).
- The output field is of type DECIMAL(9,2).
- The program for the user-defined function is written in COBOL and has a load module name of CBONUS.

Because no built-in function or user-defined function exists on which to build a sourced user-defined function, the function implementer must code an external user-defined function. The implementer performs the following steps:

- Writes the user-defined function, which is a COBOL program
- Precompiles, compiles, and links the program
- Binds a package if the user-defined function contains SQL statements
- Tests the program thoroughly
- Grants execute authority on the user-defined function package to the definer

The user-defined function definer executes this CREATE FUNCTION statement to register CALC_BONUS to Db2:

```
CREATE FUNCTION CALC_BONUS(DECIMAL(9,2),DECIMAL(9,2))
RETURNS DECIMAL(9,2)
EXTERNAL NAME 'CBONUS'
PARAMETER STYLE SQL
LANGUAGE COBOL;
```

The definer then grants execute authority on CALC_BONUS to all invokers.

User-defined function invokers write and prepare application programs that invoke CALC_BONUS. An invoker might write a statement like this, which uses the user-defined function to update the BONUS field in the employee table:

```
UPDATE EMP
SET BONUS = CALC_BONUS(SALARY,COMM);
```

An invoker can execute this statement either statically or dynamically.
User-defined function samples that ship with Db2
To assist you in defining, implementing, and invoking your user-defined functions, Db2 provides a number of sample user-defined functions. All sample user-defined function code is in data set DSN1210.SDSNSAMP.

The following table summarizes the characteristics of the sample user-defined functions.

<table>
<thead>
<tr>
<th>User-defined function name</th>
<th>Language</th>
<th>Member that contains source code</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALTDATE¹</td>
<td>C</td>
<td>DSN8DUAD</td>
<td>Converts the current date to a user-specified format</td>
</tr>
<tr>
<td>ALTDATE²</td>
<td>C</td>
<td>DSN8DUCD</td>
<td>Converts a date from one format to another</td>
</tr>
<tr>
<td>ALTTIME³</td>
<td>C</td>
<td>DSN8DUAT</td>
<td>Converts the current time to a user-specified format</td>
</tr>
<tr>
<td>ALTTIME⁴</td>
<td>C</td>
<td>DSN8DUCT</td>
<td>Converts a time from one format to another</td>
</tr>
<tr>
<td>DAYNAME</td>
<td>C++</td>
<td>DSN8EUDN</td>
<td>Returns the day of the week for a user-specified date</td>
</tr>
<tr>
<td>HDFS_READ</td>
<td>C++</td>
<td>DSN8HDFS</td>
<td>Reads data from a delimiter-separated file in the Hadoop Distributed File System (HDFS)</td>
</tr>
<tr>
<td>JAQL_SUBMIT</td>
<td>C++</td>
<td>DSN8JAQL</td>
<td>Invokes an IBM InfoSphere® BigInsights® Jaql query</td>
</tr>
<tr>
<td>MONTHNAME</td>
<td>C++</td>
<td>DSN8EUMN</td>
<td>Returns the month for a user-specified date</td>
</tr>
<tr>
<td>CURRENCY</td>
<td>C</td>
<td>DSN8DUCY</td>
<td>Formats a floating-point number as a currency value</td>
</tr>
<tr>
<td>TABLE_NAME</td>
<td>C</td>
<td>DSN8DUTI</td>
<td>Returns the unqualified table name for a table, view, or alias</td>
</tr>
<tr>
<td>TABLE_QUALIF</td>
<td>C</td>
<td>DSN8DUTI</td>
<td>Returns the qualifier for a table, view, or alias</td>
</tr>
<tr>
<td>TABLE_LOCATION</td>
<td>C</td>
<td>DSN8DUTI</td>
<td>Returns the location for a table, view, or alias</td>
</tr>
<tr>
<td>WEATHER</td>
<td>C</td>
<td>DSN8DUWF</td>
<td>Returns a table of weather information from an EBCDIC data set</td>
</tr>
</tbody>
</table>

Notes:
1. This version of ALTDATE has one input parameter, of type VARCHAR(13).
2. This version of ALTDATE has three input parameters, of type VARCHAR(17), VARCHAR(13), and VARCHAR(13).
3. This version of ALTTIME has one input parameter, of type VARCHAR(14).
4. This version of ALTTIME has three input parameters, of type VARCHAR(11), VARCHAR(14), and VARCHAR(14).

Member DSN8DUWC contains a client program that shows you how to invoke the WEATHER user-defined table function.
Member DSNTEJBI shows you how to define and prepare the IBM InfoSphere BigInsights sample user-defined functions.
Member DSNTEJ2U shows you how to define and prepare the other sample user-defined functions and the client program.
Creating stored procedures

A stored procedure is executable code that can be called by other programs. The process for creating one depends on the type of procedure.

Before you begin

You must complete some configuration tasks for the Db2 environment before you can use any of the following types of procedures:

- External stored procedures
- Native SQL procedures that satisfy any of the following conditions:
  - Calls at least one external stored procedure, external SQL procedure, or user-defined function.
  - Defined with ALLOW DEBUG MODE or DISALLOW DEBUG MODE.
- External SQL procedures (deprecated)
- Db2-supplied stored procedures

For instructions, see Installation step 21: Configure Db2 for running stored procedures and user-defined functions (Db2 Installation and Migration) or Migration step 23: Configure Db2 for running stored procedures and user-defined functions (optional) (Db2 Installation and Migration).

Procedure

Follow the process for the type of stored procedure that you want to create, and issue a CREATE PROCEDURE statement to register the stored procedure with a database server.

You can create the following types of stored procedures:

Native SQL procedures

The procedure body is written exclusively in SQL statements, including SQL procedural language (SQL PL) statements. The procedure body is contained and specified in the procedure definition along with various attributes of the procedure. A package is generated for a native SQL procedure. It contains the procedure body, including control statements. It might sometimes also include statements generated by Db2. Each time that the procedure is invoked, the package executes one or more times.

All SQL procedures that are created with a CREATE PROCEDURE statement that does not specify the FENCED or EXTERNAL options are native SQL procedures. More capabilities are supported for native SQL procedures, they usually perform better than external SQL procedures, and no associated C program is generated for them.

For more information, see “Creating native SQL procedures” on page 231.

External stored procedures

The procedure body is an external program that is written in a programming language such as C, C++, COBOL, or Java and it can contain SQL statements. The source code for an external stored procedure is separate from the procedure definition and is bound into a package. The name of the external executable is specified as part of the procedure definition along with various attributes of the procedure. All programs must be designed to run using Language Environment. Your COBOL and C++ stored procedures can contain object-oriented extensions. Each time that the stored procedure is invoked, the logic in the procedure controls whether the package executes and how many times.

For more information, see “Creating external stored procedures” on page 257.
External SQL procedures (deprecated)

The procedure body is written exclusively in SQL statements, including SQL procedural language (SQL PL) statements. The procedure body is specified in the procedure definition along with various attributes of the procedure. A C program and an associated package are generated for an external SQL procedure. It contains the procedure body, including control statements. It might sometimes also include statements generated by Db2. Each time that the procedure is invoked, the package executes one or more times.

Native SQL procedures are more fully supported, easier to maintain, and typically perform better than external SQL procedures, which are deprecated.

For more information, see “Creating external SQL procedures (deprecated)” on page 291.

Related concepts

Stored procedures

A stored procedure is a compiled program that can execute SQL statements and is stored at a local or remote Db2 server. You can invoke a stored procedure from an application program or from the command line processor. A single call to a stored procedure from a client application can access the database at the server several times.

External stored procedures

An external stored procedure is a procedure that is written in a host language and can contain SQL statements. The source code for external procedures is separate from the definition.

SQL procedures

An SQL procedure is a stored procedure that contains only SQL statements.

Related tasks

Obfuscating source code of SQL procedures, SQL functions, and triggers (Db2 Administration Guide)

Related reference

CREATE PROCEDURE (Db2 SQL)

Db2 for z/OS Exchange

Related information

Db2 for z/OS Stored Procedures: Through the CALL and Beyond (IBM Redbooks)

Stored procedures

A stored procedure is a compiled program that can execute SQL statements and is stored at a local or remote Db2 server. You can invoke a stored procedure from an application program or from the command line processor. A single call to a stored procedure from a client application can access the database at the server several times.

A typical stored procedure contains two or more SQL statements and some manipulative or logical processing in a host language or SQL procedure statements. You can call stored procedures from other applications or from the command line. Db2 provides some stored procedures, but you can also create your own.

A stored procedure provides a common piece of code that is written only once and is maintained in a single instance that can be called from several different applications. Host languages can easily call procedures that exist on a local system, and SQL can call stored procedures that exist on remote systems. In fact, a major benefit of procedures in SQL is that they can be used to enhance the performance characteristics of distributed applications. With stored procedures, you can avoid network transfer of large amounts of data obtained as part of intermediate results in a long sequence of queries.

The following diagram illustrates the processing for an application that does not use stored procedures. The client application embeds SQL statements and communicates with the server separately for each statement. This application design results in increased network traffic and processor costs.
Figure 9. Processing without stored procedures

The following diagram illustrates the processing for an application that uses stored procedures. Because a stored procedure is used on the server, a series of SQL statements can be executed with a single send and receive operation, reducing network traffic and the cost of processing these statements.

Figure 10. Processing with stored procedures

Stored procedures are useful for client/server applications that do at least one of the following things:

- Execute multiple remote SQL statements. Remote SQL statements can create many network send and receive operations, which results in increased processor costs. Stored procedures can encapsulate many of your application's SQL statements into a single message to the Db2 server, reducing network traffic to a single send and receive operation for a series of SQL statements. Locks on Db2 tables are not held across network transmissions, which reduces contention for resources at the server.

- Access tables from a dynamic SQL environment where table privileges for the application that is running are undesirable. Stored procedures allow static SQL authorization from a dynamic environment.

- Access host variables for which you want to guarantee security and integrity. Stored procedures remove SQL applications from the workstation, which prevents workstation users from manipulating the contents of sensitive SQL statements and host variables.

- Create a result set of rows to return to the client application.

Stored procedures that are written in embedded static SQL provide the following additional advantages:
Better performance because static SQL is prepared at precompile time and has no run time overhead for access plan (package) generation.

Encapsulation enables programmers to write applications that access data without knowing the details of database objects.

Improved security because access privileges are encapsulated within the packages that are associated with the stored procedures. You can grant access to run a stored procedure that selects data from tables, without granting SELECT privilege to the user.

You can create the following types of stored procedures:

**Native SQL procedures**

The procedure body is written exclusively in SQL statements, including SQL procedural language (SQL PL) statements. The procedure body is contained and specified in the procedure definition along with various attributes of the procedure. A package is generated for a native SQL procedure. It contains the procedure body, including control statements. It might sometimes also include statements generated by Db2. Each time that the procedure is invoked, the package executes one or more times.

All SQL procedures that are created with a CREATE PROCEDURE statement that does not specify the FENCED or EXTERNAL options are native SQL procedures. More capabilities are supported for native SQL procedures, they usually perform better than external SQL procedures, and no associated C program is generated for them.

For more information, see “Creating native SQL procedures” on page 231.

**External stored procedures**

The procedure body is an external program that is written in a programming language such as C, C++, COBOL, or Java and it can contain SQL statements. The source code for an external stored procedure is separate from the procedure definition and is bound into a package. The name of the external executable is specified as part of the procedure definition along with various attributes of the procedure. All programs must be designed to run using Language Environment. Your COBOL and C++ stored procedures can contain object-oriented extensions. Each time that the stored procedure is invoked, the logic in the procedure controls whether the package executes and how many times.

For more information, see “Creating external stored procedures” on page 257.

**External SQL procedures (deprecated)**

The procedure body is written exclusively in SQL statements, including SQL procedural language (SQL PL) statements. The procedure body is specified in the procedure definition along with various attributes of the procedure. A C program and an associated package are generated for an external SQL procedure. It contains the procedure body, including control statements. It might sometimes also include statements generated by Db2. Each time that the procedure is invoked, the package executes one or more times.

Native SQL procedures are more fully supported, easier to maintain, and typically perform better than external SQL procedures, which are deprecated.

For more information, see “Creating external SQL procedures (deprecated)” on page 291.

Db2 also provides a set of stored procedures that you can call in your application programs to perform a number of utility, application programming, and performance management functions. These procedures are called supplied stored procedures. Typically, you create these procedures during installation or migration.

**Difference between native SQL and external procedures**

**Related concepts**

Common SQL API stored procedures (Db2 Administration Guide)

**Related tasks**

Implementing Db2 stored procedures (Stored procedures provided by Db2)

**Related reference**

Procedures that are supplied with Db2 (Db2 SQL)
**Stored procedure parameters**

You can pass information between a stored procedure and the calling application program by using parameters. Applications pass the required parameters in the SQL CALL statement. Optionally, the application can also include an indicator variable with each parameter to allow for null values or to pass large output parameter values.

You define the stored procedure parameters as part of the stored procedure definition in the CREATE PROCEDURE statement. The stored procedure parameters can be one of the following types:

**IN**
- Input-only parameters, which provide values to the stored procedure.

**OUT**
- Output-only parameters, which return values from the stored procedure to the calling program.

**INOUT**
- Input and output parameters, which provide values to and return values from the stored procedure.

If a stored procedure fails to set one or more of the OUT or INOUT parameters, Db2 does not return an error. Instead, Db2 returns the output parameters to the calling program, with the values that were established on entry to the stored procedure.

Within a procedure body, the following rules apply to IN, OUT, and INOUT parameters:

- You can use a parameter that you define as IN on the left side or right side of an assignment statement. However, if you assign a value to an IN parameter, you cannot pass the new value back to the caller. The IN parameter has the same value before and after the SQL procedure is called.

- You can use a parameter that you define as OUT on the left side or right side of an assignment statement. The last value that you assign to the parameter is the value that is returned to the caller. The starting value of an OUT parameter is NULL.

- You can use a parameter that you define as INOUT on the left side or right side of an assignment statement. The caller determines the first value of the INOUT parameter, and the last value that you assign to the parameter is the value that is returned to the caller.

**Restrictions:**
- You cannot pass file reference variables as stored procedure parameters.
- You cannot pass parameters with the type XML to stored procedures. You can specify tables or views that contain XML columns as table locator parameters. However, you cannot reference the XML columns in the body of the stored procedure.

**Related tasks**
- Calling a stored procedure from your application
  - To run a stored procedure, you can either call it from a client program or invoke it from the command line processor.
  - Passing large output parameters to stored procedures by using indicator variables
    - If any output parameters occupy a large amount of storage, passing the entire storage area to a stored procedure can degrade performance.

**Related reference**
- CALL (Db2 SQL)
- CREATE PROCEDURE (Db2 SQL)

**Example of a simple stored procedure**

When an application that runs on a workstation calls a stored procedure on a Db2 server, the stored procedure updates a table based on the information that it receives from the application.

Suppose that an application runs on a workstation client and calls a stored procedure A on the Db2 server at location LOCA. Stored procedure A performs the following operations:

1. Receives a set of parameters containing the data for one row of the employee to project activity table (DSN8C10.EMPPROJACT). These parameters are input parameters in the SQL statement CALL:
• EMP: employee number
• PRJ: project number
• ACT: activity ID
• EMT: percent of employee's time required
• EMS: date the activity starts
• EME: date the activity is due to end

2. Declares a cursor, C1, with the option WITH RETURN, that is used to return a result set containing all rows in EMPPROJACT to the workstation application that called the stored procedure.

3. Queries table EMPPROJACT to determine whether a row exists where columns PROJNO, ACTNO, EMSTDATE, and EMPNO match the values of parameters PRJ, ACT, EMS, and EMP. (The table has a unique index on those columns. There is at most one row with those values.)

4. If the row exists, executes an SQL statement UPDATE to assign the values of parameters EMT and EME to columns EMPTIME and EMENDATE.¹

5. If the row does not exist (SQLCODE +100), executes an SQL statement INSERT to insert a new row with all the values in the parameter list.¹

6. Opens cursor C1. This causes the result set to be returned to the caller when the stored procedure ends.

7. Returns two parameters, containing these values:
   • A code to identify the type of SQL statement last executed: UPDATE or INSERT.
   • The SQLCODE from that statement.

Note:
1. Alternatively, steps 4 and 5 can be accomplished with a single MERGE statement.

The following figure illustrates the steps that are involved in executing this stored procedure.
Figure 11. Stored procedure overview

Notes:

1. The workstation application uses the SQL CONNECT statement to create a conversation with Db2.
2. Db2 creates a Db2 thread to process SQL requests.
3. The SQL statement CALL tells the Db2 server that the application is going to run a stored procedure. The calling application provides the necessary parameters.
4. The plan for the client application contains information from catalog table SYSIBM.SYSROUTINES about stored procedure A.
5. Db2 passes information about the request to the stored procedures address space, and the stored procedure begins execution.

6. The stored procedure executes SQL statements.

   Db2 verifies that the owner of the package or plan containing the SQL statement CALL has EXECUTE authority for the package associated with the Db2 stored procedure.

   One of the SQL statements opens a cursor that has been declared WITH RETURN. This causes a result set to be returned to the workstation application when the procedure ends.

   Any SQLCODE that is issued within an external stored procedure is not returned to the workstation application in the SQLCA (as the result of the CALL statement).

7. If an error is not encountered, the stored procedure assigns values to the output parameters and exits.

   Control returns to the Db2 stored procedures address space, and from there to the Db2 system. If the stored procedure definition contains COMMIT ON RETURN NO, Db2 does not commit or roll back any changes from the SQL in the stored procedure until the calling program executes an explicit COMMIT or ROLLBACK statement. If the stored procedure definition contains COMMIT ON RETURN YES, and the stored procedure executed successfully, Db2 commits all changes. The COMMIT statement closes the cursor unless it is declared with the WITH HOLD option.

8. Control returns to the calling application, which receives the output parameters and the result set. Db2 then:

   • Closes all cursors that the stored procedure opened, except those that the stored procedure opened to return result sets.

   • Discards all SQL statements that the stored procedure prepared.

   • Reclaims the working storage that the stored procedure used.

   The application can call more stored procedures, or it can execute more SQL statements. Db2 receives and processes the COMMIT or ROLLBACK request. The COMMIT or ROLLBACK operation covers all SQL operations, whether executed by the application or by stored procedures, for that unit of work.

   If the application involves IMS or CICS, similar processing occurs based on the IMS or CICS sync point rather than on an SQL COMMIT or ROLLBACK statement.

9. Db2 returns a reply message to the application describing the outcome of the COMMIT or ROLLBACK operation.

10. The workstation application executes the following steps to retrieve the contents of table EMMPPROJACT, which the stored procedure has returned in a result set:

    a. Declares a result set locator for the result set being returned.

    b. Executes the ASSOCIATE LOCATORS statement to associate the result set locator with the result set.

    c. Executes the ALLOCATE CURSOR statement to associate a cursor with the result set.

    d. Executes the FETCH statement with the allocated cursor multiple times to retrieve the rows in the result set.

    e. Executes the CLOSE statement to close the cursor.

**SQL procedures**

An SQL procedure is a stored procedure that contains only SQL statements.

The source code for these procedures (the SQL statements) is specified in CREATE PROCEDURE statement. The part of the CREATE PROCEDURE statement that contains SQL statements is called the *procedure body*. 
Types of SQL procedures

Db2 for z/OS supports the following types of SQL procedures:

Native SQL procedures

The procedure body is written exclusively in SQL statements, including SQL procedural language (SQL PL) statements. The procedure body is contained and specified in the procedure definition along with various attributes of the procedure. A package is generated for a native SQL procedure. It contains the procedure body, including control statements. It might sometimes also include statements generated by Db2. Each time that the procedure is invoked, the package executes one or more times.

All SQL procedures that are created with a CREATE PROCEDURE statement that does not specify the FENCED or EXTERNAL options are native SQL procedures. More capabilities are supported for native SQL procedures, they usually perform better than external SQL procedures, and no associated C program is generated for them.

For more information, see “Creating native SQL procedures” on page 231.

External SQL procedures (deprecated)

The procedure body is written exclusively in SQL statements, including SQL procedural language (SQL PL) statements. The procedure body is specified in the procedure definition along with various attributes of the procedure. A C program and an associated package are generated for an external SQL procedure. It contains the procedure body, including control statements. It might sometimes also include statements generated by Db2. Each time that the procedure is invoked, the package executes one or more times.

Native SQL procedures are more fully supported, easier to maintain, and typically perform better than external SQL procedures, which are deprecated.

For more information, see “Creating external SQL procedures (deprecated)” on page 291.

Native SQL procedures

A native SQL procedure is a procedure whose body is written entirely in SQL. The body is written in the SQL procedural language (SQL PL). A native SQL procedure is created by issuing a single SQL statement, CREATE PROCEDURE. Native SQL procedures do not require any other program preparation, such as precompiling, compiling, or link-editing source code. Native SQL procedures are executed as SQL statements that are bound in a Db2 package. Native SQL procedures do not have an associated external application program. Native SQL procedures are more fully supported, easier to maintain, and typically perform better than external SQL procedures, which are deprecated.

Native SQL procedures have the following advantages:

• You can create them in one step.
• They do not run in a WLM environment.
• They might be eligible for zIIP redirect if they are invoked remotely through a DRDA client.
• They usually perform better than external SQL procedures.
• They support more capabilities, such as nested compound statements, than external SQL procedures.
• Db2 can manage multiple versions of these procedures for you.
• You can specify that the SQL procedure commits autonomously, without committing the work of the calling application.

All SQL procedures that are created without the FENCED or EXTERNAL options in the CREATE PROCEDURE statement are native SQL procedures.

External SQL procedures (deprecated)

An external SQL procedure is a procedure whose body is written entirely in SQL. The body is written in the SQL procedural language (SQL PL). However, an external SQL procedure is created, implemented, and executed like other external stored procedures.
Deprecated function: External SQL procedures are deprecated and not as fully supported as native SQL procedures. For best results, create native SQL procedures instead. For more information, see “Creating native SQL procedures” on page 231 and “Migrating an external SQL procedure to a native SQL procedure” on page 292.

All SQL procedures that were created prior to DB2 9 are external SQL procedures. Starting in Version DB2 9, you can create an external SQL procedure by specifying FENCED or EXTERNAL in the CREATE PROCEDURE statement.

SQL procedure body
The body of an SQL procedure contains one or more SQL statements. In the SQL procedure body, you can also declare and use variables, conditions, return codes, statements, cursors, and handlers.

Statements that you can include in an SQL procedure body
A CREATE PROCEDURE statement for a native SQL procedure contains an SQL-routine-body, as defined in CREATE PROCEDURE (SQL - native) (Db2 SQL). The syntax diagram for SQL-routine-body defines the procedure body as a single SQL statement. The SQL statement can be one of the SQL statements that are shown in the syntax diagram for SQL-routine-body, or an SQL control statement. The syntax diagram for SQL-control-statement in SQL procedural language (SQL PL) (Db2 SQL) identifies the control statements that can be specified.

A native SQL procedure can contain multiple SQL statements if the outermost SQL statement is an SQL-control-statement that includes other SQL statements. These statements are defined as SQL procedure statements. The syntax diagram in SQL-procedure-statement (Db2 SQL) identifies the SQL statements that can be specified within a control statement. The syntax notes for SQL-procedure-statement clarify the SQL statements that are allowed in a native SQL procedure.

Examples
The following examples show how to determine whether an SQL statement is allowed in an SQL procedure.

The syntax diagrams for the control statements indicate where semicolons are needed in an SQL procedure. If the procedure contains a single statement that is not a control statement, such as Example 1, then no semicolons are in the CREATE PROCEDURE statement. If the procedure consists of multiple statements, such as Example 2, use semicolons to separate SQL statements within the SQL procedure. Do not put a semicolon after the outermost control statement.

Example 1

```
CREATE PROCEDURE UPDATE_SALARY_1
  (IN EMPLOYEE_NUMBER CHAR(10),
   IN RATE DECIMAL(6,2))
LANGUAGE SQL
MODIFIES SQL DATA
DETERMINISTIC
COMMIT ON RETURN YES
UPDATE EMP A
SET SALARY = SALARY * RATE
WHERE EMPNO = EMPLOYEE_NUMBER
```

The UPDATE statement (A) is an SQL statement that is allowed because it is listed in the syntax diagram for SQL-routine-body.

Example 2

```
CREATE PROCEDURE GETWEEKENDS(IN MYDATES DATEARRAY, OUT WEEKENDS DATEARRAY)
BEGIN
  -- ARRAY INDEX VARIABLES
  DECLARE DATEINDEX, WEEKENDINDEX INT DEFAULT 1;
  -- VARIABLE TO STORE THE ARRAY LENGTH OF MYDATES,
  -- INITIALIZED USING THE CARDINALITY FUNCTION.
  DECLARE DATESCOUNT INT;
  -- VARIABLE TO STORE WEEKENDS
  DECLARE WEEKENDS DATEARRAY;
  -- VARIABLE TO STORE EACH WEEKEND DATE
  DECLARE WEEKEND DATE;
  
  WHILE DATEINDEX <= DATESCOUNT DO
    IF MYDATES[DATEINDEX] = WEEKEND INDEX THEN
      WEEKENDS[WEEKENDINDEX] = MYDATES[DATEINDEX];
      WEEKENDINDEX = WEEKENDINDEX + 1;
    END IF;
    DATEINDEX = DATEINDEX + 1;
  END WHILE;
```

The WHILE statement is an SQL statement that is allowed because it is listed in the syntax diagram for SQL-routine-body.
The SQL procedure has the following keywords and statements:

- The BEGIN and END keywords (A) indicate the beginning and the end of a compound statement.
- The DECLARE statements (B) are components of a compound statement, and define SQL variables within the compound statement.
- The SET assignment statements (C) are SQL control statements that assign values to SQL variables.
- The WHILE statement (D) and the IF statement (E) are SQL control statements.

A compound statement is an SQL control statement. SQL control statements are allowed in the SQL procedure body because SQL-control-statement is listed in the syntax diagram for SQL-routine-body of a CREATE PROCEDURE (SQL-native) statement.

**Related concepts**

Nested compound statements in native SQL procedures

*Nested compound statements* are blocks of SQL statements that are contained by other blocks of SQL statements in native SQL procedures. Use nested compound statements to define condition handlers that execute more than one statement and to define different scopes for variables and condition handlers.

**Store procedure parameters**

You can pass information between a stored procedure and the calling application program by using parameters. Applications pass the required parameters in the SQL CALL statement. Optionally, the application can also include an indicator variable with each parameter to allow for null values or to pass large output parameter values.

**Promotion of data types (Db2 SQL)**

**Related reference**

compound-statement (Db2 SQL)

**Variables in SQL procedures**

For data that you use only within an SQL procedure, you can declare *SQL variables* and store the values in the variables. SQL variables are similar to host variables in external stored procedures. SQL variables can be defined within the same data types and lengths as SQL procedure parameters.

An SQL variable declaration has the following form:

```sql
DECLARE SQL-variable-name data-type;
```

An SQL variable is defined in a compound statement. SQL variables can be referenced anywhere in the compound statement in which they are declared, including any SQL statement that is directly or indirectly nested within that compound statement. For more information, see References to SQL parameters and variables (Db2 SQL).

You can perform any operations on SQL variables that you can perform on host variables in SQL statements.

**Related concepts**

Host variables

Use host variables to pass a single data item between Db2 and your application.

Using host variables in SQL statements
Use scalar host variables in embedded SQL statements to represent a single value. Host variables are useful for storing retrieved data or for passing values that are to be assigned or used for comparisons.

Related tasks
Controlling the scope of variables in an SQL procedure
Use nested compound statements within an SQL procedure to define the scope of SQL variables. You can reference the variable only within the compound statement in which it was declared and within any nested statements.

Examples of SQL procedures
You can use CASE statements, compound statements, and nested statements within an SQL procedure body.

Example: CASE statement: The following SQL procedure demonstrates how to use a CASE statement. The procedure receives an employee's ID number and rating as input parameters. The CASE statement modifies the employee's salary and bonus, using a different UPDATE statement for each of the possible ratings.

```
CREATE PROCEDURE UPDATESALARY2
    (IN EMPNUMBR CHAR(6),
     IN RATING INT)
    LANGUAGE SQL
    MODIFIES SQL DATA
    CASE RATING
      WHEN 1 THEN
        UPDATE CORPDATA.EMPLOYEE
        SET SALARY = SALARY * 1.10, BONUS = 1000
        WHERE EMPNO = EMPNUMBR;
      WHEN 2 THEN
        UPDATE CORPDATA.EMPLOYEE
        SET SALARY = SALARY * 1.05, BONUS = 500
        WHERE EMPNO = EMPNUMBR;
      ELSE
        UPDATE CORPDATA.EMPLOYEE
        SET SALARY = SALARY * 1.03, BONUS = 0
        WHERE EMPNO = EMPNUMBR;
    END CASE
```

Example: Compound statement with nested IF and WHILE statements: The following example shows a compound statement that includes an IF statement, a WHILE statement, and assignment statements. The example also shows how to declare SQL variables, cursors, and handlers for classes of error codes.

The procedure receives a department number as an input parameter. A WHILE statement in the procedure body fetches the salary and bonus for each employee in the department, and uses an SQL variable to calculate a running total of employee salaries for the department. An IF statement within the WHILE statement tests for positive bonuses and increments an SQL variable that counts the number of bonuses in the department. When all employee records in the department have been processed, a NOT FOUND condition occurs. A NOT FOUND condition handler makes the search condition for the WHILE statement false, so execution of the WHILE statement ends. Assignment statements then assign the total employee salaries and the number of bonuses for the department to the output parameters for the stored procedure.

If any SQL statement in the compound statement P1 receives an error, the SQLEXCEPTION handler receives control. The handler action sets the output parameter DEPTSALARY to NULL. After the handler action has completed successfully, the original error condition is resolved (SQLSTATE '00000', SQLCODE 0). Because this handler is an EXIT handler, execution passes to the end of the compound statement, and the SQL procedure ends.

```
CREATE PROCEDURE RETURNDEPTSALARY
    (IN DEPTNUMBER CHAR(3),
     OUT DEPTSALARY DECIMAL(15,2),
     OUT DEPTBONUSCNT INT)
    LANGUAGE SQL
    READS SQL DATA
    P1: BEGIN
      DECLARE EMPLOYEE_SALARY DECIMAL(9,2);
      DECLARE EMPLOYEE_BONUS DECIMAL(9,2);
      DECLARE TOTAL_SALARY DECIMAL(15,2) DEFAULT 0;
```
DECLARE BONUS_CNT INT DEFAULT 0;
DECLARE END_TABLE INT DEFAULT 0;
DECLARE C1 CURSOR FOR
   SELECT SALARY, BONUS FROM CORPDATA.EMPLOYEE
   WHERE WORKDEPT = DEPTNUMBER;
DECLARE CONTINUE HANDLER FOR NOT FOUND
   SET END_TABLE = 1;
DECLARE EXIT HANDLER FOR SQLEXCEPTION
   SET DEPTSALARY = NULL;
OPEN C1;
FETCH C1 INTO EMPLOYEE_SALARY, EMPLOYEE_BONUS;
WHILE END_TABLE = 0 DO
   SET TOTAL_SALARY = TOTAL_SALARY + EMPLOYEE_SALARY + EMPLOYEE_BONUS;
   IF EMPLOYEE_BONUS > 0 THEN
      SET BONUS_CNT = BONUS_CNT + 1;
   END IF;
   FETCH C1 INTO EMPLOYEE_SALARY, EMPLOYEE_BONUS;
END WHILE;
CLOSE C1;
SET DEPTSALARY = TOTAL_SALARY;
SET DEPTBONUSCNT = BONUS_CNT;
END P1

Example: Compound statement with dynamic SQL statements: The following example shows a compound statement that includes dynamic SQL statements.

The procedure receives a department number (P_DEPT) as an input parameter. In the compound statement, three statement strings are built, prepared, and executed:

- The first statement string executes a DROP statement to ensure that the table to be created does not already exist. The table name is the concatenation of the TABLE_PREFIX constant value, the P_DEPT parameter value, and the TABLE_SUFFIX constant value.
- The next statement string executes a CREATE statement to create DEPT_deptno_T.
- The third statement string inserts rows for employees in department deptno into DEPT_deptno_T.

Just as statement strings that are prepared in host language programs cannot contain host variables, statement strings in SQL procedures cannot contain SQL variables or stored procedure parameters. Therefore, the third statement string contains a parameter marker that represents P_DEPT. When the prepared statement is executed, parameter P_DEPT is substituted for the parameter marker.
**Autonomous procedures**

Autonomous procedures execute under their own units of work, separate from the calling program, and commit when they finish without committing the work of the calling program.

**Autonomous procedures** execute as separate units of work that are independent from the calling application programs. Autonomous procedures follow the rules of the COMMIT ON RETURN YES option for their changes before returning to the caller. However, their commit does not impact changes completed by the calling application program. The calling application program controls when its own updates are committed or rolled back.

If the calling application rolls back its own changes, the committed changes of the autonomous procedure are not affected. Therefore, autonomous procedures are useful for logging information about error conditions encountered by an application program. When the application encounters the error and rolls back its own changes, the committed changes of the autonomous procedure remain available.

Autonomous procedures can be called by normal application programs, other stored procedures, user-defined functions or triggers. Autonomous procedures can complete the following types of work:

- Execute SQL statements
- Invoke another procedure, function, or trigger, as long as the number of nested levels does not exceed 64, and the called procedure is not autonomous.
- Execute COMMIT and ROLLBACK statements that apply to the SQL operations executed by nested processes within the autonomous procedure.

The following restrictions apply to autonomous procedures:

- Only native SQL procedures can be defined as autonomous.
- Autonomous procedures and nested procedure, triggers, and functions within autonomous procedures cannot invoke other autonomous procedures.
- Autonomous procedures cannot see uncommitted changes from the calling application.
- When multiple versions of a procedure exist, all versions must be defined as autonomous.
- Autonomous procedures do not share locks with the calling application, meaning that the autonomous procedure might timeouts because of lock contention with the calling application.
- Parallelism is disabled for autonomous procedures. All statements in an autonomous procedure and for any nested levels within are run in sequential processing mode.
- DYNAMIC RESULT SETS 0 must be specified when autonomous procedures are used.
- Stored procedure parameters must not be defined as a LOB data type, or any distinct data type that is based on a LOB or XML value.

**Related tasks**

Controlling autonomous procedures (Db2 Administration Guide)

**External stored procedures**

An external stored procedure is a procedure that is written in a host language and can contain SQL statements. The source code for external procedures is separate from the definition.

An external stored procedure is much like any other SQL application. It can include static or dynamic SQL statements, IFI calls, and Db2 commands that are issued through IFI. You prepare external stored procedures as you would normally prepare application programs. You precompile, compile, and link-edit them. Then, you bind the DBRM into a package. You also need to define the procedure to Db2 by using the CREATE PROCEDURE statement. Thus, the source code for an external stored procedure is separate from the definition for the stored procedure.

**Language requirements for the external stored procedure and its caller**

You can write an external stored procedure in Assembler, C, C++, COBOL, Java, REXX, or PL/I. All programs must be designed to run using Language Environment. Your COBOL and C++ stored procedures can contain object-oriented extensions.
The program that calls the stored procedure can be in any language that supports the SQL CALL statement. ODBC applications can use an escape clause to pass a stored procedure call to Db2.

**Related concepts**

Object-oriented extensions in COBOL

When you use object-oriented extensions in a COBOL application, you need to consider where to place SQL statements, the SQLCA, the SQLDA, and host variable declarations. You also need to consider the rules for host variables.

REXX stored procedures

A REXX stored procedure is similar to any other REXX procedure and follows the same rules as stored procedures in other languages. A REXX stored procedure receives input parameters, executes REXX commands, optionally executes SQL statements, and returns at most one output parameter. However, a few differences exist.

Java stored procedures and user-defined functions (Db2 Application Programming for Java)

**Differences between native SQL procedures and external procedures**

SQL procedures are written entirely in SQL statements. External procedures are written in a host language and can contain SQL statements. You can invoke both types of procedures with an SQL CALL statement. However, you should consider several important differences in behavior and preparation.

Native SQL procedures and external procedures differ in the following ways:

**How they handles errors**

- For an SQL procedure, Db2 automatically returns SQL conditions in the SQLCA when the procedure does not include a RETURN statement or a handler. For information about the various ways to handle errors in an SQL procedure, see “Handling SQL conditions in an SQL procedure” on page 237.

- For an external stored procedure, Db2 does not return SQL conditions in the SQLCA to the invoking application. If you use PARAMETER STYLE SQL when you define an external procedure, you can set SQLSTATE to indicate an error before the procedure ends. For valid SQLSTATE values, see SQLSTATE values and common error codes (Db2 Codes).

**How they specify the code for the stored procedure**

SQL procedure definitions contain the source code for the stored procedure. An external stored procedure definition specifies the name of the stored procedure program.

**How you define the stored procedure.**

For both native SQL procedures and external procedures, you define the stored procedure to Db2 by executing the CREATE PROCEDURE statement. For external procedures, you must also separately bind the source code for procedure into a package. You can do this before or after you issue the CREATE PROCEDURE statement to define the external procedure.

**Examples**

**Creating a native SQL procedure**

The following example shows a definition for an SQL procedure.

```
CREATE PROCEDURE UPDATESALARY1
           (IN EMPNUMBR CHAR(10),
            IN RATE DECIMAL(6,2))
           LANGUAGE SQL
           UPDATE EMP
           SET SALARY = SALARY * RATE
           WHERE EMPNO = EMPNUMBR
```

**Notes:**

1. The stored procedure name is UPDATESALARY1.

2. The two parameters have data types of CHAR(10) and DECIMAL(6,2). Both are input parameters.
3 LANGUAGE SQL indicates that this is an SQL procedure, so a procedure body follows the other parameters.

4 The procedure body consists of a single SQL UPDATE statement, which updates rows in the employee table.

Creating an external stored procedure
The following example shows a definition for an equivalent external stored procedure that is written in COBOL. The stored procedure program, which updates employee salaries, is called UPDSAL.

```sql
CREATE PROCEDURE UPDATESALARY1
(IN EMPNUMBR CHAR(10),
IN RATE DECIMAL(6,2))
LANGUAGE COBOL
EXTERNAL NAME UPDSAL;
```

Notes:

1 The stored procedure name is UPDATESALARY1.

2 The two parameters have data types of CHAR(10) and DECIMAL(6,2). Both are input parameters.

3 LANGUAGE COBOL indicates that this is an external procedure, so the code for the stored procedure is in a separate, COBOL program.

4 The name of the load module that contains the executable stored procedure program is UPDSAL.

Related reference
CREATE PROCEDURE (Db2 SQL)
CREATE PROCEDURE (external) (Db2 SQL)
CREATE PROCEDURE (SQL - native) (Db2 SQL)

**COMMIT and ROLLBACK statements in a stored procedure**
When you issue COMMIT or ROLLBACK statements in your stored procedure, Db2 commits or rolls back all changes within the unit of work. For procedures that are not defined as autonomous, the committed or rolled back changes include changes that the client application made before it called the stored procedure and Db2 work that the stored procedure does. For autonomous procedures, the committed or rolled back changes include only work done by the stored procedure’s unit of work.

If your stored procedure includes COMMIT or ROLLBACK statements, define it with one of the following clauses:

- CONTAINS SQL
- READS SQL DATA
- MODIFIES SQL DATA

The COMMIT ON RETURN clause in a stored procedure definition has no effect on the COMMIT or ROLLBACK statements in the stored procedure code. If you specify COMMIT ON RETURN YES when you define the stored procedure, Db2 issues a COMMIT statement when control returns from the stored procedure. This action occurs regardless of whether the stored procedure contains COMMIT or ROLLBACK statements.

If you specify AUTONOMOUS when you define the stored procedure, the autonomous procedure is a separate unit of work from the calling application. Db2 issues a COMMIT statement when control returns from the stored procedure, but only changes completed by the autonomous procedure are committed. Similarly, COMMIT or ROLLBACK statements in the autonomous procedure code also have no effect on work done by the calling application.
A ROLLBACK statement has the same effect on cursors in a stored procedure as it has on cursors in stand-alone programs. A ROLLBACK statement closes all open cursors. A COMMIT statement in a stored procedure closes cursors that are not declared WITH HOLD and leaves open those cursors that are declared WITH HOLD. The effect of COMMIT or ROLLBACK on cursors applies to cursors that are declared in the calling application and to cursors that are declared in the stored procedure.

Restriction: You cannot include COMMIT or ROLLBACK statements in a stored procedure if any of the following conditions are true:

• The stored procedure is nested within a trigger or user-defined function.
• The stored procedure is called by a client that uses two-phase commit processing.
• The client program uses a type 2 connection to connect to the remote server that contains the stored procedure.
• Db2 is not the commit coordinator.

If a COMMIT or ROLLBACK statement in a stored procedure violates any of these conditions, Db2 puts the transaction in a must-rollback state. Also, in this case, the CALL statement fails.

Related reference
CALL (Db2 SQL)
COMMIT (Db2 SQL)
ROLLBACK (Db2 SQL)

Special registers in a stored procedure
You can use all special registers in a stored procedure. However, you can modify only some of those special registers. After a stored procedure completes, Db2 restores all special registers to the values that they had before invocation.

Creating native SQL procedures
A native SQL procedure is a procedure whose body is written entirely in SQL and is created by issuing a single SQL statement, CREATE PROCEDURE.

Before you begin
Before you create a native SQL procedure, Configure Db2 for running stored procedures and user-defined functions during installation or Configure Db2 for running stored procedures and user-defined functions during migration if the native SQL procedure satisfies at least one of the following conditions:

• The native SQL procedure calls at least one external stored procedure, external SQL procedure, or user-defined function.
• The native SQL procedure is defined with ALLOW DEBUG MODE or DISALLOW DEBUG MODE. If you specify DISABLE DEBUG MODE, you do not need to set up the stored procedure environment.

About this task
A native SQL procedure is a procedure whose body is written entirely in SQL. The body is written in the SQL procedural language (SQL PL). A native SQL procedure is created by issuing a single SQL statement, CREATE PROCEDURE. Native SQL procedures do not require any other program preparation, such as precompiling, compiling, or link-editing source code. Native SQL procedures are executed as SQL statements that are bound in a Db2 package. Native SQL procedures do not have an associated external application program. Native SQL procedures are more fully supported, easier to maintain, and typically perform better than external SQL procedures, which are deprecated.

Procedure
To create a native SQL procedure, perform one of the following actions:
• Use IBM Data Studio to specify the source statements for the SQL procedure and deploy the SQL procedure to Db2.

IBM Data Studio also allows you to create copies of the procedure package as needed and to deploy the procedure to remote servers.

• Manually deploy the native SQL procedure by completing the following steps:
  a) Issue the CREATE PROCEDURE statement:
     − Include a procedure body written entirely in the SQL procedural language (SQL OL). For more information about what you can do within the procedure body, see SQL-routine-body in CREATE PROCEDURE (SQL - native) (Db2 SQL), SQL-control-statement in SQL procedural language (SQL PL) (Db2 SQL), and the following information:
       - “Controlling the scope of variables in an SQL procedure” on page 233
       - “Declaring cursors in an SQL procedure with nested compound statements” on page 236
       - “Handling SQL conditions in an SQL procedure” on page 237
       - “Raising a condition within an SQL procedure by using the SIGNAL or RESIGNAL statements” on page 246
     − Do not include the FENCED or EXTERNAL keywords, which specify the creation of an external SQL procedures, which are deprecated.
     − You can specify the AUTONOMOUS keyword to enable the procedure to commit without committing the work of the calling application. Autonomous procedures cannot see uncommitted changes of the calling application, and they cannot call other autonomous procedures.

     When you issue this CREATE PROCEDURE statement, the first version of this procedure is defined to Db2, and a package is implicitly bound with the options that you specify on the CREATE PROCEDURE statement.

  b) If the native SQL procedure contains one or more of the following statements or references, make copies of the native SQL procedure package, as needed:
     − CONNECT
     − SET CURRENT PACKAGESET
     − SET CURRENT PACKAGE PATH
     − A table reference with a three-part name that refers to a location other than the current server or refers to an alias that resolves to such a name.

  c) If you plan to call the native SQL procedure at another Db2 server, deploy the procedure to another Db2 for z/OS server. You can customize the bind options at the same time.

  d) Authorize the appropriate users to call the stored procedure.

What to do next
After you create a native SQL procedure, you can create additional versions of the procedure as needed. For more information, see “Creating new versions of native SQL procedures” on page 251.

Related concepts
SQL procedures
An SQL procedure is a stored procedure that contains only SQL statements.

SQL procedure body
The body of an SQL procedure contains one or more SQL statements. In the SQL procedure body, you can also declare and use variables, conditions, return codes, statements, cursors, and handlers.

Related tasks
Implementing Db2 stored procedures (Db2 Administration Guide)
Developing database routines (IBM Data Studio, IBM Optim Database Administrator, IBM infoSphere Data Architect, IBM Optim Development Studio)

Related reference
CREATE PROCEDURE (SQL - native) (Db2 SQL)
**Controlling the scope of variables in an SQL procedure**

Use nested compound statements within an SQL procedure to define the scope of SQL variables. You can reference the variable only within the compound statement in which it was declared and within any nested statements.

**Procedure**

To control the scope of a variable in an SQL procedure:

1. Declare the variable within the compound statement in which you want to reference it. Ensure that the variable name is unique within the compound statement, not including any nested statements. You can define variables with the same name in other compound statements in the same SQL procedure.

2. Reference the variable within that compound statement or any nested statements.

   **Recommendation:** If multiple variables with the same name exist within an SQL procedure, qualify the variable with the label from the compound statement in which it was declared. Otherwise, you might accidentally reference the wrong variable.

   If the variable name is unqualified and multiple variables with that name exist within the same scope, Db2 uses the variable in the innermost compound statement.

**Example**

The following example contains three declarations of the variable A. One instance is declared in the outer compound statement, which has the label OUTER1. The other instances are declared in the inner compound statements with the labels INNER1 and INNER2. In the INNER1 compound statement, Db2 presumes that the unqualified references to A in the assignment statement and UPDATE statement refer to the instance of A that is declared in the INNER1 compound statement. To refer to the instance of A that is declared in the OUTER1 compound statement, qualify the variable as OUTER1.A.

```sql
CREATE PROCEDURE P2 ()
LANGUAGE SQL
-- Outermost compound statement ------------------------
OUTER1: BEGIN
  DECLARE A INT DEFAULT 100;
  -- Inner compound statement with label INNER1 ---
INNER1: BEGIN
    DECLARE A INT DEFAULT NULL;
    DECLARE W INT DEFAULT NULL;
    SET A = A + OUTER1.A;
    UPDATE T1 SET T1.B = 5
    WHERE T1.B = A;
    SET OUTER1.A = 100;
    SET INNER1.A = 200;
  END INNER1;
-- End of inner compound statement INNER1 ------
-- Inner compound statement with label INNER2 ---
INNER2: BEGIN
    DECLARE A INT DEFAULT NULL;
    DECLARE Z INT DEFAULT NULL;
    SET A = A + OUTER1.A;
  END INNER2;
-- End of inner compound statement INNER2 ------
SET OUTER1.A = 100;
END OUTER1
```

The preceding example has the following parts:

1. The beginning of the outermost compound statement, which has the label OUTER1.
2. The beginning of the inner compound statement with the label INNER1.
3. The unqualified variable A refers to INNER1.A.
4. The unqualified variable A refers to INNER1.A.
5. OUTER1.A is a valid reference, because this variable is referenced in a nested compound statement.
6. INNER1.A is a valid reference, because this variable is referenced in the same compound statement in which it is declared. You cannot reference INNER2.A, because this variable is not in the scope of this compound statement.
7. The end of the inner compound statement with the label INNER1.
8. The beginning of the inner compound statement with the label INNER2.
9. The end of the inner compound statement with the label INNER2.
10. OUTER1.A is a valid reference, because this variable is referenced in the same compound statement in which it is declared. You cannot reference INNER1.A, because this variable is declared in a nested statement and cannot be referenced in the outer statement.
11. The end of the outermost compound statement, which has the label OUTER1.

Related concepts
Variables in SQL procedures
For data that you use only within an SQL procedure, you can declare SQL variables and store the values in the variables. SQL variables are similar to host variables in external stored procedures. SQL variables can be defined with the same data types and lengths as SQL procedure parameters.

References to SQL parameters and variables (Db2 SQL)

Nested compound statements in native SQL procedures
Nested compound statements are blocks of SQL statements that are contained by other blocks of SQL statements in native SQL procedures. Use nested compound statements to define condition handlers that execute more than one statement and to define different scopes for variables and condition handlers.

The following pseudo code shows a basic structure of an SQL procedure with nested compound statements:

```
CREATE PROCEDURE ...
OUTERMOST:  BEGIN
    ...
    INNER1:  BEGIN
    ...
    INNERMOST:  BEGIN
    ...
    END INNERMOST;
    INNER2:  BEGIN
    ...
    END INNER2;
END OUTERMOST
```

In the preceding code, the OUTERMOST compound statement contains two nested compound statements: INNER1 and INNER2. INNER1 contains one nested compound statement: INNERMOST.

Related concepts
Handlers in an SQL procedure
If an error occurs when an SQL procedure executes, the procedure ends unless you include statements to tell the procedure to perform some other action. These statements are called handlers.

Related tasks
Defining condition handlers that execute more than one statement
A condition handler defines the action that an SQL procedure takes when a particular condition occurs. You must specify the action as a single SQL procedure statement.

Statement labels for nested compound statements in native SQL procedures
You can define a label for each compound statement in an SQL procedure. This label enables you to reference this block of statements in other statements such as the GOTO, LEAVE, and ITERATE SQL PL control statements. You can also use the label to qualify a variable when necessary. Labels are not required.

A label name must meet the following criteria:

- Be unique within the compound statement, including any compound statements that are nested within the compound statement.
- Not be the same as the name of the SQL procedure.

You can reference a label within the compound statement in which it is defined, including any compound statements that are nested within that compound statement.

Example of statement labels: The following example shows several statement labels and their scope:

```sql
CREATE PROCEDURE P1 ()
LANGUAGE SQL
--Outermost compound statement ------------------------
OUTER1: BEGIN
    --Inner compound statement with label INNER1 ---
    INNER1: BEGIN
        IF...
            ABC: LEAVE INNER1;
        ELSEIF
            XYZ: LEAVE OUTER1;
        END IF
    END INNER1;
    --End of inner compound statement INNER1 ------
    --Inner compound statement with label INNER2---
    INNER2: BEGIN
        XYZ:...
    END INNER2;
    -- End of inner compound statement INNER2 ----- 
END OUTER1
```

The preceding example has the following parts:

1. The beginning of the outermost compound statement, which is labeled OUTER1.
2. The beginning of an innermost compound statement that is labeled INNER1.
3. A LEAVE statement that is defined with the label ABC. This LEAVE statement specifies that Db2 is to terminate processing of the compound statement INNER1 and begin processing the next statement, which is INNER2. This LEAVE statement cannot specify INNER2, because that label is not within the scope of the INNER1 compound statement.
4. A LEAVE statement that is defined with the label XYZ. This LEAVE statement specifies that Db2 is to terminate processing of the compound statement OUTER1 and begin processing the next statement, if one exists. This example does not show the next statement.
5. The beginning of an innermost compound statement that is labeled INNER2.
6. A statement that is defined with the label XYZ. This label is acceptable even though another statement in this procedure has the same label, because the two labels are in different scopes. Neither label is contained within the scope of the other.
7. The end of the outermost compound statement that is labeled OUTER1.

The following examples show valid and invalid uses of labels:
Invalid example of labels:

```
L1:  BEGIN
L2:  SET A = B;
L1:  GOTO L2:  --This duplicate use of the label L1 causes an error, because
       --the same label is already used in the same scope.
END L1;
```

Valid example of labels:

```
L1:  BEGIN
L2:  BEGIN
   L4: BEGIN  --This line contains the first use of the label L4
      DECLARE A CHAR(5);
      SET A = B;
      END L4;
      END L2;
L3:  BEGIN
   L4: BEGIN  --This second use of the label L4 is valid, because
      --it is used in a different scope.
      DECLARE A CHAR(5);
      SET A = B;
      END L4;
      END L3;
END L1;
```

**Declaring cursors in an SQL procedure with nested compound statements**

When you declare a cursor in an SQL procedure that has nested compound statements, you cannot necessarily reference the cursor anywhere in the procedure. The scope of the cursor is constrained to the compound statement in which you declare it.

**Procedure**

Specify the DECLARE CURSOR statement within the compound statement in which you want to reference the cursor. Use a cursor name that is unique within the SQL procedure.

You can reference the cursor within the compound statement in which it is declared and within any nested statements. If the cursor is declared as a result set cursor, even if the cursor is not declared in the outermost compound statement, any calling application can reference it.

**Example**

In the following example, cursor X is declared in the outer compound statement. This cursor can be referenced within the outer block in which it was declared and within any nested compound statements.

```
CREATE PROCEDURE SINGLE_CSR
   (INOUT IR1 INT, INOUT JR1 INT, INOUT IR2 INT, INOUT JR2 INT)
   LANGUAGE SQL
   DYNAMIC RESULT SETS 2
BEGIN
   DECLARE I INT;
   DECLARE J INT;
   DECLARE X CURSOR WITH RETURN FOR --outer declaration for X
      SELECT * FROM CSRT1;
   SUB: BEGIN
      OPEN X;                   --references X in outer block
      FETCH X INTO I,J;         --references X in outer block
      SET IR1 = I;
      SET JR1 = J;
      END;
      FETCH X INTO I,J;                --references X in outer block
      SET IR2 = I;
      SET JR2 = J;
      CLOSE X;
END
```
Related reference
CREATE PROCEDURE (SQL - native) (Db2 SQL)
DECLARE CURSOR (Db2 SQL)

Handling SQL conditions in an SQL procedure
In an SQL procedure, you can specify how the program should handle certain SQL errors and SQL warnings.

About this task
If you do not include a handler or a RETURN statement in the SQL procedure, Db2 automatically returns any SQL conditions to the caller in the SQLCA.

Procedure
To handle SQL conditions, use one of the following techniques:
• Include statements called handlers to tell the procedure to perform some other action when an error or warning occurs.
• Include a RETURN statement in an SQL procedure to return an integer status value to the caller.
• Include a SIGNAL statement or a RESIGNAL statement to raise a specific SQLSTATE and to define the message text for that SQLSTATE.
• Force a negative SQLCODE to be returned by a procedure if a trigger calls the procedure.

Handlers in an SQL procedure
If an error occurs when an SQL procedure executes, the procedure ends unless you include statements to tell the procedure to perform some other action. These statements are called handlers.

Handlers are similar to WHENEVER statements in external SQL application programs. Handlers tell the SQL procedure what to do when an error or warning occurs, or when no more rows are returned from a query. In addition, you can declare handlers for specific SQLSTATEs. You can refer to an SQLSTATE by its number in a handler, or you can declare a name for the SQLSTATE and then use that name in the handler.

The general form of a handler declaration is:
DECLARE handler-type HANDLER FOR condition SQL-procedure-statement;

In general, the way that a handler works is that when an error occurs that matches condition, the SQL-procedure-statement executes. When the SQL-procedure-statement completes, Db2 performs the action that is indicated by handler-type.

Types of handlers
The handler type determines what happens after the completion of the SQL-procedure-statement. You can declare the handler type to be either CONTINUE or EXIT:

CONTINUE
• Specifies that after SQL-procedure-statement completes, execution continues with the statement after the statement that caused the error.

EXIT
• Specifies that after SQL-procedure-statement completes, execution continues at the end of the compound statement that contains the handler.

Example: CONTINUE handler: This handler sets flag at_end when no more rows satisfy a query. The handler then causes execution to continue after the statement that returned no rows.

DECLARE CONTINUE HANDLER FOR NOT FOUND SET at_end=1;

Example: EXIT handler: This handler places the string ‘Table does not exist’ into output parameter OUT_BUFFER when condition NO_TABLE occurs. NO_TABLE is previously declared as SQLSTATE 42704.
(name is an undefined name). The handler then causes the SQL procedure to exit the compound statement in which the handler is declared.

```sql
DECLARE NO_TABLE CONDITION FOR '42704';
DECLARE EXIT HANDLER FOR NO_TABLE
SET OUT_BUFFER='Table does not exist';
```

**Defining condition handlers that execute more than one statement**

A condition handler defines the action that an SQL procedure takes when a particular condition occurs. You must specify the action as a single SQL procedure statement.

**Procedure**

To define a condition handler that executes more than one statement when the specified condition occurs, specify a compound statement within the declaration of that handler.

**Examples**

**Example**

The following example shows a condition handler that captures the SQLSTATE value and sets a local flag to TRUE.

```sql
BEGIN
DECLARE SQLSTATE CHAR(5);
DECLARE PrvSQLState CHAR(5) DEFAULT '00000';
DECLARE ExceptState INT;
DECLARE CONTINUE HANDLER FOR SQLEXCEPTION
BEGIN
    SET PrvSQLState = SQLSTATE;
    SET ExceptState = TRUE;
END;
...
END
```

**Example**

The following example declares a condition handler for SQLSTATE 72822. The subsequent SIGNAL statement is within the scope of this condition handler and thus activates this handler. The condition handler tests the value of the SQL variable VAR with an IF statement. Depending on the value of VAR, the SQLSTATE is changed and the message text is set.

```sql
DECLARE EXIT HANDLER FOR SQLSTATE '72822'
    IF ( VAR = 'OK' ) THEN
        RESIGNAL SQLSTATE '72623'
        SET MESSAGE_TEXT = 'Got SQLSTATE 72822';
    ELSE
        RESIGNAL SQLSTATE '72319'
        SET MESSAGE_TEXT = VAR;
    END IF;
SIGNAL SQLSTATE '72822';
```

**Related reference**

- compound-statement (Db2 SQL)

*Controlling how errors are handled within different scopes in an SQL procedure*

You can use nested compound statements in an SQL procedure to specify that errors be handled differently within different scopes. You can also ensure that condition handlers are checked only with a particular compound statement.

**Procedure**

To control how errors are handled within different scopes in an SQL procedure:
1. Optional: Declare a condition by specifying a DECLARE CONDITION statement within the compound statement in which you want to reference it. You can reference a condition in the declaration of a condition handler, a SIGNAL statement, or a RESIGNAL statement.

**Restriction:** If multiple conditions with that name exist within the same scope, you cannot explicitly refer to a condition that is not the most local in scope. Db2 uses the condition in the innermost compound statement.

2. Declare a condition handler by specifying a DECLARE HANDLER statement within the compound statement to which you want the condition handler to apply. Within the declaration of the condition handler, you can specify a previously defined condition.

**Restriction:** Condition handlers that are declared in the same compound statement cannot handle conditions encountered in each other or themselves.

**Examples**

**Example**

In the following example, a condition with the name ABC is declared twice, and a condition named XYZ is declared once.

```
CREATE PROCEDURE... 
DECLARE ABC CONDITION...
DECLARE XYZ CONDITION...
BEGIN
DECLARE ABC CONDITION...
SIGNAL ABC; 1
END;
SIGNAL ABC; 2
```

The following notes refer to the preceding example:

1. ABC refers to the condition that is declared in the innermost block. If this statement were changed to SIGNAL XYZ, XYZ would refer to the XYZ condition that is declared in the outermost block.
2. ABC refers to the condition that is declared in the outermost block.

**Example**

The following example contains multiple declarations of a condition with the name FOO, and a single declaration of a condition with the name GORP.

```
CREATE PROCEDURE MYTEST (INOUT A CHAR(1), INOUT B CHAR(1))
L1: BEGIN
DECLARE GORP CONDITION FOR SQLSTATE '33333'; -- defines a condition with the name GORP for SQLSTATE 33333
DECLARE EXIT HANDLER FOR GORP --defines a condition handler for SQLSTATE 33333
L2: BEGIN
DECLARE FOO CONDITION FOR SQLSTATE '12345'; --defines a condition with the name FOO for SQLSTATE 12345
DECLARE CONTINUE HANDLER FOR FOO --defines a condition handler for SQLSTATE 12345
L3: BEGIN
SET A = 'A';
...more statements...
END L3;
SET B = 'B';
IF...
SIGNAL FOO; --raises SQLSTATE 12345
ELSEIF
SIGNAL GORP; --raises SQLSTATE 33333
END IF;
END L2;
L4: BEGIN
DECLARE FOO CONDITION FOR SQLSTATE '54321' --defines a condition with the name FOO for SQLSTATE 54321
DECLARE EXIT HANDLER FOR FOO...; --defines a condition handler for SQLSTATE 54321
SIGNAL FOO SET MESSAGE_TEXT = '...'; --raises SQLSTATE 54321
```
L5: BEGIN
  DECLARE FOO CONDITION
  FOR SQLSTATE '99999'; --defines a condition with the name FOO for SQLSTATE 99999
  ...more statements...
END L5;
END L4;
--At this point, the procedure cannot reference FOO, because this condition is not defined
--in this outer scope
END L1

Example
In the following example, the compound statement with the label OUTER contains two other
compound statements: INNER1A and INNER1B. The INNER1A compound statement contains
another compound statement, which has the label INNER1A2, and the declaration for a condition
handler HINNER1A. The body of the condition handler HINNER1A contains another compound
statement, which defines another condition handler, HINNER1A_HANDLER.
The following notes apply to the preceding example:

1. If an exception, warning, or NOT FOUND condition occurs within the INNER1A2 compound statement, the most appropriate handler within that compound statement is activated to handle the condition. Then, one of the following actions occurs depending on the type of condition handler:
   - If the condition handler (HINNER1A2) is an exit handler, control is returned to the end of the compound statement that contained the condition handler.
   - If the condition handler (HINNER1A2) is a continue handler, processing continues with the statement after the statement that encountered the condition.

If no appropriate handler exists in the INNER1A2 compound statement, Db2 considers the following handlers in the specified order:
   a. The most appropriate handler within the INNER1A compound statement.
   b. The most appropriate handler within the OUTER compound statement.

If no appropriate handler exists in the OUTER compound statement, the condition is an unhandled condition. If the condition is an exception condition, the procedure terminates and returns an unhandled condition to the invoking application. If the condition is a warning or NOT FOUND condition, the procedure returns the unhandled warning condition to the invoking application.

2. If an exception, warning, or NOT FOUND condition occurs within the body of the condition handler HINNER1A, and the condition handler HINNER1A_HANDLER is the most appropriate handler for the exception, that handler is activated. Otherwise, the most appropriate handler within the OUTER compound statement handles the condition. If no appropriate handler exists within the OUTER compound statement, the condition is treated as an unhandled condition.

Example

In the following example, when statement2 results in a NOT FOUND condition, the appropriate condition handler is activated to handle the condition. When the condition handler completes, the compound statement that contains that condition handler terminates, because the condition handler is an EXIT handler. Processing then continues with statement4.

```
BEGIN
  DECLARE EXIT HANDLER FOR NOT FOUND
    SET OUT_OF_DATA_FLAG = ON;
  statement1...
  statement2... --assume that this statement results in a NOT FOUND condition
  statement3...
END;
statement4 ...
```

Example

In the following example, Db2 checks for SQLSTATE 22H11 only for statements inside the INNER compound statement. Db2 checks for SQLEXCEPTION for all statements in both the OUTER and INNER blocks.

```
OUTER: BEGIN
  DECLARE var1 INT;
  DECLARE EXIT HANDLER FOR SQLEXCEPTION
    RETURN -3;
INNER: BEGIN
  DECLARE EXIT HANDLER FOR SQLSTATE '22H11'
    RETURN -1;
  DECLARE C1 CURSOR FOR SELECT col1 FROM table1;
  OPEN C1;
  CLOSE C1;
  :
  : -- more statements
```

Chapter 3. Db2 SQL programming 241
Example

In the following example, Db2 checks for SQLSTATE 42704 only for statements inside the A compound statement.

```sql
CREATE PROCEDURE EXIT_TEST ()
    LANGUAGE SQL
    BEGIN
        DECLARE OUT_BUFFER VARCHAR(80);
        DECLARE NO_TABLE CONDITION FOR SQLSTATE '42704';

        A: BEGIN
            DECLARE EXIT HANDLER FOR NO_TABLE
            BEGIN
                SET OUT_BUFFER = 'Table does not exist';
            END;
            -- Drop potentially nonexistent table:
            DROP TABLE JAVELIN;
        B: SET OUT_BUFFER = 'Table dropped successfully';
        END;
        -- Copy OUT_BUFFER to some message table:
        C: INSERT INTO MESSAGES VALUES (OUT_BUFFER);
    END;
```

The following notes describe a possible flow for the preceding example:

1. A nested compound statement with label A confines the scope of the NO_TABLE exit handler to the statements that are specified in the A compound statement.
2. If the table JAVELIN does not exist, the DROP statement raises the NO_TABLE condition.
3. The exit handler for NO_TABLE is activated.
4. The variable OUT_BUFFER is set to the string 'Table does not exist.'
5. Execution continues with the INSERT statement. No more statements in the A compound statement are processed.

Example

The following example illustrates the scope of different condition handlers.

```sql
CREATE PROCEDURE ERROR_HANDLERS(IN PARAM INTEGER)
    LANGUAGE SQL
    OUTER: BEGIN
        DECLARE I INTEGER;
        DECLARE SQLSTATE CHAR(5) DEFAULT '00000';
        DECLARE EXIT HANDLER FOR
            SQLSTATE VALUE '38H02',
            SQLSTATE VALUE '38H04',
            SQLSTATE VALUE '38HI4',
            SQLSTATE VALUE '38H06'
            OUTER_HANDLER: BEGIN
                DECLARE TEXT VARCHAR(70);
                SET TEXT = SQLSTATE || '
                ' RECEIVED AND MANAGED BY OUTER ERROR HANDLER' ;
                RESIGNAL SQLSTATE VALUE '38HE0'
                SET MESSAGE_TEXT = TEXT;
            END OUTER_HANDLER;

        INNER: BEGIN
            DECLARE EXIT HANDLER FOR SQLSTATE VALUE '38H03'
            RESIGNAL SQLSTATE VALUE '38HI3'
            SET MESSAGE_TEXT = '38H03 MANAGED BY INNER ERROR HANDLER';
            DECLARE EXIT HANDLER FOR SQLSTATE VALUE '38H04'
            RESIGNAL SQLSTATE VALUE '38HI4'
            SET MESSAGE_TEXT = '38H04 MANAGED BY INNER ERROR HANDLER';
            DECLARE EXIT HANDLER FOR SQLSTATE VALUE '38H05'
            RESIGNAL SQLSTATE VALUE '38HI5'
            SET MESSAGE_TEXT = '38H05 MANAGED BY INNER ERROR HANDLER';
    END;
```
CASE PARAM
WHEN 1 THEN  -- (1)
  SIGNAL SQLSTATE VALUE '38H01'
  SET MESSAGE_TEXT = 'EXAMPLE 1: ERROR SIGNALED FROM INNER COMPOUND STMT';
WHEN 2 THEN  -- (2)
  SIGNAL SQLSTATE VALUE '38H02'
  SET MESSAGE_TEXT = 'EXAMPLE 2: ERROR SIGNALED FROM INNER COMPOUND STMT';
WHEN 3 THEN  -- (3)
  SIGNAL SQLSTATE VALUE '38H03'
  SET MESSAGE_TEXT = 'EXAMPLE 3: ERROR SIGNALED FROM INNER COMPOUND STMT';
WHEN 4 THEN  -- (4)
  SIGNAL SQLSTATE VALUE '38H04'
  SET MESSAGE_TEXT = 'EXAMPLE 4: ERROR SIGNALED FROM INNER COMPOUND STMT';
ELSE
  SET I = 1; /*Do not do anything */
END CASE;
END INNER;
CASE PARAM
WHEN 5 THEN  -- (5)
  SIGNAL SQLSTATE VALUE '38H05'
  SET MESSAGE_TEXT = 'EXAMPLE 5: ERROR SIGNALED FROM OUTER COMPOUND STMT';
WHEN 6 THEN  -- (6)
  SIGNAL SQLSTATE VALUE '38H06'
  SET MESSAGE_TEXT = 'EXAMPLE 6: ERROR SIGNALED FROM OUTER COMPOUND STMT';
ELSE
  SET I = 1; /*Do not do anything */
END CASE;
END OUTER

The following table summarizes the behavior of the preceding example:

<table>
<thead>
<tr>
<th>Input value for PARM</th>
<th>Expected behavior</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>SQLSTATE 38H01 is signaled from the INNER compound statement. Because no appropriate handler exists, the procedure terminates and returns the unhandled exception condition, 38H01 with SQLCODE -438, to the calling application.</td>
</tr>
<tr>
<td>2</td>
<td>SQLSTATE 38H02 is signaled from the INNER compound statement. The condition handler in the OUTER compound statement is activated. A RESIGNAL statement, with SQLSTATE 38HE0, is issued from within the body of the condition handler. This exception causes control to be returned to the end of the OUTER compound statement with exception condition 38HE0 and SQLCODE -438. The procedure terminates and returns the unhandled condition to the calling application.</td>
</tr>
<tr>
<td>3</td>
<td>SQLSTATE 38H03 is signaled from the INNER compound statement. A condition handler within the INNER compound statement is activated. A RESIGNAL statement, with SQLSTATE 38HI3, is issued from within the body of the condition handler. Because no appropriate handler exists, the procedure terminates and returns the unhandled exception condition, 38HI3 with SQLCODE -438, to the calling application.</td>
</tr>
</tbody>
</table>
Input value for PARM | Expected behavior
--- | ---
4 | SQLSTATE 38H04 is signaled from the INNER compound statement. A condition handler within the INNER compound statement is activated. A RESIGNAL statement, with SQLSTATE 38H14, is issued from within the body of the condition handler. A condition handler in the OUTER compound statement is activated. A RESIGNAL statement, with SQLSTATE 38HE0, is issued from within the body of the condition handler. This exception causes control to be returned to the end of the OUTER compound statement with exception condition 38HE0 and SQLCODE -438. The procedure terminates and returns the unhandled condition to the calling application.

5 | SQLSTATE 38H05 is signaled from the OUTER compound statement. Because no appropriate handler exists, the procedure terminates and returns the unhandled exception condition, 38H05 with SQLCODE -438, to the calling application.

6 | SQLSTATE 38H06 is signaled from the OUTER compound statement. A condition handler in the OUTER compound statement is activated. A RESIGNAL statement, with SQLSTATE 38HE0, is issued from within the body of the condition handler. This exception causes control to be returned to the end of the OUTER compound statement with exception condition 38HE0 and SQLCODE -438. The procedure terminates and returns the unhandled condition to the calling application.

7 | The ELSE clause of the CASE statement executes and processes the SET statement. A successful completion code is returned to the calling application.

Example
In the following example SQL procedure, the condition handler for exception1 is not within the scope of the condition handler for exception0. If exception condition exception1 is raised in the body of the condition handler for exception0, no appropriate handler exists, and the procedure terminates with an unhandled exception.

```sql
CREATE PROCEDURE divide ( .....)
LANGUAGE SQL CONTAINS SQL
BEGIN
  DECLARE dn_too_long CHAR(5) DEFAULT 'abcde';
  -- Declare condition names --------------------------
  DECLARE exception0  CONDITION FOR SQLSTATE '22001';
  DECLARE exception1  CONDITION FOR SQLSTATE 'xxxxx';
  -- Declare cursors --------------------------------
  DECLARE cursor1 CURSOR WITH RETURN FOR
    SELECT * FROM dept;
  -- Declare handlers --------------------------------
  DECLARE CONTINUE HANDLER FOR exception0
    BEGIN
    some SQL statement that causes an error 'xxxxx'
  END
  DECLARE CONTINUE HANDLER FOR exception1
    BEGIN
    ...
  END
  -- Mainline of procedure --------------------------
  INSERT INTO DEPT (DEPTNO) VALUES (dn_too_long);
  -- Assume that this statement results in SQLSTATE '22001'
  OPEN CURSOR1;
END
```
Retrieving diagnostic information by using GET DIAGNOSTICS in a handler

Handlers specify the action that an SQL procedure takes when a particular error or condition occurs. In some cases, you want to retrieve additional diagnostic information about the error or warning condition.

About this task

Procedure

You can include a GET DIAGNOSTICS statement in a handler to retrieve error or warning information. If you include GET DIAGNOSTICS, it must be the first statement that is specified in the handler.

Example: Using GET DIAGNOSTICS to retrieve message text

Suppose that you create an SQL procedure, named divide1, that computes the result of the division of two integers. You include GET DIAGNOSTICS to return the text of the division error message as an output parameter:

```
CREATE PROCEDURE divide1
  (IN numerator INTEGER, IN denominator INTEGER,
   OUT divide_result INTEGER, OUT divide_error VARCHAR(1000))
LANGUAGE SQL
BEGIN
  DECLARE CONTINUE HANDLER FOR SQLEXCEPTION
  GET DIAGNOSTICS CONDITION 1 divide_error = MESSAGE_TEXT;
  SET divide_result = numerator / denominator;
END
```

Ignoring a condition in an SQL procedure

You can specify that you want to ignore errors or warnings within a particular scope of statements in an SQL procedure. However, do so with caution.

Procedure

Declare a condition handler that contains an empty compound statement.

Example

The following example shows a condition handler that is declared as a way of ignoring a condition. Assume that your SQL procedure inserts rows into a table that has a unique column. If the value to be inserted for that column already exists in the table, the row is not inserted. However, in this case, you do not want Db2 to notify the application about this condition, which is indicated by SQLSTATE 23505.

```
DECLARE CONTINUE HANDLER FOR SQLSTATE '23505'
BEGIN  -- ignore error for duplicate value
END;
```

Related concepts

Handlers in an SQL procedure

If an error occurs when an SQL procedure executes, the procedure ends unless you include statements to tell the procedure to perform some other action. These statements are called handlers.

Related reference

SQLSTATE values and common error codes (Db2 Codes)
Raising a condition within an SQL procedure by using the SIGNAL or RESIGNAL statements

Within an SQL procedure, you can force a particular condition to occur with a specific SQLSTATE and message text.

About this task

You can use either a SIGNAL or RESIGNAL statement to raise a condition with a specific SQLSTATE and message text within an SQL procedure. The SIGNAL and RESIGNAL statements differ in the following ways:

- You can use the SIGNAL statement anywhere within an SQL procedure. You must specify the SQLSTATE value. In addition, you can use the SIGNAL statement in a trigger body. For information about using the SIGNAL statement in a trigger, see “Creating a trigger” on page 149.
- You can use the RESIGNAL statement only within a handler of an SQL procedure. If you do not specify the SQLSTATE value, Db2 uses the same SQLSTATE value that activated the handler.

You can use any valid SQLSTATE value in a SIGNAL or RESIGNAL statement, except an SQLSTATE class with '00' as the first two characters.

The following table summarizes the differences between issuing a RESIGNAL or SIGNAL statement within the body of a condition handler. For each row in the table, assume that the diagnostics area contains the following information when the RESIGNAL or SIGNAL statement is issued:

<table>
<thead>
<tr>
<th>Specify a new condition?</th>
<th>Specify message text?</th>
<th>Example RESIGNAL statement...</th>
<th>Example SIGNAL statement...</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>No</td>
<td>No</td>
<td>RESIGNAL</td>
<td>Not possible</td>
<td>RETURNED_SQLSTATE xxxxx MESSAGE_TEXT 'this is my message'</td>
</tr>
<tr>
<td>Yes</td>
<td>No</td>
<td>RESIGNAL '98765'</td>
<td>SIGNAL '98765'</td>
<td>RETURNED_SQLSTATE 98765 MESSAGE_TEXT 'APPLICATION RAISED ERROR WITH DIAGNOSTIC TEXT: this is my message'</td>
</tr>
<tr>
<td>No</td>
<td>Yes</td>
<td>Not possible</td>
<td>Not possible</td>
<td>NA</td>
</tr>
<tr>
<td>Yes</td>
<td>Yes</td>
<td>RESIGNAL '98765' SET MESSAGE_TEXT = 'xyz'</td>
<td>SIGNAL '98765' SET MESSAGE_TEXT = 'xyz'</td>
<td>RETURNED_SQLSTATE 98765 MESSAGE_TEXT 'APPLICATION RAISED ERROR WITH DIAGNOSTIC TEXT: xyz'</td>
</tr>
</tbody>
</table>

Note:

1. This statement raises the current condition with the existing SQLSTATE, SQLCODE, message text, and tokens.
2. This statement raises a new condition (SQLSTATE '98765'). Existing message text and tokens are reset. The SQLCODE is set to -438 for an error or 438 for a warning.
3. This statement raises a new condition (SQLSTATE '98765') with new message text ('xyz'). The SQLCODE is set to -438 for an error or 438 for a warning.

Example of the SIGNAL statement in an SQL procedure

You can use the SIGNAL statement anywhere within an SQL procedure to raise a particular condition.

The following example uses an ORDERS table and a CUSTOMERS table that are defined in the following way:

```sql
CREATE TABLE ORDERS
(ORDERNO   INTEGER NOT NULL,
 PARTNO    INTEGER NOT NULL,
 ORDER_DATE DATE DEFAULT,
 CUSTNO    INTEGER NOT NULL,
 QUANTITY   SMALLINT NOT NULL,
 CONSTRAINT REF_CUSTNO FOREIGN KEY (CUSTNO)
 REFERENCES CUSTOMERS (CUSTNO) ON DELETE RESTRICT,
 PRIMARY KEY (ORDERNO, PARTNO));
```

```sql
CREATE TABLE CUSTOMERS
(CUSTNO    INTEGER NOT NULL,
 CUSTNAME  VARCHAR(30),
 CUSTADDR  VARCHAR(80),
 PRIMARY KEY (CUSTNO));
```

Example: Using SIGNAL to set message text

Suppose that you have an SQL procedure for an order system that signals an application error when a customer number is not known to the application. The ORDERS table has a foreign key to the CUSTOMERS table, which requires that the CUSTNO exist in the CUSTOMERS table before an order can be inserted:

```sql
CREATE PROCEDURE submit_order
(IN ONUM INTEGER, IN PNUM INTEGER,
 IN CNUM INTEGER, IN QNUM INTEGER)
LANGUAGE SQL
MODIFIES SQL DATA
BEGIN
DECLARE EXIT HANDLER FOR SQLSTATE VALUE '23503'
SIGNAL SQLSTATE '75002'
SET MESSAGE_TEXT = 'Customer number is not known';
INSERT INTO ORDERS (ORDERNO, PARTNO, CUSTNO, QUANTITY)
VALUES (ONUM, PNUM, CNUM, QNUM);
END
```

In this example, the SIGNAL statement is in the handler. However, you can use the SIGNAL statement to invoke a handler when a condition occurs that will result in an error.

Related concepts

Example of the RESIGNAL statement in a handler

You can use the RESIGNAL statement in an SQL procedure to assign a different value to the condition that activated the handler.

Example: Using RESIGNAL to set an SQLSTATE valu

Suppose that you create an SQL procedure, named divide2, that computes the result of the division of two integers. You include SIGNAL to invoke the handler with an overflow condition that is caused by a zero divisor, and you include RESIGNAL to set a different SQLSTATE value for that overflow condition:

```sql
CREATE PROCEDURE divide2
(IN numerator INTEGER, IN denominator INTEGER,
 OUT divide_result INTEGER)
LANGUAGE SQL
BEGIN
DECLARE overflow CONDITION FOR SQLSTATE '22003';
```
DECLARE CONTINUE HANDLER FOR overflow
    RESIGNAL SQLSTATE '22375';
IF denominator = 0 THEN
    SIGNAL overflow;
ELSE
    SET divide_result = numerator / denominator;
END IF;
END

Example: RESIGNAL in a nested compound statement

If the following SQL procedure is invoked with argument values 1, 0, and 0, the procedure returns a value of 2 for RC and sets the oparm1 parameter to 650.

CREATE PROCEDURE resig4
    (IN iparm1 INTEGER, INOUT oparm1 INTEGER, INOUT rc INTEGER)
LANGUAGE SQL
A1: BEGIN
    DECLARE c1 INT DEFAULT 1;
    DECLARE CONTINUE HANDLER FOR SQLSTATE VALUE '01ABX'
    BEGIN
        .... some other statements
        SET RC = 3;
    END;
A2: SET oparm1 = 5;
A3: BEGIN
    DECLARE c1 INT DEFAULT 1;
    DECLARE CONTINUE HANDLER
        FOR SQLSTATE VALUE '01ABC'
    BEGIN
        SET RC = 1;
        RESIGNAL SQLSTATE VALUE '01ABX'
        SET MESSAGE_TEXT = 'get out of here';
        SET RC = 2;
    END;
A7: SET oparm1 = oparm1 + 110;
SIGNAL SQLSTATE VALUE '01ABC'
    SET MESSAGE_TEXT = 'yikes';
    SET oparm1 = oparm1 + 215;
END;
SET oparm1 = oparm1 + 320;
END

The following notes refer to the preceding example:

1. oparm1 is initially set to 5.
2. oparm1 is incremented by 110. The value of oparm1 is now 115.
3. The SIGNAL statement causes the condition handler that is contained in the A3 compound statement to be activated.
4. In this condition handler, RC is set to 1.
5. The RESIGNAL statement changes the SQLSTATE to 01ABX. This value causes the continue handler in the A1 compound statement to be activated.
6. RC is set to 3 in this condition handler. Because this condition handler is a continue handler, when the handler action completes, control returns to the SET statement after the RESIGNAL statement.
7. RC is set to 2 in this condition handler. Because this condition handler is a continue handler, control returns to the SET statement that follows the SIGNAL statement that caused the condition handler to be activated.
8. oparm1 is incremented by 215. The value of oparm is now 330.
9. oparm1 is incremented by 320. The value of oparm is now 650.
How SIGNAL and RESIGNAL statements affect the diagnostics area
When you issue a SIGNAL statement, a new logical diagnostics area is created. When you issue a RESIGNAL statement, the current diagnostics area is updated.

When you issue a SIGNAL statement, a new diagnostics area is logically created. In that diagnostics area, RETURNED_SQLSTATE is set to the SQLSTATE or condition name specified. If you specified message text as part of the SIGNAL statement, MESSAGE_TEXT in the diagnostics area is also set to the specified value.

When you issue a RESIGNAL statement with a SQLSTATE value, condition name, or message text, the current diagnostics area is updated with the specified information.

Making copies of a package for a native SQL procedure
When you create a native SQL procedure, a package is implicitly bound with the options that you specified on the CREATE PROCEDURE statement. If the native SQL procedure performs certain actions, you need to explicitly make copies of that package.

About this task
If the native SQL procedure performs one or more of the following actions, you need to create copies of the package for that procedure:

- Uses a CONNECT statement to connect to a database server.
- Refers to a table with a three part name that includes a location other than the current server or refers to an alias that resolves to such a name.
- Sets the CURRENT PACKAGESET special register to control which package is invoked for that version of the procedure.
- Sets the CURRENT PACKAGE PATH special register to control which package is invoked for that version of the procedure.

The package for a version of a procedure has the following name: location.collection-id.package-id.version-id where these variables have the following values:

- location
  Value of the CURRENT SERVER special register
- collection-id
  Schema qualifier of the procedure
- package-id
  Procedure name
- version-id
  Version identifier

To make copies of a package for a native SQL procedure, specify the BIND PACKAGE command with the COPY option. For copies that are created on the current server, specify a different schema qualifier, which is the collection ID. For the first copy that is created on a remote server, you can specify the same schema qualifier. For other copies on that remote server, specify a different schema qualifier.

If you later change the native SQL procedure, you might need to explicitly rebind any local or remote copies of the package that exist for that version of the procedure.

Examples
Example
Because the following native SQL procedure contains a CONNECT statement, you must create a copy of the package at the target server, which in this case is at location SAN_JOSE. The subsequent BIND command creates a copy of the package for version ABC of the procedure TEST.MYPROC. This package is created at location SAN_JOSE and is used by Db2 when this procedure is executed.

```sql
CREATE PROCEDURE TEST.MYPROC VERSION ABC LANGUAGE SQL ...
BEGIN
  ...
  CONNECT TO SAN_JOSE
```
BIND PACKAGE (SAN_JOSE.TEST) COPY(TEST.MYPROC) COPYVER(ABC) ACTION(ADD)

Example
The following native SQL procedure sets the CURRENT PACKAGESET special register to ensure that Db2 uses the package with the collection ID COLL2 for this version of the procedure. Consequently, you must create such a package. The subsequent BIND command creates this package with collection ID COLL2. This package is a copy of the package for version ABC of the procedure TEST.MYPROC. Db2 uses this package to process the SQL statements in this procedure.

CREATE PROCEDURE TEST.MYPROC VERSION ABC LANGUAGE SQL ...
BEGIN
  SET CURRENT PACKAGESET = 'COLL2'
  ...
END
BIND PACKAGE(COLL2) COPY(TEST.MYPROC) COPYVER(ABC) ACTION(ADD) QUALIFIER(XYZ)

Related tasks
Regenerating an existing version of a native SQL procedure
When you apply Db2 maintenance that changes how native SQL procedures are generated, you need to regenerate any affected procedures. When you regenerate a version of a native SQL procedure, Db2 rebinds the associated package for that version of the procedure.

Replacing copies of a package for a version of a native SQL procedure
When you change a version of a native SQL procedure and the ALTER PROCEDURE REPLACE statement contains certain options, you must replace any local or remote copies of the package that exist for that version of the procedure.

Related reference
ALTER PROCEDURE (SQL - native) (Db2 SQL)

Replacing copies of a package for a version of a native SQL procedure
When you change a version of a native SQL procedure and the ALTER PROCEDURE REPLACE statement contains certain options, you must replace any local or remote copies of the package that exist for that version of the procedure.

About this task
If you specify any of the following ALTER PROCEDURE options, you must replace copies of the package:
- REPLACE VERSION
- REGENERATE
- DISABLE DEBUG MODE
- QUALIFIER
- PACKAGE OWNER
- DEFER PREPARE
- NODEFER PREPARE
- CURRENT DATA
- DEGREE
- DYNAMICRULES
- APPLICATION ENCODING SCHEME
- WITH EXPLAIN
- WITHOUT EXPLAIN
- WITH IMMEDIATE WRITE
To replace copies of a package for a version of a native SQL procedure, specify the BIND COPY ACTION(REPLACE) command with the appropriate package name and version ID.

**Creating new versions of native SQL procedures**

A new version of a native SQL procedure can have different parameter names, procedure options, or procedure body.

**About this task**

All versions of a procedure must have the same procedure signature. Therefore, each version of the procedure must have the same of the following items:

- Schema name
- Procedure name
- Number of parameters
- Data types for corresponding parameters

When any single version of a procedure is defined as autonomous, all versions must be defined as autonomous.

**Important:** Do not create additional versions of procedures that are supplied with Db2 by specifying the VERSION keyword. Only versions that are supplied with Db2 are supported. Additional versions of such routines cause the installation and configuration of the supplied routines to fail.

**Procedure**

To create a new version of a procedure, issue one of the following:

- **FL 507** The CREATE PROCEDURE statement with the following items:
  - The OR REPLACE clause.
  - The VERSION clause with a new version identifier.
- The ALTER PROCEDURE statement with the following items:
  - The ADD VERSION clause with a name for the new version.

For either statement, you must include the following:
• The name of the native SQL procedure for which you want to create a new version.
• The parameter list of the procedure that you want to change. For ALTER PROCEDURE ADD VERSION, this parameter list must be the same as the original procedure.
• Any procedure options. These options can be different than the options for other versions of this procedure. If you do not specify a value for a particular option, the default value is used, regardless of the value that is used by the current active version of this procedure.
• A procedure body. This body can be different than the procedure body for other versions of this procedure.

Examples

Example 1
For example, the following CREATE PROCEDURE statement defines a new native SQL procedure called UPDATE_BALANCE. The version of the procedure is V1, and it is the active version.

```sql
CREATE PROCEDURE UPDATE_BALANCE
(IN CUSTOMER_NO INTEGER,
IN AMOUNT DECIMAL(9,2))
VERSION V1
LANGUAGE SQL
READS SQL DATA
BEGIN
DECLARE CUSTOMER_NAME CHAR(20);
SELECT CUSTNAME
    INTO CUSTOMER_NAME
    FROM ACCOUNTS
    WHERE CUSTNO = CUSTOMER_NO;
END
```

Example 2
The following ALTER PROCEDURE statement creates a new version of the UPDATE_BALANCE procedure. The version name of the new version is V2. This new version has a different procedure body.

```sql
ALTER PROCEDURE UPDATE_BALANCE
ADD VERSION V2
(IN CUSTOMER_NO INTEGER,
IN AMOUNT DECIMAL (9,2))
MODIFIES SQL DATA
BEGIN
UPDATE ACCOUNTS
SET BAL = BAL + AMOUNT
WHERE CUSTNO = CUSTOMER_NO;
END
```

Example 3:
FL 507
The following CREATE PROCEDURE statement with the OR REPLACE clause creates a new version of the UPDATE_BALANCE procedure, assuming that version V3 does not already exist (if V3 already exists, this statement would replace the existing definition). This version changes the procedure body in the same way as in Example 2:

```sql
CREATE OR REPLACE PROCEDURE UPDATE_BALANCE
(IN CUSTOMER_NO INTEGER,
IN AMOUNT DECIMAL(9,2))
VERSION V3
LANGUAGE SQL
MODIFIES SQL DATA
BEGIN
UPDATE ACCOUNTS
SET BAL = BAL + AMOUNT
WHERE CUSTNO = CUSTOMER_NO;
END
```
What to do next

After you create a new version, if you want that version to be invoked by all subsequent calls to this procedure, you need to make that version the active version. You can use the ACTIVATE VERSION clause on either an ALTER PROCEDURE statement or a CREATE PROCEDURE statement with the OR REPLACE clause.

Related reference

ALTER PROCEDURE (SQL - native) (Db2 SQL)
CREATE PROCEDURE (SQL - native) (Db2 SQL)

Multiple versions of native SQL procedures

You can define multiple versions of a native SQL procedure. Db2 maintains this version information for you.

One or more versions of a procedure can exist at any point in time at the current server, but only one version of a procedure is considered the active version. When you first create a procedure, that initial version is considered the active version of the procedure.

Using multiple versions of a native SQL procedure has the following advantages:

• You can keep the existing version of a procedure active while you create another version. When the other version is ready, you can make it the active one.
• When you make another version of a procedure active, you do not need to change any existing calls to that procedure.
• You can easily switch back to a previous version of a procedure if the version that you switched to does not work as planned.
• You can drop an unneeded version of a procedure.

A new version of a native SQL procedure can have different values for the following items:

• Parameter names
• Procedure options (except for the AUTONOMOUS option, which must be specified for all versions or none)
• Procedure body

Restrictions:

• A new version of a native SQL procedure cannot have different values for the following items:
  – Number of parameters
  – Parameter data types
  – Parameter attributes for character data
  – Parameter CCSIDs
  – Whether a parameter is an input or output parameter, as defined by the IN, OUT, and INOUT options

If you need to specify different values for any of the preceding items, create a new native SQL procedure, instead of a new version.

• When the AUTONOMOUS option is specified for one version of a procedure, it must be specified for every version of that procedure.

Deploying a native SQL procedure to another Db2 for z/OS server

When deploying a native SQL procedure to another Db2 for z/OS server, you can change the bind options to better match the deploying environment. The procedure logic remains the same.

Before you begin

Deprecated function: The DEPLOY bind option is deprecated. For best results, deploy compiled SQL functions and native SQL procedures to multiple environments by issuing the same CREATE or ALTER statements separately in each Db2 environment.
Requirements:

- The remote server must be properly defined in the communications database of the Db2 subsystem from which you deploy the native SQL procedure.
- The target Db2 subsystem must be operating at a PTF level that is compatible with the PTF level of the local Db2 subsystem.

Procedure

Issue the BIND PACKAGE command with the following options:

DEPLOY
 Specify the name of the procedure whose logic you want to use on the target server.

Tip: When specifying the parameters for the DEPLOY option, consider the following naming rules for native SQL procedures:
- The collection ID is the same as the schema name in the original CREATE PROCEDURE statement.
- The package ID is the same as the procedure name in the original CREATE PROCEDURE statement.

COPYVER
 Specify the version of the procedure whose logic you want to use on the target server.

ACTION(ADD) or ACTION(REPLACE)
 Specify whether you want Db2 to create a new version of the native SQL procedure and its associated package or to replace the specified version.

Optionally, you can also specify the bind options QUALIFIER or OWNER if want to change them.

Examples

Deploying the same version of a procedure at another location
The following BIND command creates a native SQL procedure with the name PRODUCTION.MYPROC at the CHICAGO location. This procedure is created from the procedure TEST.MYPROC at the current site. Both native SQL procedures have the same content and version, ABC. However, the package for the procedure CHICAGO.PRODUCTION.MYPROC has XYZ as its qualifier.

```
CREATE PROCEDURE TEST.MYPROC VERSION ABC LANGUAGE SQL ...
BEGIN
  ...
END
BIND PACKAGE(CHICAGO.PRODUCTION) DEPLOY(TEST.MYPROC) COPYVER(ABC)
ACTION(ADD) QUALIFIER(XYZ)
```

Replacing a version of a procedure at another location
The following BIND command replaces version ABC of the procedure PRODUCTION.MYPROC at the CHICAGO location with version ABC of the procedure TEST.MYPROC at the current site.

```
BIND PACKAGE(CHICAGO.PRODUCTION) DEPLOY(TEST.MYPROC) COPYVER(ABC)
ACTION(REPLACE) REPLVER(ABC)
```

Related concepts
Communications database for the server (Managing Security)

Related reference
BIND and REBIND options for packages, plans, and services (Db2 Commands)
BIND PACKAGE (DSN) (Db2 Commands)

Related information
Db2 for z/OS Stored Procedures: Through the CALL and Beyond (IBM Redbooks)
**Removing an existing version of a native SQL procedure**
You can drop a particular version of a native SQL procedure without dropping the other versions of the procedure.

**Before you begin**
Before you remove an existing version of a native SQL procedure, ensure that the version is not active. If the version is the active version, designate a different active version before proceeding.

**Procedure**
Issue the ALTER PROCEDURE statement with the DROP VERSION clause and the name of the version that you want to drop. If you instead want to drop all versions of the procedure, use the DROP statement.

**Examples**

**Example of dropping a version that is not active**
The following statement drops the OLD_PRODUCTION version of the P1 procedure.

```
ALTER PROCEDURE P1 DROP VERSION OLD_PRODUCTION
```

**Example of dropping an active version**
Assume that the OLD_PRODUCTION version of the P1 procedure is the active version. The following example first switches the active version to NEW_PRODUCTION and then drops the OLD_PRODUCTION version.

```
ALTER PROCEDURE P1 ACTIVATE VERSION NEW_PRODUCTION;
ALTER PROCEDURE P1 DROP VERSION OLD_PRODUCTION;
```

**Related tasks**
Designating the active version of a native SQL procedure
When a native SQL procedure is called, Db2 uses the version that is designated as the active version.

**Regenerating an existing version of a native SQL procedure**
When you apply Db2 maintenance that changes how native SQL procedures are generated, you need to regenerate any affected procedures. When you regenerate a version of a native SQL procedure, Db2 rebinds the associated package for that version of the procedure.

**About this task**
ALTER PROCEDURE REGENERATE is different than the REBIND PACKAGE command. When you specify REBIND PACKAGE, Db2 rebinds only the non-control SQL statements. Use this command when you think rebinding will improve the access path. When you specify ALTER PROCEDURE REGENERATE, Db2 rebinds the SQL control statements as well as the non-control statements.

**Procedure**
To regenerate an existing version of a native SQL procedure:
1. Issue the ALTER PROCEDURE statement with the REGENERATE clause and specify the version to be regenerated.
2. If copies of the package for the specified version of the procedure exist at remote sites, replace those packages. Issue the BIND PACKAGE command with the COPY option and appropriate location for each remote package.
3. If copies of the package for the specified version of the procedure exist locally with different schema names, replace those packages. Issue the BIND PACKAGE command with the COPY option and appropriate schema for each local package.
Example
The following ALTER PROCEDURE statement regenerates the active version of the UPDATE_SALARY_1 procedure.

```sql
ALTER PROCEDURE UPDATE_SALARY_1
REGENERATE ACTIVE VERSION
```

Changing an existing version of a native SQL procedure
You can change an option or the procedure body for a particular version of a native SQL procedure. If you want to keep a copy of that stored procedure, consider creating a new version instead of changing the existing version.

Procedure
To change an existing version of a native SQL procedure, issue one of the following statements:

- **FL 507** The CREATE PROCEDURE statement with the OR REPLACE and the VERSION clause that identifies the version to be replaced.
- The ALTER PROCEDURE statement with the REPLACE VERSION clause.

Any option that you do not explicitly specify inherits the system default values. This inheritance occurs even if those options were explicitly specified for a prior version by using a CREATE PROCEDURE statement, ALTER PROCEDURE statement, or REBIND command.

Examples
Example 1
The following ALTER PROCEDURE statement updates version V2 of the UPDATE_BALANCE procedure.

```sql
ALTER PROCEDURE TEST.UPDATE_BALANCE
REPLACE VERSION V2
(IN CUSTOMER_NO INTEGER,
IN AMOUNT DECIMAL(9,2))
MODIFIES SQL DATA
ASUTIME LIMIT 100
BEGIN
UPDATE ACCOUNTS
SET BAL = BAL + AMOUNT
WHERE CUSTNO = CUSTOMER_NO
AND CUSTSTAT = 'A';
END
```

Example 2
**FL 507**
The following CREATE PROCEDURE statement will replace the version V2 of the UPDATE_BALANCE procedure if version V2 already exists or will create it if version V2 has not yet been defined:

```sql
CREATE OR REPLACE PROCEDURE TEST.UPDATE_BALANCE
(IN CUSTOMER_NO INTEGER,
IN AMOUNT DECIMAL(9,2))
VERSION V2
MODIFIES SQL DATA
ASUTIME LIMIT 100
BEGIN
UPDATE ACCOUNTS
SET BAL = BAL + AMOUNT
WHERE CUSTNO = CUSTOMER_NO
AND CUSTSTAT = 'A';
END
```

Related tasks
Creating new versions of native SQL procedures
A new version of a native SQL procedure can have different parameter names, procedure options, or procedure body.

**Related reference**
- REBIND PACKAGE (DSN) (Db2 Commands)
- ALTER PROCEDURE (SQL - native) (Db2 SQL)
- CREATE PROCEDURE (SQL - native) (Db2 SQL)

**Creating external stored procedures**

An **external stored procedure** is a procedure that is written in a host language and can contain SQL statements. The source code for external procedures is separate from the definition.

**Before you begin**

Before you create an external procedure, Configure Db2 for running stored procedures and user-defined functions during installation or Configure Db2 for running stored procedures and user-defined functions during migration.

**About this task**

**Restriction:** These instructions do not apply to Java stored procedures. The process for creating a Java stored procedure is different. The preparation process varies depending on what the procedure contains.

**Procedure**

To create an external stored procedure:

1. Write the external stored procedure body in assembler, C, C++, COBOL, REXX, or PL/I.

   Ensure that the procedure body that you write follows the guidelines for external stored procedures that are described in the following information:
   - “Accessing other sites in an external procedure” on page 277
   - “Accessing non-Db2 resources in your stored procedure” on page 277
   - “Writing an external procedure to access IMS databases” on page 278
   - “Writing an external procedure to return result sets to a distributed client” on page 279
   - “Restrictions when calling other programs from an external stored procedure” on page 280
   - “External stored procedures as main programs and subprograms” on page 281
   - “Data types in stored procedures” on page 283
   - “COMMIT and ROLLBACK statements in a stored procedure” on page 230

**Restrictions:**

- Do not include explicit attachment facility calls. External stored procedures that run in a WLM-established address space use Resource Recovery Services attachment facility (RRSAF) calls implicitly. If an external stored procedure makes an explicit attachment facility call, Db2 rejects the call.
- Do not include SRRCMIT or SRRBACK service calls. If an external stored procedure invokes either SRRCMIT or SRRBACK, Db2 puts the transaction in a state where a rollback operation is required and the CALL statement fails.

For REXX procedures, continue with step “3” on page 258.

2. For assembler, C, C++, COBOL, or PL/I stored procedures, prepare the external procedure by completing the following tasks:

   a) Precompile, compile, and link-edit the application by using one of the following techniques:

      - The Db2 precompiler and JCL instructions to compile and link-edit the program
The SQL statement coprocessor

**Recommendation:** Compile and link-edit code as reentrant.

Link-edit the application by using DSNRRI, the language interface module for the Resource Recovery Services attachment facility, or DSNULI, the Universal language interface module. You must specify the parameter AMODE(31) when you link-edit the application with either of these modules. (24-bit applications are not supported.)

If you want to make the stored procedure reentrant, see “Creating an external stored procedure as reentrant” on page 281.

If you want to run your procedure as a z/OS-authorized program, you must also perform the following tasks when you link-edit the application:

- Indicate that the load module can use restricted system services by specifying the parameter value AC=1.
- Put the load module for the stored procedure in an APF-authorized library.

You can compile COBOL stored procedures with either the DYNAM or NODYNAM COBOL compiler options. If you use DYNAM, ensure that the correct Db2 language interface module is loaded dynamically by performing one of the following actions:

- Specify the ATTACH(RRSAF) SQL processing option.
- Copy the DSNRRI module into a load library that is concatenated in front of the Db2 libraries. Use the member name DSNHLI.

b) Bind the DBRM into a Db2 package by issuing the BIND PACKAGE command.

If you want to control access to a stored procedure package, specify the ENABLE bind option with the system connection type of the calling application.

Stored procedures require only a package. You do not need to bind a plan for the stored procedure or bind the stored procedure package to the plan for the calling application. For remote access scenarios, you need a package at both the requester and server sites.

For more information about stored procedure packages, see “Packages for external stored procedures” on page 276.

The following example BIND PACKAGE command binds the DBRM EMPDTL1P to the collection DEV7083.

```sql
BIND PACKAGE(DEVL7083) -
MEMBER(EMPDTL1P) ACT(REP) ISO(UR) ENCODING(EBCDIC) -
OWNER(DEVL7083) LIBRARY('SG247083.DEVL.DBRM')
```

3. Define the stored procedure to Db2 by issuing the CREATE PROCEDURE statement with the EXTERNAL option. Use the EXTERNAL NAME clause to specify the name of the load module for the program that runs when this procedure is called.

If you want to run your procedure as a z/OS-authorized program, specify an appropriate environment with the WLM ENVIRONMENT option. The stored procedure must run in an address space with a startup procedure in which all libraries in the STEPLIB concatenation are APF-authorized.

If you want environment information to be passed to the stored procedure when it is invoked, specify the DBINFO and PARAMETER STYLE SQL options in the CREATE PROCEDURE statement. When the procedure is invoked, Db2 passes the DBINFO structure, which contains environment information, to the stored procedure. For more information about PARAMETER STYLE, see “Defining the linkage convention for an external stored procedure” on page 260.

If you compiled the stored procedure as reentrant, specify the STAY RESIDENT YES option in the CREATE PROCEDURE statement. This option makes the procedure remain resident in storage.

4. Authorize the appropriate users to use the stored procedure by issuing the GRANT EXECUTE statement.
For example, the following statement allows an application that runs under the authorization ID JONES to call stored procedure SPSchema.STORPRCA:

```
GRANT EXECUTE ON PROCEDURE SPSchema.STORPRCA TO JONES;
```

**Example of defining a C stored procedure**

Suppose that you have written and prepared a stored procedure that has the following characteristics:

- The name of the stored procedure is B.
- The stored procedure has the following two parameters:
  - An integer input parameter that is named V1
  - A character output parameter of length 9 that is named V2
- The stored procedure is written in the C language.
- The stored procedure contains no SQL statements.
- The same input always produces the same output.
- The load module name is SUMMOD.
- The package collection name is SUMCOLL.
- The stored procedure is to run for no more than 900 CPU service units.
- The parameters can have null values.
- The stored procedure is to be deleted from memory when it completes.
- The stored procedure needs the following Language Environment runtime options:

```
  MSGFILE(OUTFILE),RPTSTG(ON),RPTOPTS(ON)
```

- The stored procedure is part of the WLM application environment that is named PAYROLL.
- The stored procedure runs as a main program.
- The stored procedure does not access non-Db2 resources, so it does not need a special RACF environment.
- The stored procedure can return at most 10 result sets.
- When control returns to the client program, Db2 does not commit updates automatically.

The following CREATE PROCEDURE statement defines the stored procedure to Db2:

```
CREATE PROCEDURE B(IN V1 INTEGER, OUT V2 CHAR(9))
  LANGUAGE C
  DETERMINISTIC
  NO SQL
  EXTERNAL NAME SUMMOD
  COLLID SUMCOLL
  ASUTURE LIMIT 900
  PARAMETER STYLE GENERAL WITH NULLS
  STAY RESIDENT NO
  RUN OPTIONS 'MSGFILE(OUTFILE),RPTSTG(ON),RPTOPTS(ON)'
  WLM ENVIRONMENT PAYROLL
  PROGRAM TYPE MAIN
  SECURITY DB2
  DYNAMIC RESULT SETS 10
  COMMIT ON RETURN NO;
```

**What to do next**

You can now invoke the stored procedure from an application program or command line processor.

**Related concepts**

“Universal language interface (DSNULI)” on page 113

The universal language interface (DSNULI) subcomponent determines the runtime environment and dynamically loads and branches to the appropriate language interface module.

Java stored procedures and user-defined functions (Db2 Application Programming for Java)
Related tasks
Implementing Db2 stored procedures (Db2 Administration Guide)

Related reference
BIND and REBIND options for packages, plans, and services (Db2 Commands)
CREATE PROCEDURE (external) (Db2 SQL)
GRANT (function or procedure privileges) (Db2 SQL)

Related information
Db2 for z/OS Stored Procedures: Through the CALL and Beyond (IBM Redbooks)

Defining the linkage convention for an external stored procedure
A linkage convention specifies the rules for the parameter list that is passed by the program that calls the external stored procedure. For example, the convention can specify whether the calling program can pass null values for input parameters.

Procedure
When you define the stored procedure with the CREATE PROCEDURE statement, specify one of the following values for the PARAMETER STYLE option:

- GENERAL
- GENERAL WITH NULLS
- SQL

SQL is the default.

Linkage conventions for external stored procedures
The linkage convention for a stored procedure can be either GENERAL, GENERAL WITH NULLS, or SQL. These linkage conventions apply to only external stored procedures.

GENERAL
Specify the GENERAL linkage convention when you do not want the calling program to pass null values for input parameters (IN or INOUT) to the stored procedure. If you specify GENERAL, ensure that the stored procedure contains a variable declaration for each parameter that is passed in the CALL statement.

The following figure shows the structure of the parameter list for PARAMETER STYLE GENERAL.

![Diagram of Parameter convention GENERAL for a stored procedure]

GENERAL WITH NULLS
Specify the GENERAL WITH NULLS linkage convention when you want to allow the calling program to supply a null value for any parameter that is passed to the stored procedure. If you specify GENERAL WITH NULLS, ensure that the stored procedure performs the following tasks:

- Declares a variable for each parameter that is passed in the CALL statement.
- Declares a null indicator structure that contains an indicator variable for each parameter.
• On entry, examines all indicator variables that are associated with input parameters to determine which parameters contain null values.

• On exit, assigns values to all indicator variables that are associated with output variables. If the output variable returns a null value to the caller, assign the associated indicator variable a negative number. Otherwise, assign a value of 0 to the indicator variable.

In the CALL statement in the calling application, follow each parameter with its indicator variable. Use one of the following forms:

• `host-variable :indicator-variable`
• `host-variable INDICATOR :indicator-variable`

The following figure shows the structure of the parameter list for PARAMETER STYLE GENERAL WITH NULLS.

**Figure 13. Parameter convention GENERAL WITH NULLS for a stored procedure**

**SQL**

Specify the SQL linkage convention when you want both of the following conditions:

• The calling program to be able to supply a null value for any parameter that is passed to the stored procedure.

• Db2 to pass input and output parameters to the stored procedure that contain the following information:
  
  − The SQLSTATE that is to be returned to Db2. This value is a CHAR(5) parameter that represents the SQLSTATE that is passed into the program from the database manager. The initial value is set to '00000'. Although the SQLSTATE is usually not set by the program, it can be set as the result SQLSTATE that is used to return an error or a warning. Returned values that start with anything other than '00', '01', or '02' are error conditions.
  
  − The qualified name of the stored procedure. This is a VARCHAR(128) value.
  
  − The specific name of the stored procedure. The specific name is a VARCHAR(128) value that is the same as the unqualified name.
  
  − The SQL diagnostic string that is to be returned to Db2. This is a VARCHAR(1000) value. Use this area to pass descriptive information about an error or warning to the caller.

**Restriction:** You cannot use the SQL linkage convention for a REXX language stored procedure.

The following figure shows the structure of the parameter list for PARAMETER STYLE SQL.
Figure 14. Parameter convention SQL for a stored procedure

Related concepts
Example programs that call stored procedures
Examples can be used as models when you write applications that call stored procedures. In addition, prefix.SDNSAMP contains sample jobs DSNTEJ6P and DSNTEJ6S and programs DSN8EP1 and DSN8EP2, which you can run.

Related reference
CREATE PROCEDURE (external) (Db2 SQL)
SQLSTATE values and common error codes (Db2 Codes)

Example of GENERAL linkage convention
Specify the GENERAL linkage convention when you do not want the calling program to pass null values for input parameters (IN or INOUT) to the stored procedure.

Examples
The following examples demonstrate how an assembler, C, COBOL, or PL/I stored procedure uses the GENERAL linkage convention to receive parameters.

1 For PL/I, this value is the address of a pointer to the DBINFO data.
2 Passed if the DBINFO option is specified in the user-defined function definition.
For these examples, assume that a COBOL application has the following parameter declarations and CALL statement:

```
01 V1  PIC S9(9) USAGE COMP.
01 V2  PIC X(9).

EXEC SQL CALL A (:V1, :V2) END-EXEC.
```

In the CREATE PROCEDURE statement, the parameters are defined as follows:

```
IN V1 INT, OUT V2 CHAR(9)
```

**Assembler example**

The following example shows how a stored procedure that is written in assembler language receives these parameters.

```
* CODE FOR AN ASSEMBLER LANGUAGE STORED PROCEDURE THAT USES *
* THE GENERAL LINKAGE CONVENTION. *
******************************************************************************
A CEEENTRY AUTO=PROGSIZE,MAIN=YES,PLIST=OS USING PROGAREA,R13
******************************************************************************
* BRING UP THE LANGUAGE ENVIRONMENT. *
******************************************************************************
L R7,0(R1) GET POINTER TO V1
MVC LOCV1(4),0(R7) MOVE VALUE INTO LOCAL COPY OF V1
:
L R7,4(R1) GET POINTER TO V2
MVC 0(9,R7),LOCV2 MOVE A VALUE INTO OUTPUT VAR V2
:
CEETERM RC=0
******************************************************************************
* VARIABLE DECLARATIONS AND EQUATES *
******************************************************************************
R1 EQU 1 REGISTER 1
R7 EQU 7 REGISTER 7
PPA CEEPPA CONSTANTS DESCRIBING THE CODE BLOCK
LTORG PLACE LITERAL POOL HERE
PROGAREA DSECT
ORG **CEEDSAZ LEAVE SPACE FOR DSA FIXED PART
LOCV1 DS F LOCAL COPY OF PARAMETER V1
LOCV2 DS CL9 LOCAL COPY OF PARAMETER V2
PROGSIZE EQU **-PROGAREA
CEEDSA MAPPING OF THE DYNAMIC SAVE AREA
CEECAA MAPPING OF THE COMMON ANCHOR AREA
END A
```

**C example**

The following figure shows how a stored procedure that is written in the C language receives these parameters.

```
#include <stdlib.h>
#include <stdio.h>
/* Code for a C language stored procedure that uses the */
/* GENERAL linkage convention. */
******************************************************************************
```
main(argc,argv)
{
    int argc;                       /* Number of parameters passed */
    char *argv[];                   /* Array of strings containing */
            /* the parameter values */
    long int locv1;                 /* Local copy of V1            */
    char locv2[10];                 /* Local copy of V2            */
            /* (null-terminated)        */
    ...
    /* Get the passed parameters. The GENERAL linkage convention   */
    /* follows the standard C language parameter passing          */
    /* conventions:                                                */
    /* - argc contains the number of parameters passed             */
    /* - argv[1] to argv[n] are pointers to the n parameters      */
    /* in the SQL statement CALL.                                 */
    /***************************************************************/
    if(argc==3)                     /* Should get 3 parameters:    */
    {                               /* procname, V1, V2            */
        locv1 = *(int *) argv[1];  /* Get local copy of V1        */
        ...
        strcpy(argv[2],locv2);    /* Assign a value to V2        */
        ...
    }
    }

COBOL example
The following figure shows how a stored procedure that is written in the COBOL language receives these parameters.

CBL RENT
IDENTIFICATION DIVISION.
**************************************************************
* CODE FOR A COBOL LANGUAGE STORED PROCEDURE THAT USES THE *
* GENERAL LINKAGE CONVENTION.                               *
**************************************************************
PROGRAM-ID.    A.

DATA DIVISION.

LINKAGE SECTION.
**************************************************************
* DECLARE THE PARAMETERS PASSED BY THE SQL STATEMENT       *
* CALL HERE.                                              *
**************************************************************
01 V1  PIC S9(9) USAGE COMP.
01 V2  PIC X(9).

PROCEDURE DIVISION USING V1, V2.
**************************************************************
* THE USING PHRASE INDICATES THAT VARIABLES V1 AND V2      *
* WERE PASSED BY THE CALLING PROGRAM.                      *
**************************************************************

ASSIGN A VALUE TO OUTPUT VARIABLE V2 *
**************************************************************
MOVE '123456789' TO V2.

PL/I example
The following figure shows how a stored procedure that is written in the PL/I language receives these parameters.

*PROCESS SYSTEM(MVS);
A: PROC(V1, V2) OPTIONS(MAIN NOEXECOPS REENTRANT):
/** Code for a PL/I language stored procedure that uses the */
/** GENERAL linkage convention. */
/** Indicate on the PROCEDURE statement that two parameters */
/**
Example of GENERAL WITH NULLS linkage convention
Specify the GENERAL WITH NULLS linkage convention when you want to allow the calling program to supply a null value for any parameter that is passed to the stored procedure.

Examples
The following examples demonstrate how an assembler, C, COBOL, or PL/I stored procedure uses the GENERAL WITH NULLS linkage convention to receive parameters.

For these examples, assume that a C application has the following parameter declarations and CALL statement:

```c
DCL V1 BIN FIXED(31),
    V2 CHAR(9);
```

```c
V2 = '123456789';    /* Assign a value to output variable V2 */
```

In the CREATE PROCEDURE statement, the parameters are defined as follows:

```
IN V1 INT, OUT V2 CHAR(9)
```

Assembler example
The following figure shows how a stored procedure that is written in assembler language receives these parameters.

```
B        CEEENTRY AUTO=PROGSIZE,MAIN=YES,PLIST=OS
USING PROGAREA,R13
B        CEEENTRY AUTO=PROGSIZE,MAIN=YES,PLIST=OS
USING PROGAREA,R13
```

```
L     R7,0(R1)            GET POINTER TO V1
MVC   LOCV1(4),0(R7)      MOVE VALUE INTO LOCAL COPY OF V1
L     R7,8(R1)            GET POINTER TO INDICATOR ARRAY
```

```
R7,8(R1) GET POINTER TO V1
LOCV1(4),0(R7) MOVE VALUE INTO LOCAL COPY OF V1
R7,8(R1) GET POINTER TO INDICATOR ARRAY
```
C example

The following figure shows how a stored procedure that is written in the C language receives these parameters.

```c
#include <stdlib.h>
#include <stdio.h>

main(argc,argv)
int argc;                       /* Number of parameters passed */
char *argv[];                   /* Array of strings containing 
                              *
                              */
{                               /*  the parameter values */
  long int locv1;              /* Local copy of V1        */
  char locv2[10];               /* Local copy of V2        */
  short int locind[2];          /* Local copy of indicator */
  short int *tempint;          /* Used for receiving the */
                             /*  indicator variable array */
  if(argc==4)                     /* Should get 4 parameters: */
  {                             /*  procname, V1, V2, */
    locv1 = *(int *) argv[1];  /*  indicator variable array */
    tempint = argv[3];        /* Get local copy of V1 */
    locind[0] = *tempint;     /* Get pointer to indicator */
    locind[1] = *(++tempint); /* Get 1st indicator variable */
    if(locind[0]<0)               /* If 1st indicator variable */
    {                             /*  is negative, V1 is null */
      ...                         /*  */
    }
    ...                         /*  */
}
```
COBOL example

The following figure shows how a stored procedure that is written in the COBOL language receives these parameters.

```cobol
CBL RENT
IDENTIFICATION DIVISION.
************************************************************
* CODE FOR A COBOL LANGUAGE STORED PROCEDURE THAT USES THE *
* GENERAL WITH NULLS LINKAGE CONVENTION.                   *
************************************************************
PROGRAM-ID.    B.
:
DATA DIVISION.
:
LINKAGE SECTION.
************************************************************
* DECLARE THE PARAMETERS AND THE INDICATOR ARRAY THAT      *
*  WERE PASSED BY THE SQL STATEMENT CALL HERE.             *
************************************************************
01 V1  PIC S9(9) USAGE COMP.
01 V2  PIC X(9).
* 
01 INDARRAY.
10 INDVAR  PIC S9(4) USAGE COMP OCCURS 2 TIMES.
:
PROCEDURE DIVISION USING V1, V2, INDARRAY.
************************************************************
* THE USING PHRASE INDICATES THAT VARIABLES  V1, V2, AND   *
*  INDARRAY WERE PASSED BY THE CALLING PROGRAM.            *
************************************************************
:
:****************************************
* TEST WHETHER V1 IS NULL *
:****************************************
IF INDARRAY(1) < 0
   PERFORM NULL-PROCESSING.
:
:****************************************
* ASSIGN A VALUE TO OUTPUT VARIABLE V2 *
*  AND ITS INDICATOR VARIABLE          *
:****************************************
MOVE '123456789' TO V2.
MOVE ZERO TO INDARRAY(2).
```

PL/I example

The following figure shows how a stored procedure that is written in the PL/I language receives these parameters.

```pli
*PROCESS SYSTEM(MVS); 
A: PROC(V1, V2, INDSTRUC) OPTIONS(MAIN NOEXECOPS REENTRANT);
/*****************************/
/* Code for a PL/I language stored procedure that uses the */
/* GENERAL WITH NULLS linkage convention. */
/*****************************/
/* Indicate on the PROCEDURE statement that two parameters */
/* and an indicator variable structure were passed by the SQL */
/* statement CALL. Then declare them in the following section.*/
/* For PL/I, you must declare an indicator variable structure, */
/* not an array. */
/*****************************/
DCL V1 BIN FIXED(31),
    V2 CHAR(9);
DCL
   01 INDSTRUC,
   02 INDIV BIN FIXED(15),
```
Example of SQL linkage convention

Specify the SQL linkage convention when you want diagnostic information to be passed in the parameters and allow null values.

Examples

The following examples demonstrate how an assembler, C, COBOL, or PL/I stored procedure uses the SQL linkage convention to receive parameters. These examples also show how a stored procedure receives the DBINFO structure.

For these examples, assume that a C application has the following parameter declarations and CALL statement:

```c
/*****************************/
/* Parameters for the SQL statement CALL */
/*****************************/
long int v1;
char v2[10]; /* Allow an extra byte for */ /* the null terminator */
/*****************************/
/* Indicator variables */
/*****************************/
short int ind1;
short int ind2;
... ind1 = 0; /* Remember to initialize the */ /* input parameter's indicator */ /* variable before executing */ /* the CALL statement */
EXEC SQL CALL B (:v1 :ind1, :v2 :ind2);
```

In the CREATE PROCEDURE statement, the parameters are defined as follows:

```
IN V1 INT, OUT V2 CHAR(9)
```

Assembler example

The following figure shows how a stored procedure that is written in assembler language receives these parameters.

```
* CODE FOR AN ASSEMBLER LANGUAGE STORED PROCEDURE THAT USES
* THE SQL LINKAGE CONVENTION.
* CODE FOR A SQL LINKAGE CONVENTION. THE SQL LINKAGE
B CEEENTRY AUTO=PROGSIZE,MAIN=YES,PLIST=OS
USING PROGAREA,R13
* BRING UP THE LANGUAGE ENVIRONMENT.
*
* GET THE PASSED PARAMETER VALUES. THE SQL LINKAGE
* CONVENTION IS AS FOLLOWS:
* ON ENTRY, REGISTER 1 POINTS TO A LIST OF POINTERS. IF N
* PARAMETERS ARE PASSED, THERE ARE 2N+4 POINTERS. THE FIRST
* N POINTERS ARE THE ADDRESSES OF THE N PARAMETERS, JUST AS
```
WITH THE GENERAL LINKAGE CONVENTION. THE NEXT N POINTERS ARE THE ADDRESSES OF THE INDICATOR VARIABLE VALUES. THE LAST 4 POINTERS (5, IF DBINFO IS PASSED) ARE THE ADDRESSES OF INFORMATION ABOUT THE STORED PROCEDURE ENVIRONMENT AND EXECUTION RESULTS.

********************************************************************************
L     R7,0(R1)            GET POINTER TO V1
MVC   LOCV1(4),0(R7)      MOVE VALUE INTO LOCAL COPY OF V1
L     R7,8(R1)            GET POINTER TO 1ST INDICATOR VARIABLE
MVC   LOCI1(2),0(R7)      MOVE VALUE INTO LOCAL STORAGE
L     R7,20(R1)           GET POINTER TO STORED PROCEDURE NAME
MVC   LOCSPNM(20),0(R7)   MOVE VALUE INTO LOCAL STORAGE
LH    R7,LOCI1            GET INDICATOR VARIABLE FOR V1
LTR   R7,R7               CHECK IF IT IS NEGATIVE
BM    NULLIN              IF SO, V1 IS NULL
L     R7,4(R1)            GET POINTER TO V2
MVC   0(9,R7),LOCV2      MOVE A VALUE INTO OUTPUT VAR V2
L     R7,12(R1)           GET POINTER TO INDICATOR VAR 2
MVC   0(2,R7),=H'0'       MOVE ZERO TO V2'S INDICATOR VAR
L     R7,16(R1)           GET POINTER TO SQLSTATE
MVC   0(5,R7),=CL5'xxxxx' MOVE xxxxx TO SQLSTATE
CEETERM  RC=0

********************************************************************************

* VARIABLE DECLARATIONS AND EQUATES

R1 EQU 1                   REGISTER 1
R7 EQU 7                   REGISTER 7
PPA CEEPPA ,               CONSTANTS DESCRIBING THE CODE BLOCK
LTORG ,                   PLACE LITERAL POOL HERE
PROGAREA DSECT
ORG   +CEEDSASZ            LEAVE SPACE FOR DSA FIXED PART
LOCV1 DS F                 LOCAL COPY OF PARAMETER V1
LOCV2 DS CL9               LOCAL COPY OF PARAMETER V2
LOCI1 DS H                 LOCAL COPY OF INDICATOR 1
LOCSPNM DS H,CL27          LOCAL COPY OF STORED PROC NAME
LOCSPSNM DS H,CL18         LOCAL COPY OF SPECIFIC NAME
LOCPNAME DS H,CL1800       LOCAL COPY OF DIAGNOSTIC DATA
DBNAMELEN DS H             DATABASE NAME LENGTH
DBNAME DS CL128            DATABASE NAME
AUTHIDLS DS H              APPL AUTH ID LENGTH
AUTHID DS CL128            APPL AUTH ID
ASC_SBCS DS F              ASCII SBCS CCSID
ASC_DBCS DS F              ASCII DBCS CCSID
ASC_MIXD DS F              ASCII MIXED CCSID
EBC_SBCS DS F              EBCDIC SBCS CCSID
EBC_DBCS DS F              EBCDIC DBCS CCSID
EBC_MIXD DS F              EBCDIC MIXED CCSID
UNI_SBCS DS F              UNICODE SBCS CCSID
UNI_DBCS DS F              UNICODE DBCS CCSID
UNI_MIXD DS F              UNICODE MIXED CCSID
ENCODER DS F               PROCEDURE ENCODING SCHEME
RESERV0 DS CL20            RESERVED
TBQUALLEN DS H             TABLE QUALIFIER LENGTH
TBQUAL DS CL128            TABLE QUALIFIER
TBNAMELEN DS H             TABLE NAME LENGTH
TBNAME DS CL128            TABLE NAME
CLNAMELEN DS H             COLUMN NAME LENGTH
COLNAME DS CL128           COLUMN NAME
RELVER DS CL8              DBMS RELEASE AND VERSION
RESERV1 DS CL2             RESERVED
PLATFORM DS F              DBMS OPERATING SYSTEM
NUMTFCOL DS H              NUMBER OF TABLE FUNCTION COLS USED
RESERV2 DS CL26            RESERVED
TFCOLNUM DS A              POINTER TO TABLE FUNCTION COL LIST
APPLID DS A                POINTER TO APPLICATION ID
C example

The following figure shows how a stored procedure that is written as a main program in the C language receives these parameters.

```c
#include <stdlib.h>
#include <stdio.h>

main(argc, argv)
int argc;
char *argv[];
{
int parm1;
short int ind1;
char p_proc[28];
char p_spec[19];

/* Assume that the SQL CALL statement included */
/* 3 input/output parameters in the parameter list. */
/* The argv vector will contain these entries: */
/* argv[0] 1 contains load module */
/* argv[1-3] 3 input/output parms */
/* argv[4-6] 3 null indicators */
/* argv[7] 1 SQLSTATE variable */
/* argv[8] 1 qualified proc name */
/* argv[9] 1 specific proc name */
/* argv[10] 1 diagnostic string */

/* We end up here when invoked with wrong number of parms */

if argc<>12 {
/* We can access the null indicator for the first */
/* parameter on the SQL CALL as follows: */

parm1 = *(int *) argv[1];

/* We obtain the value of the qualified procedure */
/* name with this expression. */
strcpy(p_proc, argv[8]);

/* We obtain the value of the specific procedure */
/* name with this expression. */
strcpy(p_spec, argv[9]);

/* We can use the following expression to assign */
/* 'yyyyyyyy' to the diagnostic string returned */
/* in the SQLDA associated with the CALL statement. */
strcpy(argv[10], "yyyyyyyy/0");
```

The following figure shows how a stored procedure that is written as a subprogram in the C language receives these parameters.

```c
#include <stdlib.h>
#include <stdio.h>
#include <sqludf.h>

void myproc(*parm1 int,              /* assume INT for PARM1 */
    *parm2 char[11],         /* assume CHAR(10) parm2 */
    *p_ind1 short int,       /* null indicator for parm1 */
    *p_ind2 short int,       /* null indicator for parm2 */
    *p_sqlstate char[6],      /* SQLSTATE returned to DB2 */
    *p_proc char[28],         /* Qualified stored proc name */
    *p_spec char[19],         /* Specific stored proc name */
    *p_diag char[1001],       /* Diagnostic string */
    struct sqludf_dbinfo *udf_dbinfo);     /* DBINFO */

{ int l_p1;
  short int l_ind1;
  short int l_ind2;
  char[6] l_sqlstate;
  char[28] l_proc;
  char[19] l_spec;
  char[71] l_diag;
  sqludf_dbinfo *ludf_dbinfo;

  /* Copy each of the parameters in the parameter list into a local variable, just to demonstrate */
  /* how the parameters can be referenced. */

  l_p1 = *parm1;
  strcpy(l_p2, parm2);
  l_ind1 = *p_ind1;
  l_ind1 = *p_ind2;
  strcpy(l_sqlstate, p_sqlstate);
  strcpy(l_proc, p_proc);
  strcpy(l_spec, p_spec);
  strcpy(l_diag, p_diag);
  memcpy(&ludf_dbinfo, udf_dbinfo, sizeof(ludf_dbinfo));
}
```

**COBOL example**

The following figure shows how a stored procedure that is written in the COBOL language receives these parameters.

```
IDENTIFICATION DIVISION.
  DATA DIVISION.
  LINKAGE SECTION.
```

Chapter 3. Db2 SQL programming 271
* Declare each of the parameters
  01 PARM1 ...  
  01 PARM2 ...

* Declare a null indicator for each parameter
  01 P-IND1 PIC S9(4) USAGE COMP.  
  01 P-IND2 PIC S9(4) USAGE COMP.  

* Declare the SQLSTATE that can be set by stored proc
  01 P-SQLSTATE PIC X(5).

* Declare the qualified procedure name
  01 P-PROC.  
    49 P-PROC-LEN PIC 9(4) USAGE BINARY.  
    49 P-PROC-TEXT PIC X(27).

* Declare the specific procedure name
  01 P-SPEC.  
    49 P-SPEC-LEN PIC 9(4) USAGE BINARY.  
    49 P-SPEC-TEXT PIC X(18).

* Declare SQL diagnostic message token
  01 P-DIAG.  
    49 P-DIAG-LEN PIC 9(4) USAGE BINARY.  
    49 P-DIAG-TEXT PIC X(1000).

*********************************************************
* Structure used for DBINFO                             *
*********************************************************
01 SQLUDF-DBINFO.
  * Location name length
    05 DBNAMELEN PIC 9(4) USAGE BINARY.
  * Location name
    05 DBNAME PIC X(128).
  * authorization ID length
    05 AUTHIDLEN PIC 9(4) USAGE BINARY.
  * authorization ID
    05 AUTHID PIC X(128).
  * environment CCSID information
    05 CODEPG PIC X(48).
    10 DB2-CCSIDS OCCURS 3 TIMES.
      15 DB2-SBCS PIC 9(9) USAGE BINARY.
      15 DB2-DBCS PIC 9(9) USAGE BINARY.
      15 DB2-MIXED PIC 9(9) USAGE BINARY.
    10 ENCODING-SCHEME PIC 9(9) USAGE BINARY.
    10 RESERVED PIC X(20).

  * other platform-specific
    deprecated CCSID structures not included here
    05 TBSCHMALEN PIC 9(4) USAGE BINARY.
    * schema name
      05 TBSCHMA PIC X(128).
    * table name length
      05 TBNAMLEN PIC 9(4) USAGE BINARY.
    * table name
      05 TBNAM PIC X(128).
    * column name length
      05 COLNAMLEN PIC 9(4) USAGE BINARY.
    * column name
      05 COLNAM PIC X(128).
    * product information
      05 VER-REL PIC X(8).
    * reserved
      05 RESD0 PIC X(2).
    * platform type
      05 PLATFORM PIC 9(9) USAGE BINARY.
    * number of entries in the TF column list array (tfcolumn, below)
      05 NUMTFCOL PIC 9(4) USAGE BINARY.
    * reserved
      05 RESD1 PIC X(26).
    * tfcolumn will be allocated dynamically of it is defined
    * otherwise this will be a null pointer
      05 TFCOLUMN USAGE IS POINTER.
    * application identifier
      05 APPL-ID USAGE IS POINTER.
    * reserved
      05 RESD2 PIC X(20).

  PROCEDURE DIVISION USING PARM1, PARM2,  
                     P-IND1, P-IND2,  
                     P-SQLSTATE, P-PROC, P-SPEC, P-DIAG,
PL/I example

The following figure shows how a stored procedure that is written in the PL/I language receives these parameters.

```pli
*PROCESS SYSTEM(MVS);
MYMAIN: PROC(PARM1, PARM2, ..., P_IND1, P_IND2, ..., P_SQLSTATE, P_PROC, P_SPEC, P_DIAG, DBINFO)
OPTIONS(MAIN NOEXECOPS REENTRANT);

DCL PARM1 ... /* first parameter */
DCL PARM2 ... /* second parameter */
DCL P_IND1 BIN FIXED(15); /* indicator for 1st parm */
DCL P_IND2 BIN FIXED(15); /* indicator for 2nd parm */
DCL P_SQLSTATE CHAR(5); /* SQLSTATE to return to DB2 */
DCL P_PROC CHAR(27) /* Qualified procedure name */
VARYING;
DCL P_SPEC CHAR(18) /* Specific stored proc */
VARYING;
DCL P_DIAG CHAR(1000) /* Diagnostic string */
VARYING;
DCL DBINFO PTR;

DCL 01 SP_DBINFO BASED(DBINFO), /* Dbinfo */
    /* Dbinf */
    /* */
    03 UDF_DBINFO_LLEN BIN FIXED(15), /* location length */
    03 UDF_DBINFO_LOC CHAR(128), /* location name */
    03 UDF_DBINFO_ALEN BIN FIXED(15), /* auth ID length */
    03 UDF_DBINFO_AUTH CHAR(128), /* authorization ID */
    03 UDF_DBINFO_CCSID, /* CCSIDs for DB2 for z/OS */
    05 R1 BIN FIXED(15), /* Reserved */
    05 UDF_DBINFO_ASBCS BIN FIXED(15), /* ASCII SBCS CCSID */
    05 R2 BIN FIXED(15), /* Reserved */
    05 UDF_DBINFO_ADBCS BIN FIXED(15), /* ASCII DBCS CCSID */
    05 R3 BIN FIXED(15), /* Reserved */
    05 UDF_DBINFO_AMIXED BIN FIXED(15), /* ASCII MIXED CCSID */
    05 R4 BIN FIXED(15), /* Reserved */
    05 UDF_DBINFO_ESBCS BIN FIXED(15), /* EBCDIC SBCS CCSID */
    05 R5 BIN FIXED(15), /* Reserved */
    05 UDF_DBINFO_EDBCS BIN FIXED(15), /* EBCDIC DBCS CCSID */
    05 R6 BIN FIXED(15), /* Reserved */
    05 UDF_DBINFO_EMIXED BIN FIXED(15), /* EBCDIC MIXED CCSID */
    05 R7 BIN FIXED(15), /* Reserved */
    05 UDF_DBINFO_USBCS BIN FIXED(15), /* Unicode SBCS CCSID */
    05 R8 BIN FIXED(15), /* Reserved */
    05 UDF_DBINFO_UDBCS BIN FIXED(15), /* Unicode DBCS CCSID */
    */
    05 R9 BIN FIXED(15), /* Reserved */
    05 UDF_DBINFO_UDBCS BIN FIXED(15), /* Unicode DBCS CCSID */
    */
    05 R10 BIN FIXED(15), /* Reserved */
    05 UDF_DBINFO_MBCS BIN FIXED(15), /* Unicode BCS */
    05 UDF_DBINFO_ENCODE BIN FIXED(31), /* SP encode scheme */
    05 UDF_DBINFO_RESERV0 CHAR(2), /* reserved */
    03 UDF_DBINFO_SLEN BIN FIXED(15), /* schema length */
    03 UDF_DBINFO_SCHEMA CHAR(128), /* schema name */
    03 UDF_DBINFO_TLEN BIN FIXED(15), /* table length */
    03 UDF_DBINFO_TABLE CHAR(128), /* table name */
    03 UDF_DBINFO_CLEN BIN FIXED(15), /* column length */
    03 UDF_DBINFO_COLUMN CHAR(128), /* column name */
    03 UDF_DBINFO_RELVER CHAR(8), /* DB2 release level */
    03 UDF_DBINFO_RESERV0 CHAR(2), /* reserved */
    03 UDF_DBINFO_PLATFORM BIN FIXED(31), /* database platform */
    03 UDF_DBINFO_NUMTFCOL BIN FIXED(15), /* # of TF cols used */
    03 UDF_DBINFO_RESERV1 CHAR(26), /* reserved */
    03 UDF_DBINFO_TFCOLUMN PTR, /* -> table fun col list */
    03 UDF_DBINFO_APPLID PTR, /* -> application id */
    03 UDF_DBINFO_RESERV2 CHAR(20); /* reserved */
```

Chapter 3. Db2 SQL programming 273
**DBINFO structure**

Use the DBINFO structure to pass environment information to user-defined functions and stored procedures. Some fields in the structure are not used for stored procedures.

DBINFO is a structure that contains information such as the name of the current server, the application run time authorization ID and identification of the version and release of the database manager that invoked the procedure.

The DBINFO structure includes the following information:

**Location name length**
- An unsigned 2-byte integer field. It contains the length of the location name in the next field.

**Location name**
- A 128-byte character field. It contains the name of the location to which the invoker is currently connected.

**Authorization ID length**
- An unsigned 2-byte integer field. It contains the length of the authorization ID in the next field.

**Authorization ID**
- A 128-byte character field. It contains the authorization ID of the application from which the stored procedure is invoked, padded on the right with blanks. If this stored procedure is nested within other routines (user-defined functions or stored procedures), this value is the authorization ID of the application that invoked the highest-level routine.

**Subsystem code page**
- A 48-byte structure that consists of 10 integer fields and an eight-byte reserved area. These fields provide information about the CCSIDs of the subsystem from which the stored procedure is invoked.

**Table qualifier length**
- An unsigned 2-byte integer field. This field contains 0.

**Table qualifier**
- A 128-byte character field. This field is not used for stored procedures.

**Table name length**
- An unsigned 2-byte integer field. This field contains 0.

**Table name**
- A 128-byte character field. This field is not used for stored procedures.

**Column name length**
- An unsigned 2-byte integer field. This field contains 0.

**Column name**
- A 128-byte character field. This field is not used for stored procedures.

**Product information**
- An 8-byte character field that identifies the product on which the stored procedure executes.

The product identifier (PRDID) value is an 8-byte character value in `pppvvrrm` format, where: `ppp` is a 3-letter product code; `vv` is the version; `rr` is the release; and `m` is the modification level. In Db2 12 for z/OS, the modification level indicates a range of function levels:

- `DSN12015` for V12R1M500 or higher.
- `DSN12010` for V12R1M100.

For more information, see Product identifier (PRDID) values in Db2 for z/OS (Db2 Administration Guide).

**Reserved area**
- 2 bytes.

**Operating system**
- A 4-byte integer field. It identifies the operating system on which the program that invokes the user-defined function runs. The value is one of these:
0  Unknown
1  OS/2
3  Windows
4  AIX
5  Windows NT
6  HP-UX
7  Solaris
8  z/OS
13  Siemens Nixdorf
15  Windows 95
16  SCO UNIX
18  Linux
19  DYNIX/ptx
24  Linux for S/390
25  Linux for IBM Z
26  Linux/IA64
27  Linux/PPC
28  Linux/PPC64
29  Linux/AMD64
400  iSeries

**Number of entries in table function column list**
An unsigned 2-byte integer field. This field contains 0.

**Reserved area**
26 bytes.

**Table function column list pointer**
This field is not used for stored procedures.

**Unique application identifier**
This field is a pointer to a string that uniquely identifies the application's connection to Db2. The string is regenerated at for each connection to Db2.
The string is the LUWID, which consists of a fully-qualified LU network name followed by a period and an LUW instance number. The LU network name consists of a one- to eight-character network ID, a period, and a one- to eight-character network LU name. The LUW instance number consists of 12 hexadecimal characters that uniquely identify the unit of work.

**Reserved area**

20 bytes.

**Packages for external stored procedures**

An external stored procedure must have an associated package.

As part of the process of creating an external stored procedure, you prepare the procedure, which means that you precompile, compile, link-edit, and bind the application. The result of this process is a Db2 package. You do not need to create a Db2 plan for an external procedure. The procedure runs under the caller's thread and uses the plan from the client program that calls it.

The calling application can use a Db2 package or plan to execute the CALL statement.

Both the stored procedure package and the calling application plan or package must exist on the server before you run the calling application.

The following figure shows this relationship between a client program and a stored procedure. In the figure, the client program, which was bound into package A, issues a CALL statement to program B. Program B is an external stored procedure in a WLM address space. This external stored procedure was bound into package B.

![Figure 15. Stored procedure run time environment](image)

You can control access to the stored procedure package by specifying the ENABLE bind option when you bind the package.

In the following situations, the stored procedure might use more than one package:

- You bind a DBRM several times into several versions of the same package, all of which have the same package name but reside in different package collections. Your stored procedure can switch from one version to another by using the SET CURRENT PACKAGESET statement.
- The stored procedure calls another program that contains SQL statements. This program has an associated package. This package must exist at the location where the stored procedure is defined and at the location where the SQL statements are executed.

**Related reference**

BIND and REBIND options for packages, plans, and services (Db2 Commands)
Accessing other sites in an external procedure
External procedures can access tables at other Db2 locations.

About this task
Stored procedures can access tables at other Db2 locations by using three-part object names or CONNECT statements.

Related concepts
Accessing distributed data by using three-part table names
You can use three-part table names to access data at a remote location through DRDA access.

Accessing non-Db2 resources in your stored procedure
Applications that run in a stored procedures address space can access any resources that are available to z/OS address spaces. For example, they can access VSAM files, flat files, APPC/MVS conversations, and IMS or CICS transactions.

About this task
Accessing these resources from a stored procedure can be useful if you want to update older applications. Suppose that you have existing applications that access non-Db2 resources, but you want to use newer Db2 applications to access the same data. You do not need to rewrite the application or migrate the data to Db2. Instead, you can use stored procedures to execute the existing program or access the non-Db2 data directly.

When a stored procedure runs, the stored procedure uses the Recoverable Resource Manager Services (RRS) for commitment control. When Db2 commits or rolls back work, Db2 coordinates all updates that are made to recoverable resources by other RRS compliant resource managers in the z/OS system.

Procedure
To access non-Db2 resources in your stored procedure:
1. Consider serializing access to non-Db2 resources within your application.
   Not all non-Db2 resources can tolerate concurrent access by multiple TCBs in the same address space.
2. To access CICS, use one of the following methods:
   • Stored procedure DSNACICS
   • Message Queue Interface (MQI) for asynchronous execution of CICS transactions
   • External CICS interface (EXCI) for synchronous execution of CICS transactions
   • Advanced Program-to-Program Communication (APPC), using the Common Programming Interface Communications (CPI Communications) application programming interface
     If your system is running a release of CICS that uses z/OS RRS, z/OS RRS controls commitment of all resources.
3. To access IMS DL/I data, use one of the following methods
   • Open Database Access interface (ODBA)
   • Stored procedures DSNAIMS and DSNAIMS2
     If your system is not running a release of IMS that uses z/OS RRS, take one of the following actions:
     • Use the CICS EXCI interface to run a CICS transaction synchronously. That CICS transaction can, in turn, access DL/I data.
     • Invoke IMS transactions asynchronously using the MQI.
• Use APPC through the Common Programming Interface (CPI) Communications application programming interface.

4. Determine which of the following authorization IDs you want to use to access the non-Db2 resources.

<table>
<thead>
<tr>
<th>ID that you want to use to access the non-Db2 resources</th>
<th>SECURITY value to specify in the CREATE PROCEDURE statement</th>
</tr>
</thead>
<tbody>
<tr>
<td>The authorization ID that is associated with the stored procedures address space</td>
<td>SECURITY Db2</td>
</tr>
<tr>
<td>The authorization ID under which the CALL statement is executed</td>
<td>SECURITY USER</td>
</tr>
<tr>
<td>The authorization ID under which the CREATE PROCEDURE statement is executed</td>
<td>SECURITY DEFINER</td>
</tr>
</tbody>
</table>

5. Issue the CREATE PROCEDURE statement with the appropriate SECURITY option that you determined in the previous step.

Results
When the stored procedure runs, Db2 establishes a RACF environment for accessing non-Db2 resources and uses the specified authorization ID to access protected z/OS resources.

Related tasks
Calling a stored procedure from your application
To run a stored procedure, you can either call it from a client program or invoke it from the command line processor.

Implementing RRS for stored procedures during installation (Db2 Installation and Migration)
Controlling stored procedure access to non-Db2 resources by using RACF (Managing Security)

Related reference
DSNACICS stored procedure (Db2 SQL)
DSNAIMS stored procedure (Db2 SQL)
DSNAIMS2 stored procedure (Db2 SQL)
CREATE PROCEDURE (SQL - external) (deprecated) (Db2 SQL)
APPC/MVS Configuration (Multiplatform APPC Configuration Guide)

Related information
Db2 for z/OS Stored Procedures: Through the CALL and Beyond (IBM Redbooks)
External CICS interface (EXCI) (CICS Transaction Server for z/OS)

Writing an external procedure to access IMS databases
IMS Open Database Access (ODBA) support lets a Db2 stored procedure connect to an IMS DBCTL or IMS DB/DC system and issue DL/I calls to access IMS databases.

About this task
ODBA support uses RRS for syncpoint control of Db2 and IMS resources. Therefore, stored procedures that use ODBA can run only in WLM-established stored procedures address spaces.

When you write a stored procedure that uses ODBA, follow the rules for writing an IMS application program that issues DL/I calls.

IMS work that is performed in a stored procedure is in the same commit scope as the stored procedure. As with any other stored procedure, the calling application commits work.

A stored procedure that uses ODBA must issue a DPSB PREP call to deallocate a PSB when all IMS work under that PSB is complete. The PREP keyword tells IMS to move inflight work to an indoubt state. When work is in the indoubt state, IMS does not require activation of syncpoint processing when the DPSB
call is executed. IMS commits or backs out the work as part of RRS two-phase commit when the stored procedure caller executes COMMIT or ROLLBACK.

A sample COBOL stored procedure and client program demonstrate accessing IMS data using the ODBA interface. The stored procedure source code is in member DSN8EC1 and is prepared by job DSNTEJ61. The calling program source code is in member DSN8EC1 and is prepared and executed by job DSNTEJ62. All code is in data set DSN1210.SDSNSAMP.

The startup procedure for a stored procedures address space in which stored procedures that use ODBA run must include a DFSRESLB DD statement and an extra data set in the STEPLIB concatenation.

Related concepts
Installation step 21: Configure Db2 for running stored procedures and user-defined functions (Db2 Installation and Migration)
Migration step 23: Configure Db2 for running stored procedures and user-defined functions (optional) (Db2 Installation and Migration)

Related information
Application programming design

Writing an external procedure to return result sets to a distributed client
An external procedure can return multiple query result sets to a distributed client if the value of DYNAMIC RESULT SETS in the stored procedure definition is greater than 0.

Procedure
• For each result set you want returned, your stored procedure must complete the following steps:
  a) Declare a cursor with the option WITH RETURN.
  b) Open the cursor.
  c) If the cursor is scrollable, ensure that the cursor is positioned before the first row of the result table.
  d) Leave the cursor open.
For example, suppose you want to return a result set that contains entries for all employees in department D11. First, declare a cursor that describes this subset of employees:

```sql
EXEC SQL DECLARE C1 CURSOR WITH RETURN FOR
SELECT * FROM DSN8C10.EMP
WHERE WORKDEPT='D11';
```

Then, open the cursor:

```sql
EXEC SQL OPEN C1;
```

Db2 returns the result set and the name of the SQL cursor for the stored procedure to the client.

When the stored procedure ends, Db2 returns the rows in the query result set to the client.

Db2 does not return result sets for cursors that are closed before the stored procedure terminates. The stored procedure must execute a CLOSE statement for each cursor associated with a result set that should not be returned to the DRDA client.

• Use meaningful cursor names for returning result sets.
  The name of the cursor that is used to return result sets is made available to the client application through extensions to the DESCRIBE statement.
  Use cursor names that are meaningful to the DRDA client application, especially when the stored procedure returns multiple result sets.

• You can use any of these objects in the SELECT statement that is associated with the cursor for a result set:
Tables, synonyms, views, created temporary tables, declared temporary tables, and aliases defined at
the local Db2 subsystem.

- Return a subset of rows to the client by issuing FETCH statements with a result set cursor. does not
return the fetched rows to the client program.

Db2 does not return the fetched rows to the client program. For example, if you declare a cursor
WITH RETURN and then execute the statements OPEN, FETCH, and FETCH, the client receives data
beginning with the third row in the result set. If the result set cursor is scrollable and you fetch rows
with it, you need to position the cursor before the first row of the result table after you fetch the rows
and before the stored procedure ends.

- You can use a created temporary table or declared temporary table to return result sets from a stored
procedure.

This capability can be used to return non-relational data to a DRDA client. For example, you can access
IMS data from a stored procedure by using the following process:

a) Use APPC/MVS to issue an IMS transaction.
b) Receive the IMS reply message, which contains data that should be returned to the client.
c) Insert the data from the reply message into a temporary table.
d) Open a cursor against the temporary table. When the stored procedure ends, the rows from the
temporary table are returned to the client.

Related tasks
Writing a program to receive the result sets from a stored procedure
You can write a program to receive results set from a stored procedure for either a fixed number of result
sets, for which you know the contents, or a variable number of result sets, for which you do not know the
contents.

Restrictions when calling other programs from an external stored procedure
An external procedure can consist of more than one program, each with its own package. Your stored
procedure can call other programs, stored procedures, or user-defined functions. Use the facilities of your
programming language to call other programs.

If the stored procedure calls other programs that contain SQL statements, each of those called programs
must have a Db2 package. The owner of the package or plan that contains the CALL statement must have
EXECUTE authority for all packages that the other programs use.

When a stored procedure calls another program, Db2 determines which collection the package of the
called program belongs to in one of the following ways:

- If the stored procedure definition contains PACKAGE PATH with a specified list of collection IDs, Db2
uses those collection IDs. If you also specify COLLID, Db2 ignores that clause.
- If the stored procedure definition contains COLLID collection-id, Db2 uses collection-id.
- If the stored procedure executes SET CURRENT PACKAGE PATH and contains the NO COLLID option,
Db2 uses the CURRENT PACKAGE PATH special register. The package of the called program comes
from the list of collections in the CURRENT PACKAGE PATH special register. For example, assume that
CURRENT PACKAGE PATH contains the list COLL1, COLL2, COLL3, COLL4. Db2 searches for the first
package (in the order of the list) that exists in these collections.
- If the stored procedure does not execute SET CURRENT PACKAGE PATH and instead executes SET
CURRENT PACKAGESET, Db2 uses the CURRENT PACKAGESET special register. The package of the
called program comes from the collection that is specified in the CURRENT PACKAGESET special
register.
- If both of the following conditions are true, Db2 uses the collection ID of the package that contains the
SQL statement CALL:
  - the stored procedure does not execute SET CURRENT PACKAGE PATH or SET CURRENT PACKAGESET
  - the stored procedure definition contains the NO COLLID option
When control returns from the stored procedure, the value of the CURRENT PACKAGESET special register is reset. Db2 restores the value of the CURRENT PACKAGESET special register to the value that it contained before the client program executed the SQL statement CALL.

**Creating an external stored procedure as reentrant**

Reentrant code is code for which a single copy can be used concurrently by two or more processes. For improved performance, prepare your stored procedures to be reentrant whenever possible.

**About this task**

Reentrant stored procedures can improve performance for the following reasons:

- A reentrant stored procedure does not need to be loaded into storage every time that it is called.
- A single copy of the stored procedure can be shared by multiple tasks in the stored procedures address space. This sharing decreases the amount of virtual storage that is used for code in the stored procedures address space.

**Procedure**

To create an external stored procedure as reentrant:

1. Compile the procedure as reentrant and link-edit it as reentrant and reusable.
   
   For instructions on compiling programs to be reentrant, see the information for the programming language that you are using. For C and C++ procedures, you can use the z/OS binder to produce reentrant and reusable load modules.
   
   If your stored procedure cannot be reentrant, link-edit it as non-reentrant and non-reusable. The non-reusable attribute prevents multiple tasks from using a single copy of the stored procedure at the same time.

2. Specify STAY RESIDENT YES in the CREATE PROCEDURE or ALTER PROCEDURE statement for the stored procedure. This option makes a reentrant stored procedure remain in storage.
   
   A non-reentrant stored procedure must not remain in storage. You therefore need to specify STAY RESIDENT NO in the CREATE PROCEDURE or ALTER PROCEDURE statement for a non-reentrant stored procedure. STAY RESIDENT NO is the default.

**Related concepts**

Making programs reentrant (Enterprise COBOL for z/OS Programming Guide)

**Related reference**

Compiler options (COBOL) (Enterprise COBOL for z/OS Programming Guide)

ALTER PROCEDURE (external) (Db2 SQL)

CREATE PROCEDURE (external) (Db2 SQL)

Binder options reference (MVS Program Management: User’s Guide and Reference)

Language restricted (Enterprise PL/I for z/OS Compiler and Runtime Migration Guide)

Compile-time option descriptions (PL/I) (Enterprise PL/I for z/OS Programming Guide)

Reentrancy (XL C/C++ User’s Guide)

**External stored procedures as main programs and subprograms**

A stored procedure that runs in a WLM-established address space and uses Language Environment Release 1.7 or a subsequent release can be either a main program or a subprogram. A stored procedure that runs as a subprogram can perform better because Language Environment does less processing for it.

In general, a subprogram must do the following extra tasks that Language Environment performs for a main program:

- Initialization and cleanup processing
- Allocating and freeing storage
- Closing all open files before exiting
When you code stored procedures as subprograms, follow these rules:

- Follow the language rules for a subprogram. For example, you cannot perform I/O operations in a PL/I subprogram.
- Avoid using statements that terminate the Language Environment enclave when the program ends. Examples of such statements are STOP or EXIT in a PL/I subprogram, or STOP RUN in a COBOL subprogram. If the enclave terminates when a stored procedure ends, and the client program calls another stored procedure that runs as a subprogram, Language Environment must build a new enclave. As a result, the benefits of coding a stored procedure as a subprogram are lost.
- In COBOL stored procedures that are defined as PROGRAM TYPE SUB and STAY RESIDENT YES, if you use stored procedure parameters as host variables, set the SQL-INIT-FLAG variable to 0. This variable is generated by the Db2 precompiler. Setting it to 0 ensures that the SQLDA is updated with the current addresses.

The following table summarizes the characteristics that define a main program and a subprogram.

<table>
<thead>
<tr>
<th>Language</th>
<th>Main program</th>
<th>Subprogram</th>
</tr>
</thead>
<tbody>
<tr>
<td>Assembler</td>
<td>MAIN=YES is specified in the invocation of the CEEENTRY macro.</td>
<td>MAIN=NO is specified in the invocation of the CEEENTRY macro.</td>
</tr>
<tr>
<td>C</td>
<td>Contains a main() function. Pass parameters to it through argc and argv.</td>
<td>A fetchable function. Pass parameters to it explicitly.</td>
</tr>
<tr>
<td>COBOL</td>
<td>A COBOL program that ends with GOBACK</td>
<td>A dynamically loaded subprogram that ends with GOBACK</td>
</tr>
<tr>
<td>PL/I</td>
<td>Contains a procedure declared with OPTIONS(MAIN)</td>
<td>A procedure declared with OPTIONS(FETCHABLE)</td>
</tr>
</tbody>
</table>

The following code shows an example of coding a C stored procedure as a subprogram.

```c
/* This C subprogram is a stored procedure that uses linkage */
/* convention GENERAL and receives 3 parameters. */
#include <stdlib.h>
void cfunc(char p1[11], long *p2, short *p3)
{
    EXEC SQL BEGIN DECLARE SECTION;
    char parm1[11];
    long int parm2;
    short int parm3;
    EXEC SQL END DECLARE SECTION;

    strcpy(parm1, p1);
    parm2 = *p2;
    parm3 = *p3;

    /* Perform operations on local variables. */
}
```

282 Db2 12 for z/OS: Application Programming and SQL Guide
The following code shows an example of coding a C++ stored procedure as a subprogram.

```c++
extern  "C"  void cppfunc(char p1[11], long *p2, short *p3);
#pragma linkage(cppfunc, fetchable)
#include <stdlib.h>
EXEC SQL INCLUDE SQLCA;
void cppfunc(char p1[11], long *p2, short *p3)
{
    /* Declare variables used for SQL operations. These variables */
    /* are local to the subprogram and must be copied to and from */
    /* the parameter list for the stored procedure call. */
    EXEC SQL BEGIN DECLARE SECTION;
    char parm1[11];
    long int parm2;
    short int parm3;
    EXEC SQL END DECLARE SECTION;
    /* Receive input parameter values into local variables. */
    strcpy(parm1, p1);
    parm2 = *p2;
    parm3 = *p3;
    /* Perform operations on local variables. */
    :
    /* Set values to be passed back to the caller. */
    strcpy(parm1, "SETBYSP");
    parm2 = 100;
    parm3 = 200;
    /* Copy values to output parameters. */
    strcpy(p1, parm1);
    *p2 = parm2;
    *p3 = parm3;
}
```

**Data types in stored procedures**

A stored procedure that is written in any language except REXX must declare each parameter that is passed to it. The definition for that stored procedure must also contain a compatible SQL data type declaration for each parameter.

**For languages other than REXX**

For all data types except LOBs, ROWIDs, locators, and VARCHARs (for C language), see the tables listed in the following table for the host data types that are compatible with the data types in the stored procedure definition. You cannot have XML parameters in an external procedure.

For LOBs, ROWIDs, VARCHARs, and locators, the following table shows compatible declarations for the assembler language.
<table>
<thead>
<tr>
<th>SQL data type in definition</th>
<th>Assembler declaration</th>
</tr>
</thead>
<tbody>
<tr>
<td>TABLE LOCATOR</td>
<td>DS FL4</td>
</tr>
<tr>
<td>BLOB LOCATOR</td>
<td></td>
</tr>
<tr>
<td>CLOB LOCATOR</td>
<td></td>
</tr>
<tr>
<td>DBCLOB LOCATOR</td>
<td></td>
</tr>
</tbody>
</table>

**BLOB(n)**
- If \( n \leq 65535 \):
  - var DS 0FL4
  - var_length DS FL4
  - var_data DS CLn
  - If \( n > 65535 \):
    - var DS 0FL4
    - var_length DS FL4
    - var_data DS CL65535
    - ORG var_data+\( (n-65535) \)

**CLOB(n)**
- If \( n \leq 65535 \):
  - var DS 0FL4
  - var_length DS FL4
  - var_data DS CLn
  - If \( n > 65535 \):
    - var DS 0FL4
    - var_length DS FL4
    - var_data DS CL65535
    - ORG var_data+\( (n-65535) \)

**DBCLOB(n)**
- If \( m (=2\times n) \leq 65534 \):
  - var DS 0FL4
  - var_length DS FL4
  - var_data DS CLm
  - If \( m > 65534 \):
    - var DS 0FL4
    - var_length DS FL4
    - var_data DS CL65534
    - ORG var_data+\( (m-65534) \)

**ROWID**
- DS HL2,CL40

**VARCHAR(n)**
- If PARAMETER VARCHAR NULTERM is specified or implied:
  - char data[n+1];
- If PARAMETER VARCHAR STRUCTURE is specified:
  - struct
    - short len;
    - char data[n];
  - var;

**Note:**
1. This row does not apply to VARCHAR\( (n) \) FOR BIT DATA. BIT DATA is always passed in a structured representation.

For LOBs, ROWIDs, and locators, the following table shows compatible declarations for the C language.
### Table 53. Compatible C language declarations for LOBs, ROWIDs, and locators

<table>
<thead>
<tr>
<th>SQL data type in definition</th>
<th>C declaration</th>
</tr>
</thead>
<tbody>
<tr>
<td>TABLE LOCATOR</td>
<td>unsigned long</td>
</tr>
<tr>
<td>BLOB LOCATOR</td>
<td></td>
</tr>
<tr>
<td>CLOB LOCATOR</td>
<td></td>
</tr>
<tr>
<td>DBCLOB LOCATOR</td>
<td></td>
</tr>
</tbody>
</table>

**BLOB(n)**
```c
struct
{unsigned long length;
 char data[n];
} var;
```

**CLOB(n)**
```c
struct
{unsigned long length;
 char var_data[n];
} var;
```

**DBCLOB(n)**
```c
struct
{unsigned long length;
 sqlldbchar data[n];
} var;
```

**ROWID**
```c
struct
{short int length;
 char data[40];
} var;
```

For LOBs, ROWIDs, and locators, the following table shows compatible declarations for COBOL.

### Table 54. Compatible COBOL declarations for LOBs, ROWIDs, and locators

<table>
<thead>
<tr>
<th>SQL data type in definition</th>
<th>COBOL declaration</th>
</tr>
</thead>
<tbody>
<tr>
<td>TABLE LOCATOR</td>
<td></td>
</tr>
<tr>
<td>BLOB LOCATOR</td>
<td></td>
</tr>
<tr>
<td>CLOB LOCATOR</td>
<td></td>
</tr>
<tr>
<td>DBCLOB LOCATOR</td>
<td></td>
</tr>
</tbody>
</table>

**BLOB(n)**
01 var PIC S9(9) COMP-5.

**CLOB(n)**
01 var.

```cobol
49 var-LENGTH PIC S9(9) COMP-5.
49 var-DATA PIC X(n).
```

**DBCLOB(n)**
01 var.

```cobol
49 var-LENGTH PIC S9(9) COMP-5.
49 var-DATA PIC G(n) DISPLAY-1.
```

**ROWID**
01 var.

```cobol
49 var-LEN PIC S9(4) COMP-5.
49 var-DATA PIC X(40).
```

For LOBs, ROWIDs, and locators, the following table shows compatible declarations for PL/I.
<table>
<thead>
<tr>
<th>SQL data type in definition</th>
<th>PL/I</th>
</tr>
</thead>
<tbody>
<tr>
<td>TABLE LOCATOR</td>
<td>BIN FIXED(31)</td>
</tr>
</tbody>
</table>
| BLOB LOCATOR                | If n <= 32767:  
01 var,  
  03 var_LENGTH  
    BIN FIXED(31),  
  03 var_DATA  
    CHAR(n); |
| CLOB LOCATOR                | If n <= 32767:  
01 var,  
  03 var_LENGTH  
    BIN FIXED(31),  
  03 var_DATA  
    CHAR(n); |
| DBCLOB LOCATOR              | If n <= 16383:  
01 var,  
  03 var_LENGTH  
    BIN FIXED(31),  
  03 var_DATA  
    GRAPHIC(n); |
| ROWID                       | CHAR(40) VAR |

BLOB(n)  
If n > 32767:  
01 var,  
  02 var_LENGTH  
    BIN FIXED(31),  
  02 var_DATA,  
    03 var_DATA1(n)  
      CHAR(32767),  
    03 var_DATA2  
      CHAR(mod(n,32767)); |

CLOB(n)  
If n > 32767:  
01 var,  
  02 var_LENGTH  
    BIN FIXED(31),  
  02 var_DATA,  
    03 var_DATA1(n)  
      CHAR(32767),  
    03 var_DATA2  
      CHAR(mod(n,32767)); |

DBCLOB(n)  
If n > 16383:  
01 var,  
  02 var_LENGTH  
    BIN FIXED(31),  
  02 var_DATA,  
    03 var_DATA1(n)  
      GRAPHIC(16383),  
    03 var DATA2  
      GRAPHIC(mod(n,16383)); |
Tables of results: Each high-level language definition for stored procedure parameters supports only a single instance (a scalar value) of the parameter. There is no support for structure, array, or vector parameters. Because of this, the SQL statement CALL limits the ability of an application to return some kinds of tables. For example, an application might need to return a table that represents multiple occurrences of one or more of the parameters passed to the stored procedure. Because the SQL statement CALL cannot return more than one set of parameters, use one of the following techniques to return such a table:

- Put the data that the application returns in a Db2 table. The calling program can receive the data in one of these ways:
  - The calling program can fetch the rows from the table directly. Specify FOR FETCH ONLY or FOR READ ONLY on the SELECT statement that retrieves data from the table. A block fetch can retrieve the required data efficiently.
  - The stored procedure can return the contents of the table as a result set. See “Writing an external procedure to return result sets to a distributed client” on page 279 and “Writing a program to receive the result sets from a stored procedure” on page 768 for more information.
- Convert tabular data to string format and return it as a character string parameter to the calling program. The calling program and the stored procedure can establish a convention for interpreting the content of the character string. For example, the SQL statement CALL can pass a 1920-byte character string parameter to a stored procedure, which enables the stored procedure to return a 24x80 screen image to the calling program.

Related concepts
Compatibility of SQL and language data types
The host variable data types that are used in SQL statements must be compatible with the data types of the columns with which you intend to use them.

Installation step 21: Configure Db2 for running stored procedures and user-defined functions (Db2 Installation and Migration)
Migration step 23: Configure Db2 for running stored procedures and user-defined functions (optional) (Db2 Installation and Migration)

REXX stored procedures
A REXX stored procedure is similar to any other REXX procedure and follows the same rules as stored procedures in other languages. A REXX stored procedure receives input parameters, executes REXX commands, optionally executes SQL statements, and returns at most one output parameter. However, a few differences exist.

A REXX stored procedure is different from other REXX procedures in the following ways:

- A REXX stored procedure must not execute any of the following DSNREXX commands that are used for the Db2 subsystem thread attachment:
  ADDRESS DSNREXX CONNECT
  ADDRESS DSNREXX DISCONNECT
  CALL SQLDBS ATTACH TO
  CALL SQLDBS DETACH

  When you execute SQL statements in your stored procedure, Db2 establishes the connection for you.
- A REXX stored procedure must run in a WLM-established stored procedures address space.
- A language REXX stored procedure executes in a background TSO/E REXX environment provided by the TSO/E environment service IKJTSOEV.

Unlike other stored procedures, you do not prepare REXX stored procedures for execution. REXX stored procedures run using one of four packages that are bound during the installation of Db2 REXX Language Support. The current isolation level at which the stored procedure runs depends on the package that Db2 uses when the stored procedure runs:

Package name
Isolation level
This topic shows an example of a REXX stored procedure that executes Db2 commands. The stored procedure performs the following actions:

- Receives one input parameter, which contains a Db2 command.
- Calls the IFI COMMAND function to execute the command.
- Extracts the command result messages from the IFI return area and places the messages in a created temporary table. Each row of the temporary table contains a sequence number and the text of one message.
- Opens a cursor to return a result set that contains the command result messages.
- Returns the unformatted contents of the IFI return area in an output parameter.

The following example shows the definition of the stored procedure.

```rexx
CREATE PROCEDURE COMMAND(IN CMDTEXT VARCHAR(254), OUT CMDRESULT VARCHAR(32704)) LANGUAGE REXX EXTERNAL NAME COMMAND NO COLLID ASUTIME NO LIMIT PARAMETER STYLE GENERAL STAY RESIDENT NO RUN OPTIONS 'TRAP(ON)' WLM ENVIRONMENT WLMENV1 SECURITY DB2 DYNAMIC RESULT SETS 1 COMMIT ON RETURN NO;
```

The following example shows the COMMAND stored procedure that executes Db2 commands.

```rexx
/* REXX */
PARSE UPPER ARG CMD /* Get the DB2 command text */ /* Remove enclosing quotation marks */
IF LEFT(CMD,1) = "" & RIGHT(CMD,1) = "" THEN
  CMD = SUBSTR(CMD,2,LENGTH(CMD)-2)
ELSE
  IF LEFT(CMD,1) = '"' & RIGHT(CMD,1) = '"' THEN
    CMD = SUBSTR(CMD,2,LENGTH(CMD)-2)
  COMMAND = SUBSTR("COMMAND",1,18," ")
  /****************************************************************/
  /*  Set up the IFCA, return area, and output area for the       */
  /*  IFI COMMAND call.                                           */
  /****************************************************************/
  IFCA = SUBSTR('00'X,1,180,'00'X)
  IFCA = OVERLAY(D2C(LENGTH(IFCA),2),IFCA,1+0)
  IFCA = OVERLAY("IFCA",IFCA,4+1)
  RTRNAREASIZE = 262144 /*1048572*/
  RTRNAREA = D2C(RTRNAREASIZE+4,4)LEFT(' ',RTRNAREASIZE,' ')
  OUTPUT = D2C(LENGTH(CMD)+4,2)||'0000'X||CMD
  BUFFER = SUBSTR(" ",1,16," ")
  /****************************************************************/
  /*  Make the IFI COMMAND call.                                 */
  /****************************************************************/
  ADDRESS LINKPGM "DSNWLIIR COMMAND IFCA RTRNAREA OUTPUT"
  WRC = RC
  RTRN= SUBSTR(IFCA,12+1,4)
  RES= SUBSTR(IFCA,16+1,4)
  TOTLEN = C2D(SUBSTR(IFCA,20+1,4))
  /****************************************************************/
  /*  Set up the host command environment for SQL calls.         */
  /****************************************************************/
  "SUBCOM DSNREXX" /* Host cmd env available? */
```
IF RC THEN                             /* No--add host cmd env     */
S_RC = RXSUBCOM('ADD','DSNREXX','DSNREXX')

/*******************************************************************/
/*  Set up SQL statements to insert command output messages     */
/*  into a temporary table.                                     */
/*******************************************************************/
SQLSTMT='INSERT INTO SYSTMP.SYSPRINT(SEQNO,TEXT) VALUES(?,?)'
ADDRESS DSNREXX "EXECSQL DECLARE C1 CURSOR FOR S1"
IF SQLCODE = 0 THEN CALL SQLCA
ADDRESS DSNREXX "EXECSQL PREPARE S1 FROM :SQLSTMT"
IF SQLCODE = 0 THEN CALL SQLCA

/*******************************************************************/
/*  Extract messages from the return area and insert them into */
/*  the temporary table.                                        */
/*******************************************************************/
SEQNO = 0
OFFSET = 4+1
DO WHILE ( OFFSET < TOTLEN )
LEN = C2D(SUBSTR(RTRNAREA,OFFSET,2))
SEQNO = SEQNO + 1
TEXT = SUBSTR(RTRNAREA,OFFSET+4,LEN-4-1)
ADDRESS DSNREXX "EXECSQL EXECUTE S1 USING :SEQNO,:TEXT"
IF SQLCODE = 0 THEN CALL SQLCA
OFFSET = OFFSET + LEN
END

/*******************************************************************/
/*  Set up a cursor for a result set that contains the command  */
/*  output messages from the temporary table.                    */
/*******************************************************************/
SQLSTMT='SELECT SEQNO,TEXT FROM SYSTMP.SYSPRINT ORDER BY SEQNO'
ADDRESS DSNREXX "EXECSQL DECLARE C2 CURSOR FOR S2"
ADDRESS DSNREXX "EXECSQL PREPARE S2 FROM :SQLSTMT"
ADDRESS DSNREXX "EXECSQL OPEN C2"
S_RC = RXSUBCOM('DELETE','DSNREXX','DSNREXX') /* REMOVE CMD ENV */
EXIT SUBSTR(RTRNAREA,1,TOTLEN+4)

 /*******************************************************************/
 /*  Routine to display the SQLCA                                */
 /*******************************************************************/
SQLCA:
SAY 'SQLCODE ='SQLCODE
SAY 'SQLERRMC ='SQLERRMC
SAY 'SQLERRP ='SQLERRP
SAY 'SQLERRD  =',SQLERRD.1',','
|  SQLERRD.2'
|  SQLERRD.3'
|  SQLERRD.4'
|  SQLERRD.5'
|  SQLERRD.6
SAY 'SQLWARN  =',SQLWARN.0',','
|  SQLWARN.1'
|  SQLWARN.2'
|  SQLWARN.3'
|  SQLWARN.4'
|  SQLWARN.5'
|  SQLWARN.6'
|  SQLWARN.7'
|  SQLWARN.8'
|  SQLWARN.9'
|  SQLWARN.10
SAY 'SQLSTATE='SQLSTATE
SAY 'SQLCODE =',SQLCODE
EXIT 'SQLERRMC =',SQLERRMC'
|| 'SQLERRP =',SQLERRP'
|| 'SQLERRD  =',SQLERRD.1'
|  SQLERRD.2'
|  SQLERRD.3'
|  SQLERRD.4'
|  SQLERRD.5'
|  SQLERRD.6'
|| 'SQLWARN  =',SQLWARN.0'
|  SQLWARN.1'

Chapter 3. Db2 SQL programming 289
Related reference
Calling a stored procedure from a REXX procedure
The format of the parameters that you pass in the CALL statement in a REXX procedure must be compatible with the data types of the parameters in the CREATE PROCEDURE statement.

TSO/E services available under IKJTSOEV (TSO/E Programming Services)

Modifying an external stored procedure definition
You can modify the definition of an external stored procedure or the stored procedure source code. In either case, you need to prepare the stored procedure again.

Procedure
To modify an external stored procedure definition:
1. Issue one of the following:
   • FL 507 The CREATE PROCEDURE statement with the OR REPLACE clause and the SPECIFIC clause in the following cases:
     – When the parameter list of the existing procedure includes a table parameter.
     – When the CREATE statement specifies changes to the parameter list other than parameter names.
   • The ALTER PROCEDURE statement with the appropriate options.

   This new definition replaces the existing definition.
2. Prepare the external stored procedure again, as you did when you originally created the external stored procedure.

Example
Suppose that an existing C stored procedure was defined with the following statement:

```
CREATE PROCEDURE B(IN V1 INTEGER, OUT V2 CHAR(9))
LANGUAGE C
DETERMINISTIC
NO SQL
EXTERNAL NAME SUMMOD
COLLID SUMCOLL
ASUTIME LIMIT 900
PARAMETER STYLE GENERAL WITH NULLS
STAY RESIDENT NO
RUN OPTIONS 'MSGFILE(OUTFILE),RPTSTG(ON),RPTOPTS(ON)' WLM ENVIRONMENT PAYROLL
PROGRAM TYPE MAIN
SECURITY DB2
DYNAMIC RESULT SETS 10
COMMIT ON RETURN NO;
```

Assume that you need to make the following changes to the stored procedure definition:

• The stored procedure selects data from Db2 tables but does not modify Db2 data.
• The parameters can have null values, and the stored procedure can return a diagnostic string.
• The length of time that the stored procedure runs is unlimited.
• If the stored procedure is called by another stored procedure or a user-defined function, the stored procedure uses the WLM environment of the caller.

Either of the following statements can make these changes:

```
CREATE OR REPLACE PROCEDURE B(IN V1 INTEGER, OUT V2 CHAR(9))
  LANGUAGE C
  DETERMINISTIC
  READS SQL DATA
  EXTERNAL NAME SUMMOD
  COLLID SUMCOLL
  ASUTIME NO LIMIT
  PARAMETER STYLE SQL
  STAY RESIDENT NO
  RUN OPTIONS 'MSGFILE(OUTFILE),RPTSTG(ON),RPTOPTS(ON)'
  WLM ENVIRONMENT (PAYROLL,*)
  PROGRAM TYPE MAIN
  SECURITY DB2
  DYNAMIC RESULT SETS 10
  COMMIT ON RETURN NO;
```

```
ALTER PROCEDURE B
  READS SQL DATA
  ASUTIME NO LIMIT
  PARAMETER STYLE SQL
  WLM ENVIRONMENT (PAYROLL,*)
;```

**Related tasks**

Creating external stored procedures

An external stored procedure is a procedure that is written in a host language and can contain SQL statements. The source code for external procedures is separate from the definition.

**Related reference**

ALTER PROCEDURE (external) (Db2 SQL)

Creating external SQL procedures (deprecated)

An external SQL procedure is a procedure whose body is written entirely in SQL. The body is written in the SQL procedural language (SQL PL). However, an external SQL procedure is created, implemented, and executed like other external stored procedures. All SQL procedures that were created prior to DB2 9 are external SQL procedures.

**Before you begin**

Deprecated function: External SQL procedures are deprecated and not as fully supported as native SQL procedures. For best results, create native SQL procedures instead. For more information, see “Creating native SQL procedures” on page 231 and “Migrating an external SQL procedure to a native SQL procedure” on page 292.

Before you create an external SQL procedure, Configure Db2 for running stored procedures and user-defined functions during installation or Configure Db2 for running stored procedures and user-defined functions during migration.

**Procedure**

To create an external SQL procedure:

1. Use one of the following methods to create the external SQL procedure:
   - IBM Data Studio. See Developing database routines (IBM Data Studio, IBM Optim Database Administrator, IBM infoSphere Data Architect, IBM Optim Development Studio).
   - JCL
   - The Db2 for z/OS SQL procedure processor (DSNTIPSMP)

   The preceding methods that you use to create an external SQL procedure perform the following actions:
• Convert the external SQL procedure source statements into a C language program by using the Db2 precompiler.
• Create an executable load module and a Db2 package from the C language program.
• Define the external SQL procedure to Db2 by issuing a CREATE PROCEDURE statement either statically or dynamically.

**Restriction:** If you plan to use the Db2 stored procedure debugger or the Unified Debugger, do not use JCL. Use either IBM Data Studio or DSNTPSMP.

If you plan to use IBM Data Studio or DSNTPSMP, you must set up support for external SQL procedures.

2. Authorize the appropriate users to use the stored procedure by issuing the GRANT EXECUTE statement.

**Example**

For examples of how to prepare and run external SQL procedures, see “Sample programs to help you prepare and run external SQL procedures” on page 307.

**Related concepts**

SQL procedures
An SQL procedure is a stored procedure that contains only SQL statements.

**Related tasks**

Implementing Db2 stored procedures (Db2 Administration Guide)

**Related reference**

CREATE PROCEDURE (SQL - external) (deprecated) (Db2 SQL)
GRANT (function or procedure privileges) (Db2 SQL)

**Migrating an external SQL procedure to a native SQL procedure**

You can migrate an existing external SQL procedure, which is deprecated, to a native SQL procedure by dropping the existing procedure and creating it again as a native SQL procedure. Native SQL procedures are more fully supported, easier to maintain, and typically perform better than external SQL procedures, which are deprecated.

**Before you begin**

If you created the external SQL procedure in a previous release of Db2, consider the release incompatibilities for applications that use stored procedures, examine your external SQL procedure source code, and make any necessary adjustments. See Application and SQL release incompatibilities (Db2 for z/OS What's New?).

**About this task**

A *native SQL procedure* is a procedure whose body is written entirely in SQL. The body is written in the SQL procedural language (SQL PL). A native SQL procedure is created by issuing a single SQL statement, CREATE PROCEDURE. Native SQL procedures do not require any other program preparation, such as precompiling, compiling, or link-editing source code. Native SQL procedures are executed as SQL statements that are bound in a Db2 package. Native SQL procedures do not have an associated external application program. Native SQL procedures are more fully supported, easier to maintain, and typically perform better than external SQL procedures, which are deprecated.

An *external SQL procedure* is a procedure whose body is written entirely in SQL. The body is written in the SQL procedural language (SQL PL). However, an external SQL procedure is created, implemented, and executed like other external stored procedures.

**Procedure**

To migrate an external SQL procedure to a native SQL procedure, complete the following steps:
1. Find and save the existing CREATE PROCEDURE and GRANT EXECUTE statements for the existing external SQL procedure.

2. Drop the existing external SQL procedure by using the DROP PROCEDURE statement.

3. Re-create the procedure as a native SQL procedure by using the same CREATE PROCEDURE statement that you used to originally create the procedure, with both of the following changes:
   - If the procedure was defined with the options FENCED or EXTERNAL, remove these keywords.
   - Either remove the WLM ENVIRONMENT keyword, or add the FOR DEBUG MODE clause.
   - If the procedure body contains statements with unqualified names that could refer to either a column or an SQL variable or parameter, qualify these names. Otherwise, you might need to change the statement.

   Db2 resolves these names differently depending on whether the procedure is an external SQL procedure or a native SQL procedure. For external SQL procedures, Db2 first treats the name as a variable or parameter if one exists with that name. For native SQL procedures, Db2 first treats the name as a column if a column exists with that name. For example, consider the following statement:

   ```sql
   CREATE PROCEDURE P1 (INOUT C1 INT) ... SELECT C1 INTO xx FROM T1
   ```

   In the preceding example, if P1 is an external SQL procedure, C1 is a parameter. For native SQL procedures, C1 is a column in table T1. If such a column does not exist, C1 is a parameter.

4. Issue the same GRANT EXECUTE statements that you used to originally grant privileges for this stored procedure.

5. Increase the value of the TIME parameter on the job statement for applications that call stored procedures.

   **Important:** This change is necessary because time for SQL external stored procedures is charged to the WLM address space, while time for native SQL stored procedures is charged to the address space of the task.

6. Test your new native SQL procedure.

**Related tasks**

Creating native SQL procedures

A **native SQL procedure** is a procedure whose body is written entirely in SQL and is created by issuing a single SQL statement, CREATE PROCEDURE.

**Related reference**

- CREATE PROCEDURE (SQL - native) (Db2 SQL)
- GRANT (function or procedure privileges) (Db2 SQL)
- DROP (Db2 SQL)

*Using the Db2 precompiler to assist you in converting an external SQL procedure to a native SQL procedure*

The Db2 precompiler can be useful when considering any conversion of an external SQL procedure to a native SQL procedure.

**About this task**

Use the Db2 precompiler to inspect the SQL procedure source from a native SQL PL perspective. A listing is produced that helps to isolate problems and incompatibilities between external and native SQL procedure coding. Source changes can then be made before making any changes in Db2.

**Procedure**

To inspect the quality of native SQL PL source coding using the Db2 precompiler:

1. Copy the original SQL PL source code to a FB80 data set. Reformat the source as needed to fit within the precompiler margins.

2. Precompile the SQL PL source by executing program DSNHPSM with the HOST(SQLPL) option.

3. Inspect the produced listing (SYSPRINT). Pay attention to error and warning messages.
4. Modify the SQL PL source to address coding problems that are identified by messages in the listing.
5. Repeat steps “1” on page 293 through “4” on page 294 until all error and warning messages are resolved. Address informational messages as needed.
6. Copy the modified SQL PL source file back to its original source format, reformatting as needed.

Results
Sample JCL DSNTEJ67 demonstrates this process for an external SQL procedure that was produced using the Db2 SQL procedure processor DSNTPSMP.

Related reference
Sample programs to help you prepare and run external SQL procedures
Db2 provides sample jobs to help you prepare and run external SQL procedures. All samples are in data set DSN1210.SDSNSAMP. Before you can run the samples, you must customize them for your installation.

Creating an external SQL procedure by using DSNTPSMP
The SQL procedure processor, DSNTPSMP, is one of several methods that you can use to create and prepare an external SQL procedure. DSNTPSMP is a REXX stored procedure that you can invoke from your application program.

Before you begin
Deprecated function: External SQL procedures are deprecated and not as fully supported as native SQL procedures. For best results, create native SQL procedures instead. For more information, see “Creating native SQL procedures” on page 231 and “Migrating an external SQL procedure to a native SQL procedure” on page 292.
Set up support for external SQL procedures.
Also ensure that you have the required authorizations, as indicated in the following table, for invoking DSNTPSMP.

<table>
<thead>
<tr>
<th>Required authorization</th>
<th>Associated syntax for the authorization</th>
</tr>
</thead>
<tbody>
<tr>
<td>Procedure privilege to run application programs that invoke the stored procedure.</td>
<td>EXECUTE ON PROCEDURE SYSPROC.DSNTPSMP</td>
</tr>
<tr>
<td>Collection privilege to use BIND to create packages in the specified collection. You can use an asterisk (*) as the identifier for a collection.</td>
<td>CREATE ON COLLECTION collection-id</td>
</tr>
<tr>
<td>Package privilege to use BIND or REBIND to bind packages in the specified collection.</td>
<td>BIND ON PACKAGE collection-id.*</td>
</tr>
<tr>
<td>System privilege to use BIND with the ADD option to create packages and plans.</td>
<td>BINDADD</td>
</tr>
<tr>
<td>Schema privilege to create, alter, or drop stored procedures in the specified schema. The BUILDOWNER authorization ID must have the CREATEIN privilege on the schema. You can use an asterisk (*) as the identifier for a schema.</td>
<td>CREATEIN, ALTERIN, DROPIN ON SCHEMA schema-name</td>
</tr>
</tbody>
</table>
### Table 56. Required authorizations for invoking DSNTPSMP (continued)

<table>
<thead>
<tr>
<th>Required authorization</th>
<th>Associated syntax for the authorization</th>
</tr>
</thead>
<tbody>
<tr>
<td>Table privileges to select or delete from, insert into, or update the specified catalog tables.</td>
<td>SELECT ON TABLE SYSIBM.SYSROUTINES SELECT ON TABLE SYSIBM.SYSPARMS SELECT, INSERT, UPDATE, DELETE ON TABLE SYSIBM.SYSROUTINES_SRC SELECT, INSERT, UPDATE, DELETE ON TABLE SYSIBM.SYSROUTINES_OPTS ALL ON TABLE SYSIBM.SYSPSMOUT</td>
</tr>
<tr>
<td>Any privileges that are required for the SQL statements and that are contained within the SQL procedure body. These privileges must be associated with the OWNER authorization-id that is specified in your bind options. The default owner is the user that is invoking DSNTPSMP.</td>
<td>Syntax varies depending on the SQL procedure body</td>
</tr>
</tbody>
</table>

### Procedure

To create an external SQL procedure by using DSNTPSMP:

1. Write an application program that calls DSNTPSMP. Include the following items in your program:
   - A CLOB host variable that contains a CREATE PROCEDURE statement for the external SQL procedure. That statement should include the FENCED keyword or the EXTERNAL keyword, and the procedure body, which is written in SQL.
   - Alternatively, instead of defining a host variable for the CREATE PROCEDURE statement, you can store the statement in a data set member.
   - An SQL CALL statement with the BUILD function. The CALL statement should use the proper syntax for invoking DSNTPSMP.

   Pass the SQL procedure source to DSNTPSMP as one of the following input parameters:
   - **SQL-procedure-source**
     - Use this parameter if you defined a host variable in your application to contain the CREATE PROCEDURE statement.
   - **source-data-set-name**
     - Use this parameter if you stored the CREATE PROCEDURE statement in a data set.

   Based on the return value from the CALL statement, issue either an SQL COMMIT or a ROLLBACK statement. If the return value is 0 or 4, issue a COMMIT statement. Otherwise, issue a ROLLBACK statement.

   You must process the result set before issuing the COMMIT or ROLLBACK statement.

   A QUERYLEVEL request must be followed by the COMMIT statement.

2. Precompile, compile, and link-edit the application program.
3. Bind a package for the application program.
4. Run the application program.

### Related concepts

**SQL procedure body**
The body of an SQL procedure contains one or more SQL statements. In the SQL procedure body, you can also declare and use variables, conditions, return codes, statements, cursors, and handlers.

### Related reference

**CREATE PROCEDURE (SQL - external) (deprecated) (Db2 SQL)**
Db2 for z/OS SQL procedure processor (DSNTPSMP)
The SQL procedure processor, DSNTPSMP, is a REXX stored procedure that you can use to prepare an external SQL procedure for execution.

You can also use DSNTPSMP to perform selected steps in the preparation process or delete an existing external SQL procedure. DSNTPSMP is the only preparation method for enabling external SQL procedures to be debugged with either the SQL Debugger or the Unified Debugger.

DSNTPSMP requires that your system EBCDIC CCSID also be compatible with the C compiler. Using an incompatible CCSID results in compile-time errors. Examples of incompatible CCSIDs include 290, 930, 1026, and 1155. If your system EBCDIC CCSID is not compatible, do not just change it. Contact IBM Support for help.

Sample startup procedure for a WLM address space for DSNTPSMP
You must run DSNTPSMP in a WLM-established stored procedures address space. You should run only DSNTPSMP in that address space, and you must limit the address space to run only one task concurrently.

This example shows how to set up a WLM address space for DSNTPSMP.

Recommendation: Use the core WLM environment DSNWLM_REXX. Job DSNTIJMV creates an address space procedure called DSNWLMR for this environment.

The following example shows sample JCL for a startup procedure for the address space in which DSNTPSMP runs.

```
//DSNWLMR  PROC DB2SSN=DSN,NUMTCB=1,APPLENV=DSNWLM_REXX
//WLMTPSMP EXEC PGM=DSNX9WLM,TIME=1440,
//      PARM='&DB2SSN,&NUMTCB,&APPLENV',
//      REGION=8M,DYNALEN=10
//STEPLIB DD DISP=SHR,DSN=DSN1010.DSNSEXIT
//      DD DISP=SHR,DSN=DSN1010.DSNSLOAD
//      DD DISP=SHR,DSN=CBC.SCCNCMP
//      DD DISP=SHR,DSN=CEE.SCEERUN
//      DD DISP=SHR,DSN=DSN1010.DBRMLIB_DATA
//SYSEXEC DD DISP=SHR,DSN=DSN1010.DSNCLST
//SYSTSPRT DD SYSTSPRT=A
//CEEDUMP DD CEEDUMP=A
//SYSMSGS DD SYSMSGS=G

//SQLDBRM DD DISP=SHR,DSN=DSN1010.DBRMLIB.DATA
//SQLCSRC DD DISP=SHR,DSN=DSN1010.SRCLIB.DATA
//SQLMOD DD DISP=SHR,DSN=DSN1010.RUNLIB.LOAD
//SQLLIBC DD DISP=SHR,DSN=CEE.SCEEH.H
//      DD DISP=SHR,DSN=DSN1010.SDSNLOAD
//      DD DISP=SHR,DSN=CEE.SCEELKD
//SYSMSGS DD DISP=SHR,DSN=CEE.SCEEMSGP(EDCPMSGE)

//DSNTPSMP Configuration File - CFGTPSMP (optional)
//      A site-provided sequential data set or member, used to
//      define customized operation of DSNTPSMP in this APPLENV

//CFGTPSMP DD DISP=SHR,DSN=
//SQLSRC DD UNIT=SYSALLDA,SPACE=(23440,(20,20)),
//      DCB=(RECFM=FB,LRECL=80,BLKSIZE=23440)
//SQLPRINT DD UNIT=SYSALLDA,SPACE=(23476,(20,20)),
//      DCB=(RECFM=VB,LRECL=137,BLKSIZE=23476)
//SQLTERM DD UNIT=SYSALLDA,SPACE=(23476,(20,20)),
//      DCB=(RECFM=VB,LRECL=137,BLKSIZE=23476)
//SOLUTF DD UNIT=SYSALLDA,SPACE=(23476,(20,20)),
//      DCB=(RECFM=FB,LRECL=137,BLKSIZE=23476)
//SOLCPRT DD UNIT=SYSALLDA,SPACE=(23476,(20,20)),
//      DCB=(RECFM=FB,LRECL=137,BLKSIZE=23476)
//SOLUT1 DD UNIT=SYSALLDA,SPACE=(23440,(20,20)),
//      DCB=(RECFM=FB,LRECL=80,BLKSIZE=23440)
//SOLUT2 DD UNIT=SYSALLDA,SPACE=(23440,(20,20)),
//      DCB=(RECFM=FB,LRECL=80,BLKSIZE=23440)
//SOLCIN DD UNIT=SYSALLDA,SPACE=(32000,(20,20))
//      DCB=(RECFM=FB,LRECL=80,BLKSIZE=32000)
//SOLLIN DD UNIT=SYSALLDA,SPACE=(32000,(30,30)),
//      DCB=(RECFM=FB,LRECL=80,BLKSIZE=32000)
//SYSMOD DD UNIT=SYSALLDA,SPACE=(23440,(20,20))
```
Notes:

1. **APPLENV** specifies the application environment in which DSNTPSMP runs. To ensure that DSNTPSMP always uses the correct data sets and parameters for preparing each external SQL procedure, you can set up different application environments for preparing stored procedures with different program preparation requirements. For example, if all payroll applications use the same set of data sets during program preparation, you could set up an application environment called PAYROLL for preparing only payroll applications. The startup procedure for PAYROLL would point to the data sets that are used for payroll applications.

   DB2SSN specifies the Db2 subsystem name.

   NUMTCB specifies the number of programs that can run concurrently in the address space. You should always set NUMTCB to 1 to ensure that executions of DSNTPSMP occur serially.

2. **WLMTPSMP** specifies the address space in which DSNTPSMP runs.

   DYNAMNBR reserves space for dynamic allocation of data sets during the SQL procedure preparation process.

3. **STEPLIB** specifies the Db2 load libraries, the z/OS C/C++ compiler library, and the Language Environment run time library that DSNTPSMP uses when it runs. At least one library must not be APF authorized.

4. **SYSEXEC** specifies the library that contains the REXX exec DSNTPSMP.

5. **SQLDBRM** is an output data set that specifies the library into which DSNTPSMP puts the DBRM that it generates when it precompiles your external SQL procedure.

6. **SQLCSRC** is an output data set that specifies the library into which DSNTPSMP puts the C source code that it generates from the external SQL procedure source code. This data set should have a logical record length of 80.

7. **SQLMOD** is an output data set that specifies the library into which DSNTPSMP puts the load module that it generates when it compiles and link-edits your external SQL procedure.

8. **SQLLIBC** specifies the library that contains standard C header files. This library is used during compilation of the generated C program.

9. **SQLLIBL** specifies the following libraries, which DSNTPSMP uses when it link-edits the external SQL procedure:
   - Language Environment link-edit library
   - Db2 load library

10. **SYMSGS** specifies the library that contains messages that are used by the C prelink-edit utility.

11. **CFGTPSMP** specifies an optional data set that you can use to customize DSNTPSMP, including specifying the compiler level. For details on all of the options that you can set in this file and how to set them, see the DSNTPSMP CLIST comments.

12. The DD statements that follow describe work file data sets that are used by DSNTPSMP.
Related tasks
Converting from the AMI-based MQ functions to the MQI-based MQ functions (Db2 Installation and Migration)

CALL statement syntax for invoking DSNTPSMP
You can invoke the SQL procedure processor, DSNTPSMP, from an application program by using an SQL CALL statement. DSNTPSMP prepares an external SQL procedure.

The following diagrams show the syntax of invoking DSNTPSMP through the SQL CALL statement:

\[
\text{CALL} \quad \text{SYSPROC.DSNTPSMP} \quad (\quad \text{function} \quad , \quad \text{SQL-procedure-name} \quad , \quad \text{SQL-procedure-source} \quad \text{empty-string} \quad , \quad \text{bind-options} \quad \text{empty-string} \quad , \quad \text{compiler-options} \quad \text{empty-string} \quad , \quad \text{precompiler-options} \quad \text{empty-string} \quad , \quad \text{prelink-options} \quad \text{empty-string} \quad , \quad \text{link-options} \quad \text{empty-string} \quad , \quad \text{alter-statement} \quad \text{empty-string} \quad , \quad \text{source-data-set-name} \quad \text{empty-string} \quad , \quad \text{build-owner} \quad \text{empty-string} \quad , \quad \text{build-utility} \quad \text{empty-string} \quad , \quad \text{return-code} \quad )
\]

Figure 16. DSNTPSMP syntax

\[
\text{CALL} \quad \text{DSNTPSMP} \quad (\quad \text{option} \quad )
\]

Figure 17. CALL DSNTPSMP bind-options, compiler-options, precompiler-options, prelink-options, link-options

Note: You must specify:
- The DSNTPSMP parameters in the order listed
- The empty string if an optional parameter is not required for the function
- The options in the order: bind, compiler, precompiler, prelink, and link

The DSNTPSMP parameters are:

**function**
A VARCHAR(20) input parameter that identifies the task that you want DSNTPSMP to perform. The tasks are:

**BUILD**
Creates the following objects for an external SQL procedure:
- A DBRM, in the data set that DD name SQLDBRM points to
- A load module, in the data set that DD name SQLLMOD points to
- The C language source code for the external SQL procedure, in the data set that DD name SQLCSRC points to
- The stored procedure package
• The stored procedure definition

The following input parameters are required for the BUILD function:

  SQL-procedure name
  SQL-procedure-source or source-data-set-name

If you choose the BUILD function, and an external SQL procedure with name SQL-procedure-name already exists, DSNTPSMP issues an error message and terminates.

**BUILD_DEBUG**

Creates the following objects for an external SQL procedure and includes the preparation necessary to debug the external SQL procedure with the SQL Debugger and the Unified Debugger:

• A DBRM, in the data set that DD name SQLDBRM points to
• A load module, in the data set that DD name SQLLMOD points to
• The C language source code for the external SQL procedure, in the data set that DD name SQLCSRC points to
• The stored procedure package
• The stored procedure definition

The following input parameters are required for the BUILD_DEBUG function:

  SQL-procedure name
  SQL-procedure-source or source-data-set-name

If you choose the BUILD_DEBUG function, and an external SQL procedure with name SQL-procedure-name already exists, DSNTPSMP issues an error message and terminates.

**REBUILD**

Replaces all objects that were created by the BUILD function for an external SQL procedure, if it exists, otherwise creates those objects.

The following input parameters are required for the REBUILD function:

  SQL-procedure name
  SQL-procedure-source or source-data-set-name

**REBUILD_DEBUG**

Replaces all objects that were created by the BUILD_DEBUG function for an external SQL procedure, if it exists, otherwise creates those objects, and includes the preparation necessary to debug the external SQL procedure with the SQL Debugger and the Unified Debugger.

The following input parameters are required for the REBUILD_DEBUG function:

  SQL-procedure name
  SQL-procedure-source or source-data-set-name

**REBIND**

Binds the external SQL procedure package for an existing external SQL procedure.

The following input parameter is required for the REBIND function:

  SQL-procedure name

**DESTROY**

Deletes the following objects for an existing external SQL procedure:

• The DBRM, from the data set that DD name SQLDBRM points to
• The load module, from the data set that DD name SQLLMOD points to
• The C language source code for the external SQL procedure, from the data set that DD name SQLCSRC points to
• The stored procedure package
• The stored procedure definition
The following input parameter is required for the DESTROY function:

```
SQL-procedure name
```

**ALTER**

Updates the registration for an existing external SQL procedure.

The following input parameters are required for the ALTER function:

```
SQL-procedure name
alter-statement
```

**ALTER_REBUILD**

Updates an existing external SQL procedure.

The following input parameters are required for the ALTER_REBUILD function:

```
SQL-procedure name
SQL-procedure-source or source-data-set-name
```

**ALTER_REBUILD_DEBUG**

Updates an existing external SQL procedure, and includes the preparation necessary to debug the external SQL procedure with the SQL Debugger and the Unified Debugger.

The following input parameters are required for the ALTER_REBUILD_DEBUG function:

```
SQL-procedure name
SQL-procedure-source or source-data-set-name
```

**ALTER_REBIND**

Updates the registration and binds the SQL package for an existing external SQL procedure.

The following input parameters are required for the ALTER_REBIND function:

```
SQL-procedure name
alter-statement
```

**QUERYLEVEL**

Obtains the interface level of the build utility invoked. No other input is required.

**SQL-procedure-name**

A VARCHAR(261) input parameter that specifies the external SQL procedure name.

The name can be qualified or unqualified. The name must match the procedure name that is specified within the CREATE PROCEDURE statement that is provided in `SQL-procedure-source` or that is obtained from `source-data-set-name`. In addition, the name must match the procedure name that is specified within the ALTER PROCEDURE statement that is provided in `alter-statement`. Do not mix qualified and unqualified references.

**SQL-procedure-source**

A CLOB(2M) input parameter that contains the CREATE PROCEDURE statement for the external SQL procedure. If you specify an empty string for this parameter, you need to specify the name `source-data-set-name` of a data set that contains the external SQL procedure source code.

**bind-options**

A VARCHAR(1024) input parameter that contains the options that you want to specify for binding the external SQL procedure package. Do not specify the MEMBER or LIBRARY option for the Db2 BIND PACKAGE command.

**compiler-options**

A VARCHAR(255) input parameter that contains the options that you want to specify for compiling the C language program that Db2 generates for the external SQL procedure.

**precompiler-options**

A VARCHAR(255) input parameter that contains the options that you want to specify for precompiling the C language program that Db2 generates for the external SQL procedure. Do not specify the HOST option.
prelink-options
A VARCHAR(255) input parameter that contains the options that you want to specify for prelinking the
C language program that Db2 generates for the external SQL procedure.

link-options
A VARCHAR(255) input parameter that contains the options that you want to specify for linking the C
language program that Db2 generates for the external SQL procedure.

alter-statement
A VARCHAR(32672) input parameter that contains the SQL ALTER PROCEDURE statement to process
with the ALTER or ALTER_REBIND function.

source-data-set-name
A VARCHAR(80) input parameter that contains the name of a z/OS sequential data set or partitioned
data set member that contains the source code for the external SQL procedure. If you specify an
empty string for this parameter, you need to provide the external SQL procedure source code in
SQL-procedure-source.

build-owner
A VARCHAR(130) input parameter that contains the SQL identifier to serve as the build owner for
newly created SQL stored procedures.

When this parameter is not specified, the value defaults to the value in the CURRENT SQLID special
register when the build utility is invoked.

build-utility
A VARCHAR(255) input parameter that contains the name of the build utility that is invoked. The
qualified form of the name is suggested, for example, SYSPROC.DSNTPSMP.

return-code
A VARCHAR(255) output parameter in which Db2 puts the return code from the DSNTPSMP
invocation. The values are:

0
Successful invocation. The calling application can optionally retrieve the result set and then issue
the required SQL COMMIT statement.

4
Successful invocation, but warnings occurred. The calling application should retrieve the warning
messages in the result set and then issue the required SQL COMMIT statement.

8
Failed invocation. The calling application should retrieve the error messages in the result set and
then issue the required SQL ROLLBACK statement.

99x
Where x is a digit between 0 and 9. Failed invocation with severe errors. The calling application
should retrieve the error messages in the result set and then issue the required SQL ROLLBACK
statement. To view error messages that are not in the result set, see the job log of the address
space for the DSNTPSMP execution.

999
Unknown severe internal error

998
APF environment setup error

997
DSNREXX setup error

996
Global temporary table setup error

995
Internal REXX programming error
1.2x
Where \( x \) is a digit between 0 and 9. Level of DSNTPSMP when request is QUERYLEVEL. The calling application can retrieve the result set for additional information about the release and service level and then issue the required SQL COMMIT statement.

**Related reference**

Descriptions of SQL processing options
You can specify any SQL processing options regardless of whether you use the Db2 precompiler or the Db2 coprocessor. However, the Db2 coprocessor might ignore certain options because host language compiler options exist that provide the same information.

BIND and REBIND options for packages, plans, and services (Db2 Commands)
Compiler Options (C/C++) (XL C/C++ User’s Guide)
Binder options reference (MVS Program Management: User’s Guide and Reference)

*Examples of invoking the SQL procedure processor (DSNTPSMP)*
You can invoke the BUILD, DESTROY, REBUILD, and REBIND functions of DSNTPSMP.

**DSNTPSMP BUILD function:** Call DSNTPSMP to build an external SQL procedure. The information that DSNTPSMP needs is listed in the following table:

<table>
<thead>
<tr>
<th>Function</th>
<th>BUILD</th>
</tr>
</thead>
<tbody>
<tr>
<td>External SQL procedure name</td>
<td>MYSHEMA.SQLPROC</td>
</tr>
<tr>
<td>Source location</td>
<td>String in CLOB host variable procsrc</td>
</tr>
<tr>
<td>Bind options</td>
<td>VALIDATE(BIND)</td>
</tr>
<tr>
<td>Compiler options</td>
<td>SOURCE, LIST, LONGNAME, RENT</td>
</tr>
<tr>
<td>Precompiler options</td>
<td>SOURCE, XREF, STDSQL(NO)</td>
</tr>
<tr>
<td>Prelink options</td>
<td>None specified</td>
</tr>
<tr>
<td>Link options</td>
<td>AMODE=31, RMODE=ANY, MAP, RENT</td>
</tr>
<tr>
<td>Build utility</td>
<td>SYSPROC.DSNTPSMP</td>
</tr>
<tr>
<td>Return value</td>
<td>String returned in varying-length host variable returnval</td>
</tr>
</tbody>
</table>

The CALL statement is:

```sql
EXEC SQL CALL SYSPROC.DSNTPSMP('BUILD','MYSHEMA.SQLPROC',:procsrc,
  'VALIDATE(BIND)',
  'SOURCE, LIST, LONGNAME, RENT',
  'SOURCE, XREF, STDSQL(NO)',
  'AMODE=31, RMODE=ANY, MAP, RENT',
  'SYSPROC.DSNTPSMP',
  :returnval);
```

**DSNTPSMP DESTROY function:** Call DSNTPSMP to delete an external SQL procedure definition and the associated load module. The information that DSNTPMSP needs is listed in the following table:

<table>
<thead>
<tr>
<th>Function</th>
<th>DESTROY</th>
</tr>
</thead>
<tbody>
<tr>
<td>External SQL procedure name</td>
<td>MYSHEMA.OLDPROC</td>
</tr>
<tr>
<td>Return value</td>
<td>String returned in varying-length host variable returnval</td>
</tr>
</tbody>
</table>
The CALL statement is:
```
EXEC SQL CALL SYSPROC.DSNTPSMP('DESTROY','MYSCHEMA.OLDPROC','
  ',
  ',',' ',' ',' ',' ',' ','
  :returnval);
```

**DSNTPSMP REBUILD function:** Call DSNTPSMP to re-create an existing external SQL procedure. The information that DSNTPMSP needs is listed in the following table:

### Table 59. The functions DSNTPSMP needs to REBUILD an SQL procedure

<table>
<thead>
<tr>
<th>Function</th>
<th>REBUILD</th>
</tr>
</thead>
<tbody>
<tr>
<td>External SQL procedure name</td>
<td>MYSCHEMA.SQLPROC</td>
</tr>
<tr>
<td>Bind options</td>
<td>VALIDATE(BIND)</td>
</tr>
<tr>
<td>Compiler options</td>
<td>SOURCE, LIST, LONGNAME, RENT</td>
</tr>
<tr>
<td>Precompiler options</td>
<td>SOURCE, XREF, STDSQL(NO)</td>
</tr>
<tr>
<td>Prelink options</td>
<td>None specified</td>
</tr>
<tr>
<td>Link options</td>
<td>AMODE=31, RMODE=ANY, MAP, RENT</td>
</tr>
<tr>
<td>Source data set name</td>
<td>Member PROCSRC of partitioned data set DSN1210.SDSNSAMP</td>
</tr>
<tr>
<td>Return value</td>
<td>String returned in varying-length host variable returnval</td>
</tr>
</tbody>
</table>

The CALL statement is:
```
EXEC SQL CALL SYSPROC.DSNTPSMP('REBUILD','MYSCHEMA.SQLPROC','
  'VALIDATE(BIND)',
  'SOURCE, LIST, LONGNAME, RENT',
  'SOURCE, XREF, STDSQL(NO)',
  'AMODE=31, RMODE=ANY, MAP, RENT',
  'DSN1210.SDSNSAMP(PROCSRC)',
  :returnval);
```

If you want to re-create an existing external SQL procedure for debugging with the SQL Debugger and the Unified Debugger, use the following CALL statement, which includes the REBUILD_DEBUG function:

```
EXEC SQL CALL SYSPROC.DSNTPSMP('REBUILD_DEBUG','MYSCHEMA.SQLPROC','
  'VALIDATE(BIND)',
  'SOURCE, LIST, LONGNAME, RENT',
  'SOURCE, XREF, STDSQL(NO)',
  'AMODE=31, RMODE=ANY, MAP, RENT',
  'DSN1210.SDSNSAMP(PROCSRC)',
  :returnval);
```

**DSNTPSMP REBIND function:** Call DSNTPSMP to rebind the package for an existing external SQL procedure. The information that DSNTPMSP needs is listed in the following table:

### Table 60. The functions DSNTPSMP needs to REBIND an SQL procedure

<table>
<thead>
<tr>
<th>Function</th>
<th>REBIND</th>
</tr>
</thead>
<tbody>
<tr>
<td>External SQL procedure name</td>
<td>MYSCHEMA.SQLPROC</td>
</tr>
<tr>
<td>Bind options</td>
<td>VALIDATE(RUN), ISOLATION(RR)</td>
</tr>
<tr>
<td>Return value</td>
<td>String returned in varying-length host variable returnval</td>
</tr>
</tbody>
</table>

The CALL statement is:
Result set that the SQL procedure processor (DSNTPSMP) returns

DSNTPSMP returns one result set that contains messages and listings. You can write your client program to retrieve information from this result set. Because DSNTPSMP is a stored procedure, use the same technique that you would use to write a program to receive result sets from any stored procedure.

Each row of the result set contains the following information:

**Processing step**
- The step in the DSNTPSMP function process to which the message applies.

**DD name**
- The DD statement that identifies the data set that contains the message.

**Sequence number**
- The sequence number of a line of message text within a message.

**Message**
- A line of message text.

Rows in the message result set are ordered by processing step, DD name, and sequence number.

For an example of how to process a result set from DSNTPSMP, see the Db2 sample program DSNTEJ65.

**Related concepts**
- Db2 for z/OS SQL procedure processor (DSNTPSMP)
- The SQL procedure processor, DSNTPSMP, is a REXX stored procedure that you can use to prepare an external SQL procedure for execution.
- Job DSNTEJ65 (Db2 Installation and Migration)

**Related tasks**
- Writing a program to receive the result sets from a stored procedure
  - You can write a program to receive results set from a stored procedure for either a fixed number of result sets, for which you know the contents, or a variable number of result sets, for which you do not know the contents.

Creating an external SQL procedure by using JCL

Using JCL is one of several ways that you can create and prepare an external SQL procedure.

**Before you begin**

**Deprecated function:** External SQL procedures are deprecated and not as fully supported as native SQL procedures. For best results, create native SQL procedures instead. For more information, see “Creating native SQL procedures” on page 231 and “Migrating an external SQL procedure to a native SQL procedure” on page 292.

**About this task**

**Restriction:** You cannot use JCL to prepare an external SQL procedure for debugging with the Db2 stored procedure debugger or the Unified Debugger. If you plan to use either of these debugging tools, use either DSNTPSMP or IBM Data Studio to create the external SQL procedure.

**Procedure**
- To create an external SQL procedure by using JCL, include the following job steps in your JCL job:
  1. Issue a CREATE PROCEDURE statement that includes either the FENCED keyword or the EXTERNAL keyword and the procedure body, which is written in SQL.
Alternatively, you can issue the CREATE PROCEDURE statement dynamically by using an application such as SPUFI, DSNTEP2, DSNTIAD, or the command line processor.

**Tip:** If the routine body of the CREATE PROCEDURE statement contains embedded semicolons, change the default SQL terminator character from a semicolon to some other special character, such as the percent sign (%).

This statement defines the stored procedure to Db2. Db2 stores the definition in the Db2 catalog.

2. Run program DSNHPC with the HOST(SQL) option.

This program converts the external SQL procedure source statements into a C language program. DSNHPC also writes a new CREATE PROCEDURE statement in the data set that is specified in the SYSUT1 DD statement.

3. Precompile, compile, and link-edit the generated C program by using one of the following techniques:
   - The Db2 precompiler and JCL instructions to compile and link-edit the program
   - The SQL statement coprocessor

When you perform this step, specify the following settings:

   - Give the DBRM the same name as the name of the load module for the external SQL procedure.
   - Specify MARGINS(1,80) for the MARGINS SQL processing option.
   - Specify the NOSEQ compiler option.

This process produces an executable C language program.

4. Bind the resulting DBRM into a package.

**Example**

Suppose that you define an external SQL procedure by issuing the following CREATE PROCEDURE statement dynamically:

```sql
CREATE PROCEDURE DEVL7083.EMPDTLSS
  (IN  PEMPNO        CHAR(6)
   ,OUT PFIRSTNME     VARCHAR(12)
   ,OUT PMIDINIT      CHAR(1)
   ,OUT PLASTNAME     VARCHAR(15)
   ,OUT PWORKDEPT     CHAR(3)
   ,OUT PHIREDATE     DATE
   ,OUT PSALARY       DEC(9,2)
   ,OUT PSQLCODE      INTEGER
  )
RESULT SETS 0
MODIFIES SQL DATA
FENCED
NO DBINFO
WLM ENVIRONMENT DB2AWLMR
STAY RESIDENT NO
COLLID DEVL7083
PROGRAM TYPE MAIN
RUN OPTIONS 'TRAP(OFF),RPTOPTS(OFF)'
COMMIT ON RETURN NO
LANGUAGE SQL
BEGIN
DECLARE SQLCODE INTEGER;
DECLARE SQLSTATE CHAR(5);
DECLARE EXIT HANDLER FOR SQLEXCEPTION SET PSQLCODE = SQLCODE;
SELECT
  FIRSTNME
 , MIDINIT
 , LASTNAME
 , WORKDEPT
 , HIREDATE
 , SALARY
INTO   PFIRSTNME
 , PMIDINIT
 , PLASTNAME
 , PWORKDEPT
 , PHIREDATE
 , PSALARY
```
You can use JCL that is similar to the following JCL to prepare the procedure:

```
//ADMF001S JOB (999,POK),'SQL C/L/B/E',CLASS=A,MSGCLASS=T,
// NOTIFY=ADMF001,TIME=1440,REGION=0M
/*JOBPARM SYSAFF=5C63,L=9999
 // JCLLIB ORDER=(DB2AU.PROCLIB)
 /*
 /*JOBLIB DD DSN=DB2A.SDNSEXIT,DISP=SHR
 // DD DSN=DB2A.SDNSLOAD,DISP=SHR
 // DD DSN=CEE.SCEERUN,DISP=SHR
 /*--------------------- STEP 01: PRECOMP, COMP, LKED AN SQL PROCEDURE
 /*
 /*SQL01 EXEC DSNHSQL,MEM=EMPDTLSS,
 // PARM.PC='HOST(SQL),SOURCE,XREF,MAR(1,80),STDSQL(NO)',
 // PARM.PCC='HOST(C),SOURCE,XREF,MAR(1,80),STDSQL(NO),TWOPASS',
 // PARM.C='SOURCE LIST MAR(1,80) NOSEQ LO RENT',
 // PARM.LKED='AMODE=31,MMODE=ANY,MAP,RENT'
 //PC.SYSLIB DD DUMMY
 //PC.SYST2 DD DSN=&&SPDML,DISP=(,PASS), &lt;=MAKE IT PERMANENT, IF YOU
 // UNIT=SYSDA,SPACE=(TRK,1), WANT TO USE IT LATER
 // DCB=(RECFM=FB,LRECL=80),
 //PC.SYSCIN DD DISP=SHR,DSN=SG247083.PROD.DDL(&MEM.)
 //PCC.SYSIN DD DISP=SHR,DSN=SG247083.DEVL.DBRM(&MEM.)
 //LKED.SYSLMOD DD DISP=SHR,DSN=SG247083.DEVL.LOAD(&MEM.)
 //LKED.SYSIN DD * INCLUDE SYSLIB(DSNRL1) NAME EMPDTLSS(R)
 */
/*--------------------- STEP 02: BIND THE PROGRAM
 /*
 /*SQL02 EXEC PGM=IKJEFT01,DYNAMNBR=20,COND=(4,LT)
 //DBRMLIB DD DSN=SG247083.DEVL.DBRM,DISP=SHR
 //SYSTSPRT DD SYSOUT=* 
 //SYSPRINT DD SYSOUT=* 
 //SYSUDUMP DD SYSOUT=* 
 //SYSDUMP DD SYSOUT=* 
 //SYSDUMP DD SYSOUT=* 
 //REPORT DD SYSOUT=* 
 //SYSIN DD *
 //SYSIN DD * 
 //SYSTSM DD 
 /*DSN SYSTEM(DB2A)
 //BIND PACKAGE(DEVL7083) MEMBER(EMPDTLSS) VALIDATE(BIND) -
 //OWNER(DEVL7083)
 //END
 /*
```

Related concepts

SQL procedure body
The body of an SQL procedure contains one or more SQL statements. In the SQL procedure body, you can also declare and use variables, conditions, return codes, statements, cursors, and handlers.

The Db2 command line processor (Db2 Commands)

Related tasks

Changing SPUFI defaults
Before you execute SQL statements in SPUFI, you can change the default execution behavior, such as the SQL terminator and the isolation level.

Creating an external SQL procedure by using DSNTPSMP
The SQL procedure processor, DSNTPSMP, is one of several methods that you can use to create and prepare an external SQL procedure. DSNTPSMP is a REXX stored procedure that you can invoke from your application program.

Developing database routines (IBM Data Studio, IBM Optim Database Administrator, IBM infoSphere Data Architect, IBM Optim Development Studio)

Related reference

Descriptions of SQL processing options
You can specify any SQL processing options regardless of whether you use the Db2 precompiler or the Db2 coprocessor. However, the Db2 coprocessor might ignore certain options because host language compiler options exist that provide the same information.

**DSNTEP2 and DSNTEP4 sample programs**
You can use the DSNTEP2 or DSNTEP4 programs to execute SQL statements dynamically.

**DSNTIAD sample program**
You can use the DSNTIAD program to execute dynamic SQL statements other than SELECT statements.

**BIND PACKAGE (DSN) (Db2 Commands)**

**CREATE PROCEDURE (SQL - external) (deprecated) (Db2 SQL)**

---

**Sample programs to help you prepare and run external SQL procedures**

Db2 provides sample jobs to help you prepare and run external SQL procedures. All samples are in data set DSN1210.SDSNSAMP. Before you can run the samples, you must customize them for your installation.

**Deprecated function:** External SQL procedures are deprecated and not as fully supported as native SQL procedures. For best results, create native SQL procedures instead. For more information, see “Creating native SQL procedures” on page 231 and “Migrating an external SQL procedure to a native SQL procedure” on page 292.

See the prolog of each sample for specific instructions.

The following table lists the sample jobs that Db2 provides for external SQL procedures.

<table>
<thead>
<tr>
<th>Member that contains source code</th>
<th>Contents</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>DSNHSQL</td>
<td>JCL procedure</td>
<td>Precompiles, compiles, prelink-edits, and link-edits an external SQL procedure</td>
</tr>
<tr>
<td>DSNTEJ63</td>
<td>JCL job</td>
<td>Invokes JCL procedure DSNHSQL to prepare external SQL procedure DSN8ES1 for execution</td>
</tr>
<tr>
<td>DSN8ES1</td>
<td>External SQL procedure</td>
<td>A stored procedure that accepts a department number as input and returns a result set that contains salary information for each employee in that department</td>
</tr>
<tr>
<td>DSNTEJ64</td>
<td>JCL job</td>
<td>Prepares client program DSN8ED3 for execution</td>
</tr>
<tr>
<td>DSN8ED3</td>
<td>C program</td>
<td>Calls SQL procedure DSN8ES1</td>
</tr>
<tr>
<td>DSN8ES2</td>
<td>External SQL procedure</td>
<td>A stored procedure that accepts one input parameter and returns two output parameters. The input parameter specifies a bonus to be awarded to managers. The external SQL procedure updates the BONUS column of DSN1210.SDSNSAMP. If no SQL error occurs when the external SQL procedure runs, the first output parameter contains the total of all bonuses awarded to managers and the second output parameter contains a null value. If an SQL error occurs, the second output parameter contains an SQLCODE.</td>
</tr>
<tr>
<td>DSN8ED4</td>
<td>C program</td>
<td>Calls the SQL procedure processor, DSNTPSMP, to prepare DSN8ES2 for execution</td>
</tr>
<tr>
<td>DSN8WLMP</td>
<td>JCL procedure</td>
<td>A sample startup procedure for the WLM-established stored procedures address space in which DSNTPSMP runs</td>
</tr>
<tr>
<td>DSN8ED5</td>
<td>C program</td>
<td>Calls external SQL procedure DSN8ES2</td>
</tr>
</tbody>
</table>
### Table 61. External SQL procedure samples shipped with Db2 (continued)

<table>
<thead>
<tr>
<th>Member that contains source code</th>
<th>Contents</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>DSNTEJ65</td>
<td>JCL job</td>
<td>Prepares and executes programs DSN8ED4 and DSN8ED5. DSNTEJ65 uses DSNTGPSM, the SQL procedure processor, which requires that the default EBCDIC CCSID that is used by Db2 also be compatible with the C compiler. Do not run DSNTEJ65 if the default EBCDIC CCSID for Db2 is not compatible with the C compiler. Examples of incompatible CCSIDs include 290, 930, 1026, and 1155.</td>
</tr>
<tr>
<td>DSNTEJ67</td>
<td>JCL job</td>
<td>Prepares an existing external SQL procedure (sample DSN8.DSN8ES2) for conversion to a native SQL procedure. DSNTEJ67 obtains the source of external SQL procedure DSN8.DSN8ES2 from the catalog and formats it into a data set. DSNTEJ67 executes DSNHPSM with HOST(SQLPL), obtains a listing for the source, and replaces the offending procedure options in the source data set.</td>
</tr>
<tr>
<td>DSNTIJRT</td>
<td>JCL job</td>
<td>Prepares a Db2 for z/OS server for operation with the SQL Debugger and the Unified Debugger</td>
</tr>
</tbody>
</table>

### DSN8ED4

Demonstrates how to use an application program to call DSNTGPSM, the Db2 SQL Procedures Processor.

```c
/*************************************************************** 00010000
* Module name = DSN8ED4 (sample program)                      * 00020000
* DESCRIPTIVE NAME: Sample client for:                        * 00030000
*     DSNTGPSM (DB2 SQL Procedures Processor)                 * 00040000
* LICENSED MATERIALS - PROPERTY OF IBM                        * 00050000
* 5625-DB2                                                    * 00060000
* (C) COPYRIGHT 1982, 2003 IBM CORP. ALL RIGHTS RESERVED.     * 00070000
* STATUS = VERSION 8                                          * 00080000
* Function: Demonstrates how to use an application program to call DSNTGPSM, the DB2 SQL Procedures Processor. DSN8ED4 collects and passes user-provided SQL Procedure source code and prep options to DSNTGPSM, and outputs the report(s), if any, returned from DSNTGPSM by result set. * 00090000
* Notes:                                                     * 00100000
* Dependencies: Requires SYSPROC.DSNTGPSM                    * 00110000
* Restrictions:                                              * 00120000
* Module type: C program                                      * 00130000
* Processor: DB2 Precompiler                                  * 00140000
* IBM C/C++ for OS/390 V1R3 or higher                        * 00150000
* Module size: See linkedit output                            * 00160000
* Attributes: Reentrant and reusable                         * 00170000
* Entry point: DSN8ED4                                        * 00180000
* Purpose: See Function                                       * 00190000
* Linkage: Standard MVS program invocation, three parameters. * 00200000
* Parameters: DSN8ED4 uses the C "main" argument convention of argv (argument vector) and argc (argument count). * 00210000
* - ARGV[0]: (input) pointer to a char[9],                  * 00220000
*   null-terminated string having the name of this program (DSN8ED4) * 00230000
* - ARGV[1]: (input) pointer to a char[21],                 * 00240000
*   null-terminated string having the action                 * 00250000
***************************************************************/
```
that DSNTPSMP is to perform:

- **BUILD**: Prepare a new SQL Procedure
- **REBUILD**: Prepare an existing SQL
- **QUERYLEVEL**: Verify DSNTPSMP level
- **DESTROY**: Remove an SQL Procedure
- **REBIND**: Rebind the package of an existing SQL Procedure

- **ARGV[2]**: (input) pointer to a char[262], null-terminated string having the schema name of the SQL Procedure to be processed by DSNTPSMP (e.g. DSNB.DSNBES2)

- **ARGV[3]**: (input) pointer to a char[9], null-terminated string having the authorization id to be used for BUILDOWNER and for calling DSNTPSMP.

- **ARGV[4]**: (input) pointer to a char[17], null-terminated string having the name of the server where DSNTPSMP is to be run.

This is an optional parameter; the local server is used if no argument is provided.

- **Inputs**: DSNBED4 allocates these input DDs:
  - **PCOPTS**: Options for the DB2 precompiler
  - **COPTS**: Options for the C compiler
  - **PLKOPTS**: Options for the pre-link editor
  - **LKEDOPTS**: Options for the link editor
  - **BINDOPTS**: Options for the DB2 BIND
  - **SQLIN**: Source code for the SQL Procedure

- **Outputs**: DSNBED4 allocates these output DD

  - **REPORT01**: First report data set from DSNTPSMP
  - **REPORT02**: Second report data set from DSNTPSMP
  - **REPORT03**: Third report data set from DSNTPSMP

- **Normal Exit**: Return Code: 0000

  - Message: DSNTPSMP has completed with return code 0
  - Message: SQL changes have been committed

- **Normal with Warnings Exit**: Return Code: 0004

  - Message: DSNTPSMP has completed with return code 4
  - Message: SQL changes have been committed

- **Error Exit**: Return Code: 0012

  - Message: DSNTPSMP has completed with return code <n>

  - Message: The length of the argument specified for the <parameter-name> does not fall within the required bounds of <minimum-length> and <maximum-length>
  - Message: The argument <parameter-name> does not fall within the required bounds of <minimum-length> and <maximum-length>
  - Message: The argument specified for the action parameter is invalid
  - Message: Invalid sequence number <sequence-number> specified for REPORTn DD
  - Message: DSNBED4 was invoked with <parameter-count> parameters. At least 3 parameters are required
  - Message: Unable to open <DD-name>
  - Message: Unable to close <DD-name>
  - Message: <formatted SQL text from DSNTIAR>

- Message: SQL changes have been rolled back

- **External References:**

  - Routines/Services: DSNTIAR: DB2 msg text formatter
  - Data areas: None
  - Control blocks: None

- **Pseudocode:**

  - call getCallParms to receive and validate call parm arguments

  - case action
    - when **BUILD**, call getRebuildData
    - when **DESTROY**, call getDestroyData
    - when **REBUILD**, call getRebuildData
    - when **REBIND**, call getRebindData
    - when **QUERYLEVEL**, call getLevelData
    - otherwise call issueInvalidActionError

  - call connectToLocation

  - call **DSNTPSMP** to invoke the DB2 SQL Procedures Processor

  - call process**DSNTPSMP**PressultSet to write reports from DSNTPSMP

  - If no errors, call processSqlCommit to commit work
  - Else call processSqlRollback to undo work
*
End DSN8ED4
*
*
*
*
*
* Change activity =
*
*
PQ46962 03/28/2001 changed line feed character to hex 25
@01 *
*
PQ43444 04/12/2001 Disable LEOPTS DD (LE options are not
@02 *
*
processed by DSNTPSMP). Remission the
@02 *
*
leOptions hostvar as alterStmt.
@02 *
*
PQ56601 03/06/2002 Trim +/- continuation characters from
@03 *
*
BIND options to prevent BIND errors.
@03 *
*
These characters are often used to con- @03 *
*
tinue BIND statements being processed
@03 *
*
by the DB2 DSN command processor (which @03 *
*
uses TSO i/o services that recognize
@03 *
*
them as continuation characters) but
@03 *
*
they are not otherwise valid in DB2
@03 *
*
commands.
@03 *
*
PQ61782 07/16/2002 Distinguish between DSNTPSMP return code @04 *
*
and DSN8ED4 return code; Issue SQL COMMIT@04 *
*
when DSNTPSMP returns rc = 0 or rc = 4; @04 *
*
Otherwise issue SQL ROLLBACK
@04 *
*
D55199 12/08/2003 Adjust to use DSNTPSMP 1.2x interface
@05 *
*
D56462 02/12/2004 Allocate maximum of 6 output reports
@06 *
*********************************************************************/
/********************** C library definitions ***********************/
#include
<errno.h>
#include
<stdio.h>
#include
<stdlib.h>
#include
<string.h>
/**************************** Constants *****************************/
#define
NULLCHAR
'\0' /* Null character
*/
#define
RETNRM
0 /* Normal return code
@04*/
#define
RETWRN
4 /* Warning return code
*/
#define
RETERR
8 /* Error return code
*/
#define
RETSEV
12 /* Severe error return code
*/
#define
INTERFACE
"1.2" /* DSNTPSMP innterface level */
enum flag

{No, Yes}; /* Settings for flags

*/

/***************** Input: SQL Procedure Source Code *****************/
FILE
*sqlInFile;
/* Pointer to SQL source DD
*/
/*********** Output: DB2 SQL Procedures Processor Reports ***********/
FILE
*reportDD;
/* Pointer to curr report DD */
char
reportDDName[12];
/* For generated DD name
*/
unsigned short reportLRECL;
/* length req'd for output rec*/
/************************ Working variables *************************/
unsigned short resultSetReturned = 0;/* DSNTPSMP result set stat@04*/
long int
DSNTPSMP_rc
= -1; /* DSNTPSMP return code
@04*/
long int
rc
= 0; /* program return code
*/
char
levelquery
='N'; /* Is this a level check? @05*/
/******************** DB2 SQL Communication Area ********************/
EXEC SQL INCLUDE SQLCA;
/************************ DB2 Host Variables ************************/
EXEC SQL BEGIN DECLARE SECTION;
char

authID[9];

/* Authorization id-BUILDOWNER*/

char

locationName[17];

/* Server location name

char
action[21];
char
routineName[262];
SQL TYPE IS CLOB(2M) sqlSource;

*/

/* Command for PSM processor */
/* SQL Procedure schema.name */
/* SQL Procedure source
@05*/

char
char
char
char
char
char

precompOptions[256];/*
compileOptions[256];/*
prelinkOptions[256];/*
linkOptions[256];
/*
bindOptions[1025]; /*
alterStmt[32672];
/*

char

sqlSourceDsn[81];

precompiler options
compilation parameters
prelink options
link-edit options
DB2 bind options
ALTER PROC text

/* Source data set name

310 Db2 12 for z/OS: Application Programming and SQL Guide

*/
*/
*/
*/
*/
@02*/
*/

01240000
01250000
01260000
01270000
01280000
01290000
01300000
01310000
01320000
01330000
01340000
01350000
01360000
01370000
01380000
01390000
01400000
01410000
01420000
01430000
01440000
01450004
01460005
01470000
01480000
01490000
01500000
01510000
01520000
01530000
01540000
01550000
01560000
01570000
01580000
01590000
01600000
01610003
01620000
01630000
01640000
01650000
01660000
01670000
01680000
01690000
01700000
01710000
01720000
01730000
01740000
01750000
01760000
01770000
01780000
01790000
01800004
01810000
01820000
01830000
01840000
01850000
01860000
01870000
01880000
01890000
01900000
01910000
01920000
01930000
01940000
01950000
01960004
01970000
01980000
01990000
02000000
02010000
02020000
02030000
02040000
02050000


EXEC SQL END DECLARE SECTION;

/******* DB2 Result Set Locator Host Variables **************/
EXEC SQL BEGIN DECLARE SECTION;
char reportLine[256]; /* DSNTPSMP report line */
long int reportLineNumber; /* DSNTPSMP report line no. */
EXEC SQL END DECLARE SECTION;

Chapter 3. Db2 SQL programming
void issueSqlError  /* Handler for SQL error */ 02940000
( char *locMsg  /* - in: Call location */ 02950000
);

int main  /* DSN8ED4 driver */ 02990000
( int argc,  /* - Input argument count */ 03000000
  char **argv[]  /* - Input argument vector */ 03010000
)
{ 03020000
  /*******************************************************************************/ 03030000
  * Extract the following information from the call parms: 03040000
  * 1) DB2 location name where where SQL Procedure is to be built,* 03050000
  *    destroyed, rebuilt, rebound, etc.) 03060000
  * 2) DB2 SQL Procedure Processor action (Build,Destroy,...) 03070000
  * 3) Name of SQL Procedure to be built, destroyed, rebound, etc.* 03080000
  /*******************************************************************************/ 03090000
  getCallParms( argc,argv ); 03100000

  /*******************************************************************************/ 03110000
  * Collect DSNTPSMP parms appropriate for the user-passed action * 03120000
  /*******************************************************************************/ 03130000
  if( rc < RETSEV ) 03140000
  { 03150000
    if( memcmp( action,"BUILD",5 ) == 0 ) 03160000
    { 03170000
      getReBuildData(); 03180000
    } 03190000
    else if( memcmp( action,"DESTROY",7 ) == 0 ) 03200000
    { 03210000
      getDestroyData(); 03220000
    } 03230000
    else if( memcmp( action,"REBUILD",7 ) == 0 ) 03240000
    { 03250000
      getReBuildData(); 03260000
    } 03270000
    else if( memcmp( action,"REBIND",6 ) == 0 ) 03280000
    { 03290000
      getRebindData(); 03300000
    } 03310000
    else if( memcmp( action,"QUERYLEVEL",10 ) == 0 ) 03320000
    { 03330000
      getLevelData(); 03340000
      levelquery='Y'; 03350000
    } 03360000
    else 03370000
    { 03380000
      issueInvalidActionError( action ); 03390000
    } 03400000
  } 03410000

  /*******************************************************************************/ 03420000
  * Connect to location where the SQL Procedure is to be processed * 03430000
  /*******************************************************************************/ 03440000
  if( rc < RETSEV && strlen(locationName) > 0 ) 03450000
  connectToLocation(); 03460000

  /*******************************************************************************/ 03470000
  * Set current DB2 authorization id to use when calling DSNTPSMP * 03480000
  /*******************************************************************************/ 03490000
  if( rc < RETSEV ) 03500000
  setAuthID(); 03510000

  /*******************************************************************************/ 03520000
  * Call the PSM processor * 03530000
  /*******************************************************************************/ 03540000
  if( rc < RETSEV ) 03550000
  callDSNTPSMP(); 03560000

  /*******************************************************************************/ 03570000
  * Process the result set, if any, from DSNTPSMP * 03580000
  /*******************************************************************************/ 03590000
  if( resultSetReturned ) 03600000
  processDSNTPSMP resultSet(); 03610000

} 03620000

void issueInvalidParmLengthError  /* Handler for parm len error */ 02890000
( char *parmName,  /* - Input: identify of parm */ 02900000
  int minLength,  /* - in: min valid length */ 02910000
  int maxLength,  /* - in: max valid length */ 02920000
); 02930000

} 02800000
/* if( rc < RETSEV )                                         */
getOptions( bindOptions,1024,"BINDOPTS" );
/* if( rc < RETSEV )                                                  */
getOptions( linkOptions,255,"LKEDOPTS" );
/* if( rc < RETSEV )                                                  */
getOptions( prelinkOptions,255,"PLKDOPTS" );
/* if( rc < RETSEV )                                                  */
getOptions( compileOptions,255,"COPTS" );
/* if( rc < RETSEV )                                                  */
getOptions( precompOptions,255,"PCOPTS" );
*******************************************************************/
* Get program prep, bind, and runtime options                    *
/*******************************************************************/
{                                                                    }
*******************************************************************/
* perform a BUILD or REBUILD operation.                            *
/*******************************************************************/
void getReBuildData( void )           /* Get SQL Proc re/build data */
{                                                                    }
char *argv[]                      /* - Input argument vector    */
(int argc,                         /* - Input argument count     */
void getCallParms                     /* Process args to call parms */
{                                                                    }
/* Verifies that correct call parms have been passed in: *
- Three prep parameters (action, routine name, and authorization id)*
- require arguments
- The fourth parameter (location name) is optional *
*******************************************************************/
if( argc < 4 || argc > 5 )                                        
{ issueInvalidCallParmCountError( argc );        }
else if( strlen( argv[1] ) < 1 || strlen( argv[1] ) > 20 )          
{ issueInvalidParmLengthError("DSNTPSMP Action",1,20);       }
{ issueInvalidParmLengthError("SQL Procedure schema.name",1,261); } 
{ issueInvalidParmLengthError("Authorization ID",1,8);       }
else                                                                 
{ strcpy( action, argv[1] );                                
strcpy( routineName, argv[2] );                             
strcpy( authID, argv[3] );                                    }
if( argc > 4 )                                                
{ issueInvalidCallParmCountError( argc );        }
{ issueInvalidParmLengthError("Server Location Name",1,16);  }
else                                                                 
{ strcpy( locationName,argv[4] );                        
locationName[0] = NULLCHAR;                                   }
} /* end of getCallParms */                                          
return( rc );                                                      
*****************************************************************/
* Return highest completion code                                 *
*****************************************************************
/* getOptions( LeOptions,254,"LEOPTS" ); @02*/ 04520000
04530000
/* getOptions( LeOptions,254,"LEOPTS" ); */ 04540000
045450000
* Get the source for the SQL procedure to be prepared 
* 04550000
0455500000
if( rc < RETSEV ) 04560000
045650000
getSqlSource(); 04570000
045750000
getSqlSource(); 04580000
045850000
} /* end of getRebindData */ 04590000
045950000

void getDestroyData( void ) /* Get SQL Proc destroy data */ 04600000
0460500000
* Gets the name of the package to be freed by DSNTPSMP during a
* DESTROY operation. 04610000
0461500000
* Set program prep and runtime options to NULLCHAR 04620000
046250000
046300000
} /* end of getRebindData */ 04640000
046450000

void getRebindData( void ) /* Rebind an SQL Procedure */ 04650000
0465500000
* Gets the name of the package to be rebound by DSNTPSMP during a
* REBIND operation. 04660000
0466500000
* Set program prep and runtime options to NULLCHAR 04670000
046750000
046800000
} /* end of getRebindData */ 04690000
046950000

void getLevelData( void ) /* QueryLevel of DSNTPSMP */ 04700000
0470500000
* Prepare for a DSNTPSMP QUERYLEVEL operation. 04710000
0471500000
* Set program prep and runtime options to NULLCHAR 04720000
047250000
047300000
} /* end of getRebindData */ 04740000
047450000

/* Set program prep and runtime options to NULLCHAR */ 04750000
047550000
047600000
void getRebindData( void ) /* Get Rebind Data */ 04770000
0477500000
047800000
} /* end of getRebindData */ 04790000
047950000

/* Get the source for the SQL procedure to be prepared */ 04800000
0480500000
048100000
/* Set program prep and runtime options to NULLCHAR */ 04820000
048250000
048300000
/* Get name of package to free */ 04840000
0484500000
/* Get parameters to pass for rebind */ 04850000
0485500000
/* Get the source for the SQL procedure to be prepared */ 04860000
0486500000
/* Get the source for the SQL procedure to be prepared */ 04870000
0487500000
/* Get the source for the SQL procedure to be prepared */ 04880000
0488500000
/* Get the source for the SQL procedure to be prepared */ 04890000
0489500000
/* Get the source for the SQL procedure to be prepared */ 04900000
0490500000
/* Get the source for the SQL procedure to be prepared */ 04910000
0491500000
/* Get the source for the SQL procedure to be prepared */ 04920000
0492500000
/* Get the source for the SQL procedure to be prepared */ 04930000
0493500000
/* Get the source for the SQL procedure to be prepared */ 04940000
0494500000
/* Get the source for the SQL procedure to be prepared */ 04950000
0495500000
/* Get the source for the SQL procedure to be prepared */ 04960000
0496500000
/* Get the source for the SQL procedure to be prepared */ 04970000
0497500000
/* Get the source for the SQL procedure to be prepared */ 04980000
0498500000
/* Get the source for the SQL procedure to be prepared */ 04990000
0499500000
/* Get the source for the SQL procedure to be prepared */ 05000000
0500500000
/* Get the source for the SQL procedure to be prepared */ 05010000
0501500000
/* Get the source for the SQL procedure to be prepared */ 05020000
0502500000
/* Get the source for the SQL procedure to be prepared */ 05030000
0503500000
/* Get the source for the SQL procedure to be prepared */ 05040000
0504500000
/* Get the source for the SQL procedure to be prepared */ 05050000
0505500000
/* Get the source for the SQL procedure to be prepared */ 05060000
0506500000
/* Get the source for the SQL procedure to be prepared */ 05070000
0507500000
/* Get the source for the SQL procedure to be prepared */ 05080000
0508500000
/* Get the source for the SQL procedure to be prepared */ 05090000
0509500000
/* Get the source for the SQL procedure to be prepared */ 05100000
0510500000
/* Get the source for the SQL procedure to be prepared */ 05110000
0511500000
/* Get the source for the SQL procedure to be prepared */ 05120000
0512500000
/* Get the source for the SQL procedure to be prepared */ 05130000
0513500000
/* Get the source for the SQL procedure to be prepared */ 05140000
0514500000
/* Get the source for the SQL procedure to be prepared */ 05150000
0515500000
/* Get the source for the SQL procedure to be prepared */ 05160000
0516500000
/* Get the source for the SQL procedure to be prepared */ 05170000
0517500000
/* Get the source for the SQL procedure to be prepared */ 05180000
0518500000
/* Get the source for the SQL procedure to be prepared */ 05190000
0519500000
/* Get the source for the SQL procedure to be prepared */ 05200000
0520500000
/* Get the source for the SQL procedure to be prepared */ 05210000
0521500000
/* Get the source for the SQL procedure to be prepared */ 05220000
0522500000
/* Get the source for the SQL procedure to be prepared */ 05230000
0523500000
/* Get the source for the SQL procedure to be prepared */ 05240000
0524500000
/* Get the source for the SQL procedure to be prepared */ 05250000
0525500000
/* Get the source for the SQL procedure to be prepared */ 05260000
0526500000
/* Get the source for the SQL procedure to be prepared */ 05270000
0527500000
/* Get the source for the SQL procedure to be prepared */ 05280000
0528500000
/* Get the source for the SQL procedure to be prepared */ 05290000
0529500000
/* Get the source for the SQL procedure to be prepared */ 05300000
0530500000
/* Get the source for the SQL procedure to be prepared */ 05310000
0531500000
/* Get the source for the SQL procedure to be prepared */ 05320000
0532500000
/* Get the source for the SQL procedure to be prepared */ 05330000
0533500000

void getOptions                       /* Read processing options */
  ( char *options,                     /* -out: list of options read */
    int maxBytes,                      /* - in: max size of list */
    char *optionsDDname               /* - in: name of DD to read */
  )
  /* Read up to maxBytes bytes of data from optionsDDname into the */
  /* options buffer. */

FILE *optionsFile;      /* Ptr to specified options DD*/
char optionsDD[12];     /* DD handle                  */
char optionsRec[80];    /* Options file input record  */
short int recordLength  = 0; /* Length of record */
unsigned short moreRecords = Yes; /* EOF indicator */

sprintf( optionsDD,                                                
  "DD:%s\0",                                                
  optionsDDname );                                          

errno = 0;                        /* clear LE errno             */
optionsFile = fopen( optionsDD,                                    
  "rb,lrecl=80,type=record" );                   
if( optionsFile == NULL )                                          
  issueDataSetOpeningError( optionsDD,errno );                     
while( moreRecords == Yes  &&  rc < RETSEV )                       
  { recordLength                                                   
    = fread( optionsRec,         /* Read into options rec area */
      1,                 /* ..1 record                 */
      80,                /* ..of 80 bytes              */
      optionsFile );     /* ..from current options file*/
  
  if( ferror(optionsFile) )     /* Handle IO errors           */
    issueDataSetReadingError( optionsDD,errno );                
  else if( feof(optionsFile) )  /* Handle EOF                 */
    moreRecords = No;                                           
    /* Discard bytes 73-80 and                                */
  else                          /* strip off trailing blanks  */
    { strncat( options,optionsRec,72 );                          
      trimTrailingBlanks( options );                             
      /* Remove +/- continuation chars from BIND input     @03*/
      if( memcmp( optionsDDname,"BINDOPTS",8 ) == 0 )    /*@03*/
        stripContinuationCharacter( options );           /*@03*/  
      } /* Don't overfill return area */ 05820000
  if( rc < RETSEV  &&  strlen(options) > maxBytes )              
    issueInvalidParmLengthError( optionsDD,0,maxBytes );         
}

if( rc < RETSEV )                                                  
  if( fclose( optionsFile ) != 0 )                                 
    issueDataSetClosingError( optionsDD,errno );                   

void getSqlSource( void )             /* Read SQL Procedure Source */

char sourceRec[80];     /* Source file input record */
short int recordLength  = 0; /* Length of record */
unsigned short moreRecords = Yes; /* EOF indicator */

sqlInFile = fopen( "DD:SQLIN",                                    
  "rb,lrecl=80,type=record" );                    
if( sqlInFile == NULL )                                            
  issueDataSetOpeningError( "DD:SQLIN",errno );                   
while( moreRecords == Yes  &&  rc < RETSEV )                       
  { recordLength                                                   
    = fread( sourceRec,         /* Read into source rec area */
      1,                 /* ..1 record                 */
      80,                /* ..of 80 bytes              */
      sqlInFile );     /* ..from current source file*/
  
  if( ferror(sqlInFile) )     /* Handle IO errors           */
    issueDataSetReadingError( "DD:SQLIN",errno );                
  else if( feof(sqlInFile) )  /* Handle EOF                 */
    moreRecords = No;                                           
    /* Discard bytes 73-80 and                                */
  else                          /* strip off trailing blanks  */
    { strncat( options,optionsRec,72 );                          
      trimTrailingBlanks( options );                             
      /* Remove +/- continuation chars from BIND input     @03*/
      if( memcmp( optionsDDname,"BINDOPTS",8 ) == 0 )    /*@03*/
        stripContinuationCharacter( options );           /*@03*/
    } /* Don't overfill return area */ 05820000
  if( rc < RETSEV  &&  strlen(options) > maxBytes )              
    issueInvalidParmLengthError( optionsDD,0,maxBytes );         
}

if( rc < RETSEV )                                                  
  if( fclose( sqlInFile ) != 0 )                                  
    issueDataSetClosingError( "DD:SQLIN",errno );                   

Chapter 3. Db2 SQL programming
void connectToLocation( void )        /* Connect to DB2 location    */
{ EXEC SQL
  CONNECT TO :locationName;
EXEC SQL
if( SQLCODE != 0 )
{ issueSqlError( "Connect to location failed" );
}
} /* end of connectToLocation */

void setAuthID( void )                /* Set the current DB2 auth id*/
{ EXEC SQL
  SET CURRENT SQLID = :authID;
EXEC SQL
if( SQLCODE != 0 )
{ issueSqlError( "Set current SQLID failed" );
}
} /* end of setAuthID */

void callDSNTPSMP( void )             /* Run SQL Procedure Processor*/
{ listDSNTPSMPcallParms();
 EXEC SQL
 CALL SYSPROC.DSNTPSMP( :action,  
 :routineName,  
 :sqlSource,  
 :bindOptions,  
 :compileOptions,  
 :precompOptions,  
 :prelinkOptions,  
 :linkOptions,  
 :alterStmt, /*@02*/  
 :sqlSourceDsn, /*@05*/  
 :DSNTPSMP_pname, /*@05*/  
 :outputString );
EXEC SQL
if( SQLCODE != 0  &&  SQLCODE != 466 ) /*+@04*/
316 Db2 12 for z/OS: Application Programming and SQL Guide
void listDSNTPSMPcallParms( void ) /* List parms sent to DSNTPSMP */
{
    printf( "** DSNBED4 is now invoking the DB2 SQL Procedures " )
    printf( "Processor (SYSPROC.DSNTPSMP)\n" );
    printf( "\n" );
    printf( "\n" );
    printf( "Location name: %s\n", locationName );
    printf( "\n" );
    printf( "SQL Procedure name: %s\n", routineName );
    printf( "\n" );
    printf( "Action specified: %s\n", action );
    printf( "\n" );
    printf( "Link Editor Options:\n%*%s\n", 0, linkOptions );
    printf( "\n" );
    printf( "Prelink Editor Options:\n%*%s\n", 0, prelinkOptions );
    printf( "\n" );
    printf( "Compiler Options:\n%*%s\n", 0, compileOptions );
    printf( "\n" );
    printf( "DB2 Bind Options:\n%*%s\n", 0, bindOptions );
    printf( "\n" );
    if( strlen(alterStmt) > 0 )
    {
        printf( "** ALTER statement:\n%*%s\n", 0, alterStmt );
        printf( "\n" );
    }
    else /* SQLCODE == 0 */
    {
        issueSqlError( "Call to DSNTPSMP failed" );
    }
}

void processDSNTPSMPresultSet( void ) /* Handle DSNTPSMP result sets */
{
    /* Outputs data from the result set returned by DSNTPSMP */
    issueSqlError( "Call to DSNTPSMP failed" );
}

else if( levelquery != 'Y' )
{
    DSNTPSMP_rc = atoi( outputString );
    if( SQLCODE == 466 )
    {
        resultSetReturned = Yes;
    } else /* SQLCODE == 0 */
    {
        resultSetReturned = No;
    }
    if( memicmp( outputString, INTERFACE, 3 ) != 0 )
    {
        issueInvalidLevelError( outputString );
    } else /* SQLCODE == 0 */
    {
        resultSetReturned = Yes;
    }
    if( SQLCODE == 466 )
    {
        resultSetReturned = Yes;
    } else /* SQLCODE == 0 */
    {
        resultSetReturned = No;
    }
    if( levelquery == 'Y' )
    {
        DSNTPSMP_rc = 0;
    } else /* not applicable */
    {
        printf( "\n" );
    }
} /* end of listDSNTPSMPcallParms */
void associateResultSetLocator(void) /* Assoc DSNTPSMP RS locator */
{
  EXEC SQL
  ASSOCIATE
  LOCATORS( :DSNTPSMP_rs_loc1 )
  WITH PROCEDURE SYSPROC.DSNTPSMP;

  if( SQLCODE != 0 )
  { issueSqlError( "Associate locator call failed" );
  }
}

void allocateResultSetCursor( void ) /* Alloc DSNTPSMP RS cursor */
{
  EXEC SQL
  ALLOCATE DSNTPSMP_RS_CSR1
  CURSOR FOR RESULT SET :DSNTPSMP_rs_loc1;

  if( SQLCODE != 0 )
  { issueSqlError( "Allocate result set cursor "
      "call failed" );
  }
}

void writeDSNTPSMPReports( void ) /* Print DSNTPSMP report */
{
  short int reportNumber = 1;       /* Sequence number of report */
  char prevStepName[17];       /* Track step name changes */
  char prevFileName[9];        /* Track file name changes */
  short int recordLength = 0;       /* Length of record */

  /* Get the first entry in the result set */
  fetchFromResultSetCursor();

  /* Allocate an outout DD for the first report */
  openReportDataSet( reportNumber );

  /* Save step and file, to monitor for when they change */

  /* Outputs the reports returned in the result set from DSNTPSMP */
  */

  /* The result set returned by DSNTPSMP contains one or more reports. * 
   * With the result set, reports are distinguished from one another * 
   * by the STEP and FILE columns: * 
   * - STEP refers to the phase (e.g. precompile, compile, bind, etc.) * 
   * - FILE distinguishes reports that are generated by the same STEP. * 
   * Report line data are stored in the LINE column, and arranged ac- * 
   * cording to the sequence number in the SEQN column. * 
   * In summary, STEPs contain FILEs, FILEs contain LINEs, and LINEs * 
   * are ordered according to SEQN (sequence). * 
   */

  /* are ordered according to SEQN (sequence). */
  if( rc < RETSEV )
  { openReportDataSet( reportNumber );
    if( rc < RETSEV )
    { strcpy( prevStepName,stepName,17 );
      strcpy( prevFileName,fileName,9 );
    }
  }

  if( SQLCODE == 0 & & rc < RETSEV )
  { if( ( strcmp( prevStepName,stepName ) != 0 ) ||
      ( strcmp( prevFileName,fileName ) != 0 ) )
    & & reportNumber < 6 )
    return; /*@06*/
  }
}

/* Server message data set */
/* Associate the result set from DSNTPSMP with a result set locator */
*/

/* Get the first entry in the result set */
/* Allocate an outout DD for the first report */
*/

/* Outputs the reports returned in the result set from DSNTPSMP */
/* The result set returned by DSNTPSMP contains one or more reports. */
/* Within the result set, reports are distinguished from one another */
/* by the STEP and FILE columns: */
/* - STEP refers to the phase (e.g. precompile, compile, bind, etc.) */
/* - FILE distinguishes reports that are generated by the same STEP. */
/* Report line data are stored in the LINE column, and arranged ac- */
/* cording to the sequence number in the SEQN column. */
/* In summary, STEPs contain FILEs, FILEs contain LINEs, and LINEs */
/* are ordered according to SEQN (sequence). */
/* are ordered according to SEQN (sequence). */

/* Get the first entry in the result set */
/* Allocate an outout DD for the first report */
*/

/* Outputs the reports returned in the result set from DSNTPSMP */
/* The result set returned by DSNTPSMP contains one or more reports. */
/* Within the result set, reports are distinguished from one another */
/* by the STEP and FILE columns: */
/* - STEP refers to the phase (e.g. precompile, compile, bind, etc.) */
/* - FILE distinguishes reports that are generated by the same STEP. */
/* Report line data are stored in the LINE column, and arranged ac- */
/* cording to the sequence number in the SEQN column. */
/* In summary, STEPs contain FILEs, FILEs contain LINEs, and LINEs */
/* are ordered according to SEQN (sequence). */
/* are ordered according to SEQN (sequence). */
* If the step or file changes, allocate next report DD * 08620000
* up to and including report no. 6 08630000
********************************************************************** 08640000
{ closeReportDataSet(); 08650000
  if( rc < RETSEV ) 08660000
    openReportDataSet( ++reportNumber ); 08670000
  if( rc < RETSEV ) 08680000
    { strncpy( prevStepName,stepName,17 ); 08690000
      strncpy( prevFileName,fileName,9 ); 08700000
    } 08710000
} 08720000
/************************************************************* 08730000
* Write the current report line to the current report DD 08740000
*************************************************************/ 08750000
if( rc < RETSEV ) 08760000
  { recordLength 08770000
    = fwrite( reportLine, /* write from reportLine */ 08780000
              1, /* ...a record */ 08790000
              sizeof( reportLine ), 08800000
              reportDD ); /* ..into the report data set */ 08810000
  } 08820000
if( rc < RETSEV ) 08830000
  { fetchFromResultSetCursor(); 08840000
  } 08850000
} 08860000
} /* end of writeDSNTPSMPreports */ 08870000
08880000
void fetchFromResultSetCursor( void ) /* Read DSNTSPMP RS cursor */ 08890000
/************************************************************************** 08900000
* Reads the cursor for the DSNTPSMP result set * 08910000
**********************************************************************/ 08920000
{ memset( reportLine,' ',256 ); 08930000
  EXEC SQL 08940000
  FETCH DSNTPSMP_RS_CSR1 08950000
  INTO :stepName, 08960000
      :fileName, 08970000
      :reportLineNumber, 08980000
      :reportLine; 08990000
  if( SQLCODE != 0  &&  SQLCODE != 100  &&  rc < RETSEV ) 09000000
    { issueSqlError( "*** Fetch from " 09010000
      "result set cursor failed" ); 09020000
    } 09030000
} /* end of fetchFromResultSetCursor */ 09040000
09050000
09060000
void openReportDataSet( short int reportNumber ) /* Alloc DD for a report */ 09070000
( /* - in: Sequence number */ 09080000
) 09090000
/************************************************************************** 09100000
* Opens the DD REPORTnn, where "nn" is the report number passed in * 09110000
* and associates it with the file handler reportDD. * 09120000
**********************************************************************/ 09130000
{ char          reportDDdcb[36]; /* for generated DCB */ 09140000
  if( reportNumber < 1 || reportNumber > 99 ) 09150000
    issueInvalidDDnumError( reportNumber ); 09160000
  else 09170000
    { sprintf( reportDDName, /* Generate DD name REPORTnn */ 09180000
      "DD:REPORT%2.2i\0", 09190000
      reportNumber ); 09200000
      if( reportLine[0] == '1' ) 09210000
        { sprintf( reportDDdcb, /* Does this look like FBA? */ 09220000
          "wb.recfm=FBA," 09230000
          "lrecl=256" ); 09240000
        } 09250000
        else 09260000
          { sprintf( reportDDdcb, /* No: Specify */ 09270000
            "wb.recfm=FB," 09280000
            "lrecl=256" ); 09290000
          } 09300000
        errno = 0; /* clear LE errno */ 09310000
        reportDD = fopen( reportDDName,reportDDdcb ); 09320000
  } 09330000
}
if( reportDD == NULL ) /* If unable to open data set */
    issueDataSetOpeningError( reportDDName,errno );
} /* end of openReportDataSet */

void closeReportDataSet( void ) /* Dealloc DD for a report */
/*********************************************************************/
/* Closes the DD associated with the file handler reportDD. */
*********************************************************************/
{
    if( fclose(reportDD) != 0 )
        issueDataSetClosingError( reportDDName,errno );
} /* end of closeReportDataSet */

void trimTrailingBlanks /* Strip off trailing blanks */
(void *string /* - in: string to be trimmed */)
/*********************************************************************/
/* Strips trailing blanks from a string */
*********************************************************************/
{
    int i;
    for( i = strlen(string) - 1; string[i] == ' '; i-- );
    string[++i] = '\0';
} /* end of trimTrailingBlanks */

void stripContinuationCharacter /* Strip off trailing - or + */
(void *string /* - in: string to be trimmed */)
/*********************************************************************/
/* Strips trailing '+' or '-' from a blank-trimmed string */
*********************************************************************/
{
    int i;
    i = strlen(string) - 1;
    if( string[i] == '+' || string[i] == '-' )
        string[i] = '\0';
    trimTrailingBlanks( string );
} /* end of stripContinuationCharacter */

void processSqlCommit( void ) /* Commit SQL changes */
/*********************************************************************/
/* Commits the current unit of SQL work */
*********************************************************************/
{
    EXEC SQL
        COMMIT;
    if( SQLCODE != 0 )
        issueSqlError( "*** Commit failed " );
    else
        printf( "** SQL changes have been committed\n" );
} /* end of processSqlCommit */

void processSqlRollback( void ) /* Rollback SQL changes */
/*********************************************************************/
/* Rolls back the current unit of SQL work */
*********************************************************************/
{
    EXEC SQL
        ROLLBACK;
    if( SQLCODE != 0 )
        issueSqlError( "*** Rollback failed " );
    else
        printf( "** SQL changes have been rolled back\n" );
} /* end of processSqlRollback */

void issueDataSetClosingError /* Handler for ds close error */
(char *DDname,          /* - in: name of errant DD */
    int LEerrno           /* - in: LE diagnostic errno */)
/*********************************************************************/
/* Called when a TSO data set cannot be closed */
*********************************************************************/
Chapter 3. Db2 SQL programming

void issueDataSetOpeningError /* Handler for ds open error */
(char *DDname, /* - in: name of errant DD */
int LEerrno /* - in: LE diagnostic errno */
)
#endif

void issueDataSetReadingError /* Handler for ds read error */
(char *DDname, /* - in: name of errant DD */
int LEerrno /* - in: LE diagnostic errno */
)
#endif

void issueDataSetClosingError /* Handler for ds close error */
(short invalidDDnum /* - in: invalid DD sequence no. */
)
#endif

void issueInvalidCallParmCountError /* Handler for parm count error */
(int argc /* - in: no. parms received */
)
#endif

void issueInvalidDDnumError /* Handler for unknown DD seq */
(short invalidDDnum /* - in: invalid DD sequ. no. */
)
#endif

void issueInvalidActionError /* Handler for unknown action */
(char *action /* - in: action specified */
)
#endif
void issueInvalidParmLengthError  /* Handler for parm len error */
( char *parmName,  /* - in: identify of parm */
  int minLength,  /* - in: min valid length */
  int maxLength,  /* - in: max valid length */
  int *error_len );        /* Error code */
} /* end of issueInvalidParmLengthError */

void issueInvalidLevelError  /* Handler for wrong DSNTPSMP */
( char *level,  /* - in: level encountered */
  int *len );        /* Error code */
} /* end of issueInvalidLevelError */

#pragma linkage(dsntiar, OS)
void issueSqlError  /* Handler for SQL error */
( char *locMsg,  /* - in: Call location */
  int *len );        /* Error code */
} /* end of issueSqlError */

short int DSNTIARrc;  /* DSNTIAR Return code */
int j;  /* Loop control */
static int lrecl = 80;  /* Width of message lines */

void issueInvalidParmLengthError( char *parmName, int minLength, int maxLength, int error_len );
void issueInvalidLevelError( char *level, int error_len );
void issueSqlError( char *locMsg, int *len );

#define DSNTPSMP 118

void issueInvalidParmLengthError( char *parmName, int minLength, int maxLength, int *error_len )
{
  if ( DSNTIARrc == 0 )
  {
    printf( "The length of the argument specified for the %s ");
    printf( "parameter\n", parmName );
    printf( "does not fall within the required bounds of %i ");
    printf( "and %i\n", minLength, maxLength );
    printf( "----- > Processing halted\n" );
    rc = RETSEV;
  }
  else
  {
    printf( "The length of the argument specified for the %s ");
    printf( "parameter\n", parmName );
    printf( "does not fall within the required bounds of %i ");
    printf( "and %i\n", minLength, maxLength );
    printf( "----- > Processing halted\n" );
    rc = RETSEV;
  }
}

void issueInvalidLevelError( char *level, int error_len )
{
  if ( DSNTIARrc == 0 )
  {
    printf( "ERROR: The DSNTPSMP interface level %s is not ");
    printf( "supported by this client\n", level );
    printf( "----- > Processing halted\n" );
    rc = RETSEV;
  }
  else
  {
    printf( "ERROR: The length of the argument specified for the %s ");
    printf( "parameter\n", level );
    printf( "does not fall within the required bounds of %i ");
    printf( "and %i\n", minLength, maxLength );
    printf( "----- > Processing halted\n" );
    rc = RETSEV;
  }
}

void issueSqlError( char *locMsg, int *len )
{
  if ( DSNTIARrc == 0 )
  {
    printf( "ERROR: The length of the argument specified for the %s ");
    printf( "parameter\n", locMsg );
    printf( "does not fall within the required bounds of %i ");
    printf( "and %i\n", minLength, maxLength );
    printf( "----- > Processing halted\n" );
    rc = RETSEV;
  }
  else
  {
    printf( "ERROR: The length of the argument specified for the %s ");
    printf( "parameter\n", locMsg );
    printf( "does not fall within the required bounds of %i ");
    printf( "and %i\n", minLength, maxLength );
    printf( "----- > Processing halted\n" );
    rc = RETSEV;
  }
}
Related reference
“Sample programs to help you prepare and run external SQL procedures” on page 307

Db2 provides sample jobs to help you prepare and run external SQL procedures. All samples are in data set DSN1210.SDSNSAMP. Before you can run the samples, you must customize them for your installation.

DSN8WLMP
This JCL can be customized to establish the WLM startup PROC needed to run DSNTPSMP, the Db2 SQL Procedures Processor, and to run ADMIN_UPDATE_SYSPARM, the Db2 stored procedure that changes subsystem parameters.

```/* *********************************************************************
  * Name = DSN8WLMP
  * Description Name = DB2 Sample WLM startup PROC for DSNTPSMP, the DB2 SQL Procedures Processor, and for ADMIN_UPDATE_SYSPARM, the DB2 stored procedure that changes subsystem parameters.
  * Licensed Materials - Property of IBM
  * 5635-D2B
  * (C) COPYRIGHT 1982, 2006 IBM Corp. All Rights Reserved.
  * STATUS = Version 11
  * Function = This JCL can be customized to establish the WLM startup PROC needed to run DSNTPSMP, the DB2 SQL Procedures Processor, and to run ADMIN_UPDATE_SYSPARM, the DB2 stored procedure that changes subsystem parameters.
  * Before you can use this procedure, you need to have defined a WLM Application Environment for running DSNTPSMP and ADMIN_UPDATE_SYSPARM.
  * *** *** *** *** *** IMPORTANT *** *** *** *** *** *** ***
  * For DSNTPSMP and ADMIN_UPDATE_SYSPARM, NUMTCB=1 is required.
  * Specify no other value. This assures concurrent executions of DSNTPSMP and ADMIN_UPDATE_SYSPARM will run in their own address space, which is needed for proper dataset operation from within a REXX/TSO DB2 stored procedure.
  * (1) Customize this proc for use on your system by locating and changing all occurrences of the following strings as indicated:
    (A) '!WLMENV!' to the name of the WLM Application Environment you have chosen for running DSNTPSMP and ADMIN_UPDATE_SYSPARM
    (B) '!DSN8WLMP!' to the name of the WLM Procedure associated with that environment
    (C) '!DSN!' to the name of your DB2 subsystem
    (D) 'CBC!!' to the prefix of your target library for IBM C/C++ for z/OS
    (E) 'CEE!!' to the prefix of your target library for IBM Language Environment for z/OS
    (F) 'DSN!!0' to the prefix of your target library for DB2 for z/OS
  * (2) Copy the customized proc to your MVS proclib, to the member you specified as the WLM procedure name for the WLM application environment you have chosen for running DSNTPSMP and ADMIN_UPDATE_SYSPARM
  * Note: This should be the same value as you specified in step 1B, above.
  * CHANGE LOG:
    09/20/2012 Add ZPMDFLTS for ADMIN_UPDATE_SYSPARM DK1557/PM71114
    ********************************************************************************
    !DSN8WLMP! PROC DB2SSN=!DSN!,NUMTCB=1,APPLENV=!WLMENV!
    ********************************************************************************
    NUMTCB=1 SET NUMTCB=          <= Null NUMTCB symbol
    DSNTPSMP EXEC PGM=DSNX9WLM,TIME=1440, PARM='&DB2SSN,1,&APPLENV', <= Use 1, not NUMTCB
*/ ```
// REGION=OM,DYNAMNBR=5              <== Allow for Dyn Allocations
// Include SDSNEXIT to use Secondary Authids (DSN3@ATH DSN3@SGN exits)
//STEPLIB DD DISP=SHR,DSN=DSN!!0.SDSNEXIT
//                DD DISP=SHR,DSN=DSN!!0.SDSNLOAD
//                DD DISP=SHR,DSN=CBC!!:SCCNMP
//                DD DISP=SHR,DSN=CCE!!:SCERUN
//SYSEXEC DD DISP=SHR,
//                DSN=DSN!!0.SDSNCLST
//SYSSTSPRT DD SYSSOUT=*  
//CEEDUMP DD SYSSOUT=*  
//SYSPRINT DD SYSSOUT=*  
//SYSAEND DD DUMMY
//DSNTRACE DD  
//* **** Data sets required by the SQL Procedures Processor
//SQLDBRM DD DISP=SHR,  
//                DSN=DSN!!0.DBRMLIB.DATA
//SQLCSRC DD DISP=SHR,  
//                DSN=DSN!!0.SRCLIB.DATA
//SQLMOD DD DISP=SHR,  
//                DSN=DSN!!0.RUNLIB.LOAD
//SQLIBC DD DISP=SHR,  
//                DSN=CCE!!:SCEEH.H
//                DD DISP=SHR,  
//                DSN=CCE!!:SCEEH.SYS.H
//                DD DISP=SHR,  
//                DSN=DSN!!0.SDSNC.H
//                DD DISP=SHR,  
//                DSN=DSN!!0.SDSNLOAD
//SYMSGS DD DISP=SHR,  
//                DSN=CCE!!:SCEMSGP(EDCPMSG)
//*/
//**** DSNTPSMP Configuration File - CFGTPSMP (optional)
//**             A site provided sequential dataset or member, used to
//**             define customized operation of DSNTPSMP in this APPLENV.
//*/
//**** Workfiles required by the SQL Procedures Processor
//SQLSRC DD UNIT=SYSALLDA,SPACE=(23440,(20,20)),
//                DCB=(RECFM=FB,LRECL=80,BLKSIZE=23440)
//SQLPRINT DD UNIT=SYSALLDA,SPACE=(23476,(20,20)),
//                DCB=(RECFM=FB,LRECL=137,BLKSIZE=23476)
//SQLTERM DD UNIT=SYSALLDA,SPACE=(23476,(20,20)),
//                DCB=(RECFM=VB,LRECL=137,BLKSIZE=23476)
//SQLOUT DD UNIT=SYSALLDA,SPACE=(23476,(20,20)),
//                DCB=(RECFM=VB,LRECL=137,BLKSIZE=23476)
//SQLCERT DD UNIT=SYSALLDA,SPACE=(23476,(20,20)),
//                DCB=(RECFM=VB,LRECL=137,BLKSIZE=23476)
//SQLDUMP DD UNIT=SYSALLDA,SPACE=(32000,(20,20)),
//                DCB=(RECFM=FB,LRECL=80,BLKSIZE=32000)
//SYSDUMMY DD DUMMY
//** **** Data sets required by ADMIN_UPDATE_SYSPARM
//ZPMDFLTS DD DISP=SHR,  
//                DSN=DSN!!0.NEW.SDNSAMP(DSNADMZW)
//*/

Related reference

“Sample programs to help you prepare and run external SQL procedures” on page 307
Db2 provides sample jobs to help you prepare and run external SQL procedures. All samples are in data
set DSN1210.SDNSAMP. Before you can run the samples, you must customize them for your installation.

**DSN8ED5**
Demonstrates how to call the sample SQL procedure DSN8.
Chapter 3. Db2 SQL programming 325
/***************************** Equates ******************************/
#define NULLCHAR '\0' /* Null character */
#define OUTLEN 80 /* Length of output line */
#define DATA_DIM 10 /* Number of message lines */
#define NOT_OK 0 /* Run status indicator: Error*/
#define OK 1 /* Run status indicator: Good */

/**************** DB2 SQL Communication Area *******************/
EXEC SQL INCLUDE SQLCA;

/**************** DB2 Host Variables *******************/
EXEC SQL BEGIN DECLARE SECTION;
char locationName[17]; /* Server location name */
decimal(15,2) hvBonusBase = 0; /* base bonus for managers */
short int niBonusBase = 0; /* Indic var for hvBonusBase */
decimal(15,2) hvBonuses = 0; /* tot bonuses rtnd by DSN8ES2*/
short int niBonuses = 0; /* Indic var for hvBonuses */
long int hvSqlErrCd = 0; /* Err SQLCODE from DSN8ES2 */
short int niSqlErrCd = 0; /* Indic var for hvSqlErrCd */
EXEC SQL END DECLARE SECTION;

/**************** DB2 Message Formatter *******************/
struct {short int error_len; char error_text[DATA_DIM][OUTLEN];} error_message = {DATA_DIM * (OUTLEN)};
#pragma linkage( dsntiar, OS )
extern short int dsntiar( struct sqlca *sqlca, struct error_struct *msg, int *len );

/**************** DSN8ED5 Global Variables *******************/
short int status = OK; /* DSN8ED5 run status */
long int completion_code = 0; /* DSN8ED5 return code */

/**************** DSN8ED5 Function Prototypes *******************/
int main( int argc, char *argv[] );
void sql_error( char locmsg[] );

int main( int argc, char *argv[] )
{
    printf( "**** DSN8ED5: Sample client for DB2 SQL Procedure Sample (DSN8.DSN8ES2)\n\n" );
    printf( "\n\n" );
    if( argc < 2 || argc > 3 )
    {
        printf( "DSN8ED5 failed: Invalid parameter count\n" );
        status = NOT_OK;
    }
    else if( strlen(argv[1]) > 9 )
    {
        printf( "DSN8ED5 failed: Bonus base exceeds 9 bytes. Use format: nnnnnn.nn\n" );
        status = NOT_OK;
    }
    else
    {
        /* Convert the input parameter from a string to a decimal */
        hvBonusBase = atof( argv[1] );
    }
    printf( "\n\n" );
    if( status != OK )
    {
        printf( "DSN8ED5 failed: "%s\n" );
        if( status == NOT_OK )
            printf( "Invalid parameter count\n" );
    }
    return( status );
}

/**************************** DSN8ED5: Sample client for DB2 SQL Procedure Sample */
/**************************** DSN8.DSN8ES2*\n\n*/
/**************************** DSN8ED5: Sample client for DB2 SQL Procedure Sample */
/**************************** DSN8.DSN8ES2*

326  Db2 12 for z/OS: Application Programming and SQL Guide
/* Validate remote location name, if one is specified */

* Connect to the remote location, if one was specified *

* Process the call to DSN8.DSN8ES2 *

/* SQL error handler */

void sql_error( char locmsg[] ) /*proc*/

*********************************************************************/

**********************************************************************

** SQL error handler **

**********************************************************************

/** SQL error handler **

**********************************************************************

void sql_error( char locmsg[] ) /*proc*/

short int rc;  /* DSNTIAR Return code */
int j,k;       /* Loop control */
static int lrecl = OUTLEN;  /* Width of message lines */

Chapter 3. Db2 SQL programming
/**
 * set status to prevent further processing
 * *********************************************************************/
status = NOT_OK;

/*********************************************************************/
* print the locator message
* printf(" %.80s
", locmsg);

/*********************************************************************/
* format and print the SQL message
* rc = dsntiar( &sqlca, &error_message, &lrecl );
if( rc == 0 ) {
for( j=0; j<DATA_DIM; j++ ) {
  for( k=0; k<OUTLEN; k++ )
    putchar(error_message.error_text[j][k] );
  putchar('\n');
}
else {
  printf(" *** ERROR: DSNTIAR could not format the message\n" );
  printf(" *** SQLCODE is %d\n",SQLCODE );
  printf(" *** SQLERRM is \n" );
  for( j=0; j<sqlca.sqlerrml; j++ )
    printf("%c", sqlca.sqlerrmc[j] );
  printf("\n" );
}
} /* end of sql_error */

Related reference
“Sample programs to help you prepare and run external SQL procedures” on page 307
Db2 provides sample jobs to help you prepare and run external SQL procedures. All samples are in data
set DSN1210.SDSNSAMP. Before you can run the samples, you must customize them for your installation.

DSNTEJ367
This job demonstrates two important steps to follow when considering the conversion of an external SQL
procedure to a native SQL procedure.
/* Define Input. Identify the name of an external SQL SP with */
/* source saved in DB2 (SYSIBM.SYSROUTINES_SRC).
*/
// PH067S02 Step
/* Define Output. Specify an output data set where the extracted */
/* and modified SQL SP source is to be placed.
*/
// PH067S03 Step
/* Setup the sample REXX services to used for this job.
*/
// PH067S04 Step
/* Execute the DSNTEJ67 sample conversion process.
*/
/*   - Validate the SP name              (using the NAMPARTS service)
*/
/*   - Verify the output file is usable  (using the CHKANYVF service)
*/
/*   - Deploy DSNBEN1, a sample native */
/*     SQL SP helper for use later       (using the CRSQLPL service)
*/
/*   - Extract external SQL SP source   (using the SQLPLSRC service)
*/
/*   - Save the source in the output file */
/*     - for a RECFM V output file       (using the ANY2SQLV service)
*/
/*   - for a RECFM F output file       (using the SQLV2F service)
*/
/*   - Validate and inspect the source  (using the CHKSQPL service)
*/
/*   - Produce a table of contents to */
/*     describe the DDL syntax elements
*/
/*     present in the SQL PL source      (using the SQLPLTOC service)
*/
/*   - Dissect the external SQL SP source
*/
/*     removing all the SP options
*/
/*   - Get the replacement options for */
/*     native SQL PL use by calling the */
/*     helper SQL PL deployed earlier    (using the SQLCALL service)
*/
/*   - Reassemble the SQL SP source as a */
/*     string and write it to a RECFM V
*/
/*     temporary file (aka SQLV)
*/
/*   - Update the output file
*/
/*     - for a RECFM V output
*/
/*     - for a RECFM F output
*/
/*   - Write a special format temp file
*/
/*     for the precompiler
*/
/*   - Obtain a HOST(SQLPL) Checkout */
/*     precompiler listing of the SQL SP
*/
/*     source for job log output.        (using the CHKSQPL service)
*/
/*   - Set the Job step RC.
*/
/* Dependencies =
*/
/* (1) Run sample job DSNTEJ65 prior to running this job.
*/
/* That job uses the DB2 SQL procedure processor DSNTPSMP to */
/* deploy the sample external SQL procedure DSN0.DSN0ES2, which
*/
/* is the external SQL Procedure this job processes by default.
*/
/* Note: Run this job at the same site where DSNTEJ65 created */
/* DSN0.DSN0ES2. Otherwise this job will terminate in */
/* Job step PH067S04 with rc=8 and the following messages:
*/
/*   *SQLPLSRC* Error obtaining Source, SQLPL procedure
*/
/*   was not found
*/
/*   *SQLPLSRC* RC=6
*/
/* DSNTEJ67 Cannot extract SQL procedure source
*/
/* Notes =
*/
/* Prior to running this job, customize it for your system:
*/
/* (1) Add a valid job card
*/
/* (2) Locate and change all occurrences of the following strings
*/
/* as indicated:
*/
/* (A) 'DSN!' to the subsystem name of your DB2. This is */
/* located in Step 0.
*/
/* (B) 'DSN!0' to the prefix of the target library for the */
/* current DB2 release. This is located in the
*/
/* JOBLIB, Step 3 and Step 4.
*/
/* (3) (Optional) Change either of the following to customize the
*/
/* input and output of this job for your particular purposes:
*/
/* (A) Change the name of the external SQL procedure to be */
/* processed by this job. This is defined in job Step 1. */
/* The name must include the schema qualifier. It must
*/
/* designate an operational external SQL SP which was
*/
/* deployed using the DB2 SQL procedure processor DSNTPSMP.
*/
/* (B) Change the data set where the extracted and modified
*/
/* SQL procedure source will be written. This is defined in */
/* job Step 2. The specification can be for an existing
*/
/* data set or data set member, qualified or not qualified.
*/
/* or represented by a DD descriptor (in the form of
*/
/* DD:ddname). Any existing data set or ddname allocation
*/
/* must be for a RECFM=F,FB,V,VB sequential data set or
*/
/* data set member. (If an unallocated ddname is provided
*/
/* Chapter 3. Db2 SQL programming 329
a temporary sequential data set will be allocated.)

Change Activity = ( V11 base pm76443 )
Apr2013 - Add version activation of native SQL PL helper routine
Aug2013 - Clarify prolog notes on DSN8ES2 dependency

DD DISP=SHR,DSN=DSN!0.SDSNEXIT
DD DISP=SHR,DSN=DSN!0.SDSNLOAD

Step 0: Store the default DB2 System SSID in a temporary data set
for use by various steps and services that are run.

JOBLIB   DD  DISP=SHR,DSN=DSN!!0.SDSNEXIT
          DD  DISP=SHR,DSN=DSN!!0.SDSNLOAD

Step 1: Store the desired external SQL SP name to process.
Specify a fully qualified SP name (2-parts, schema+name).

Step 2: Identify the desired data set name to store the extracted
and modified SQL procedure source. Must be Recfm F or V,
sequential or member, or a non-existing data set/member.
The specification can be a qualified name, a non-qualified
name or a DD descriptor (in the form of DD:ddname).

Step 3: Populate a temporary PDS with REXX services used locally
parse value "NAMPARTS" ( spname ) with p#. namSpec
if p# <> 2 then do;
say 'DSNTEJ67 the passed SP name' spname,
    'was not schema qualified (2-parts)'
exit 8;
end;
parse var namSpec a b c d e . ':' +1 sNam +(a) sSch +(b) . +(c),
    qNam +(d) qSch +(e)
spname = qSch'.qNam             /* 2-part fully qualified form now */

/* From DD:SOURCEDS read the data set specification for where the */
/* extracted and modified SQL proc source should be written at JOB end. */
'EXECIO * DISKR SOURCEDS (OPEN FINIS STEM TEMP.);
sourceFile='';
do i = 1 to TEMP.0;
    sourceFile = sourceFile || TEMP.i;
end;
sourceFile = "STRIP"(sourceFile,'B');

/* Find the status of the target data set for the source. It must be a */
/* Sequential data set, F or V record format (or capable of same). */
parse value "CHKANYFV"(sourceFile) with oRecfm oLrecl oEmpty;
if oRecfm='' then do;
say 'DSNTEJ67 Cannot use the designated data set' sourceFile,
    'for SQL PL source'
exit 8;
end;

prepConv:
/* From DD:HELPRSP1 deploy the native SQL SP DSN8.DSN8EN1 that will */
/* provide native options for the external SQL proc to be migrated. */
/* Use the subroutine form of the CRSQLPL service, asking for return */
/* of a version activation statement, for processing after deployment. */
/* Say 'Setting up the native SQL PL helper routine...' */
call "CRSQLPL" 'DD:HELPRSP1', , , 'RETACTIVATE';
parse var result rc . 1 tok1 VerStmt ;
if "DATATYPE"(rc,'W')=0 then select;  /* OK, 1st word not a number */
    when tok1='/*CP*/' then rc=0;       /* Skip ACTIVATE for CREATE */
    when "WORDPOS"(tok1, '//*APR*/ /*APA*/')>0 /* ALTER REP, ALTER ADD */
        then rc = ActivateRtnVer( VerStmt );
    otherwise rc=8;                     /* Problem, force an error. */
end /* select */;
if RC>4 then do;
say 'DSNTEJ67 cannot deploy the native SP used for migration.';
exit 8;
end;

extractSQLproc:
/* Extract the SP source to a temporary SQLV file. Also get an S80 */
/* format edition to use with the precompiler for inspection purposes. */
/* 'SQLPLSRC' spname 'DD:SQLVE' 'DD:S80E' 'ASIS';
if RC>4 then do;
say 'DSNTEJ67 Cannot extract SQL procedure source';
exit 8;
end;
allocList='SQLVE,S80E';               /* allocated FOR us */
/* Write extracted source ASIS now to the output data set. */
/* We will rewrite it again later after reaching the point of editing. */
if oRecfm='F' then do;
    if oLrecl=80 then seq='SEQ'; else seq '';
    "SQLV2F" 'DD:SQLVE' sourceFile seq
end;
else "ANY2SQLV" 'DD:SQLVE' sourceFile 'EXTEND'
if RC>0 then do;
say 'DSNTEJ67 Cannot write extracted source to data set' sourceFile
exit 8;
end;
else say 'Source for SP' spname 'written to data set' sourceFile
verifyExtSQL:
/* Verify the extracted source is valid external SQL procedure source */
/* before going to far. Use the HOST(SQL) precompiler. */
/* Keep DD:LISTING active till then end. */
'ALLOCATE DDNAME (LISTING) NEW REUSE';
'CHKSQLPL' 'DD:S80E' 'DD:LISTING' '.' 'MAR(1,80) HOST(SQL)'
if RC>4 then do;
    msg='DSNTEJ67 Extracted external SQL procedure source has errors';
call endWithListing msg, allocList;
end;

chkoutSQLPL1:
/* Inspect the extracted external SQL procedure source without change
 * using the HOST(SQLPL) Checkout precompiler.
 * RC=8 errors are anticipated.
 */
allocList = allocList||',';UT1';
'ALLOCATE DDNAME(UT1) NEW REUSE';
'CHKSQLPL' 'DD:S80E' 'DD:LISTING' 'DD:UT1' 'MAR(1,80) HOST(SQLPL)
if RC>8 then do;
    msg='DSNTEJ67 Fatal error running HOST(SQLPL) precompiler',
    'with external SQL procedure source'
call endWithListing msg, allocList;
end;
/* Getting no UT1 content typically represents a native SQL PL syntax
 * issue. In this context, that could be caused by some unforeseen
 * difference between valid external SQL PL and native SQL PL syntax.
 */
parses Value "CHKANYFV"('DD:UT1') with utR . utE;
if utE<>1 then do;
    msg='DSNTEJ67 Syntax error in external SQL proc source',
    'when viewed as native SQL PL';
call endWithListing msg, allocList;
end;
editPrep:
/* Obtain an SQLPL TOC description, to use for editing the source. */
'SQLPLTOC' 'DD:S80E' 'DD:UT1' 'DD:TOC'
if RC<>0 then do;
    msg='DSNTEJ67 Unable to prepare for source editing (no TOC).'
call endWithListing msg, allocList;
end;
allocList = allocList||',';TOC';
/* Read TOC to get the OPTIONS element descriptor */
opts=''
"EXECIO * DISKR SQLVE (OPEN FINIS STEM TOC.

for i = 1 to TOC.0;
parses var TOC.i elem desc;
if elem='OPTS:' then do;
    opts = desc;
    leave;
end;
end;
parses var opts o1 o2 o3 .
parses var o1 or1 ':' oc1; parses var o2 or2 ':' oc2;
/* Bring original external source into memory now, splitting into
 * three parts, Front (ahead of options), Back (after options)
 * and Middle (the options which will be replaced).
 */
"EXECIO 0 DISKR SQLVE (OPEN"
/* Front: all complete lines before options, into stem FR. */
FR.=''; FR.0=0;
if or1>1
then "EXECIO" or1-1 "DISKR SQLVE (STEM FR.
/* Middle: All records that have options on them
 * This will be at least one record (where options WOULD go).
 */
if o3='0'
/* no options were present */
then "EXECIO" 1 "DISKR SQLVE (STEM MD.";
else "EXECIO" 1+or2-or1 "DISKR SQLVE (STEM MD.
/* Back: all the remaining complete lines */
BK.=''; BK.0=0;
"EXECIO * DISKR SQLVE (FINIS STEM BK.
/* The Middle likely has portions of the Front, the Back, or both.
 * Separate those now, and then toss the middle. Process Back first.
 */
if Frm=''
then Frm=''; /* collapse existing option indentation */
i=MD.0;
if o3='0'
then j=oc1; /* When no options, the BK middle starts at oc1. */
else j=oc2+1; /* With options, the BK middle starts after oc2. */
parses var MD.1 MD.1=('j) BKm
if oc1<2
then Frm='';
else parses var MD.1 Frm =(oc1) MD.1

If MD.0 > 0 then do;
say 'Removing these external SQL procedure options:';
do i = 1 to MD.0; say MD.i; end;
end;
drop MD. TOC.  
/* We now have the external source in stems FR., BK.  
* and strings FRm and BKm. Release the old options.  
* Free our allocated data sets now. Current LISTING remains...  
*/  
'FREE DDNAME('allocList')';
allocList='';
/* Get the replacement options from the helper routine DSN8.DSN8EN1  
* Use the FUNCTION invocation of the SQLCALL service to obtain  
* the value of the last parameter (the SP output parm).  
*/  
call "OUTTRAP" 'TEMP.';
nat_opt=SQLCALL("DSN8.DSN8EN1('"sSch"', '"sNam"', VARCHAR('?', 5120))");
call "OUTTRAP" 'OFF';
if nat_opt='' | nat_opt='8' then do;
  msg='DSNTEJ67 Unable to obtain native options for replacement';
call endWithListing msg ;
end;
say 'Inserting these native SQL PL options:';
temp=nat_opt;
do while temp<>'';
  parse var temp opn '25'x temp;
  say opn;
end;
/* Rewrite Original source using the new Native Options */
new_src='';
do i = 1 to FR.0; new_src=new_src || FR.i || '25'x; end; drop FR.
new_src=new_src || FRm               ; drop FRm
new_src=new_src || nat_opt ; drop nat_opt
new_src=new_src || BKm               ; drop BKm
new_src=new_src || BK.0; new_src=new_src || BK.i || '25'x; drop BK.i;  end;
call "STR2SQLV" new_src, 'DD:SQLV'
if oRecfm='F' then do;
  if oLrecl=80 then seq='SEQ'; else seq='';
  "SQLV2F" 'DD:SQLV' sourceFile seq
end;
else "ANY2SQLV" 'DD:SQLV' sourceFile 'EXTEND'
"SQLV2F" 'DD:SQLV' 'DD:S80' 'S80'
chkoutSQLPL2:
/* Inspect the modified procedure source one last time  
* using the HOST(SQLPL) Checkout precompiler.  
*/
'CHKSQLPL' 'DD:S80' 'DD:LISTING' '. ' 'MAR(1,80)'
rrc=RC;
say 'Final HOST(SQLPL) Checkout Precompile ended with RC='rrc;
if rrc=0 then say 'Inspect the Listing for'
  'additional SQLPL source coding issues';
'EXECIO * DISKR LISTING (OPEN FINIS STEM LISTING.);'
call trimListing;
'EXECIO' listing.0 'DISKW LISTOUT (OPEN FINIS STEM LISTING.);'
'FREE DDNAME(LISTING,SQLV,S80)';
exit rRC;
/* ----------------------------------------------------------------- */
activateRtnVer: procedure
/* Process the Activate Version statement passed. Returns 0 or 4. */
parse arg AVstmt ;
rcod=0;
"EXECIO * DISKR DB2SSID (OPEN FINIS STEM TEMP."
parse var TEMP.1 ssid ;
say 'Issuing... AVstmt';
call SQLDBS "ATTACH TO" ssid;
call SQLEXEC "EXECUTE IMMEDIATE :AVSTMT";
if result<0 then do;
say 'Trouble activating the native SQL PL routine version';
say '==>'sqlca.sqlcode'<';
say '==>'sqlca.sqlerrm'<';
call SQLEXEC "ROLLBACK";
rccd=4;
end;
else call SQLEXEC "COMMIT";
call SQLDBS 'DETACH';
return rcod;

Chapter 3. Db2 SQL programming 333
endWithListing:
    'EXECIO * DISKR LISTING (OPEN FINIS STEM LISTING.';
endWithListing:
    call trimListing;
    'EXECIO' listing.0 'DISKW LISTOUT (OPEN FINIS STEM LISTING.';
    FREE DDNAME('LISTING')
    if arg(2,'E')
        then 'FREE DDNAME('arg(2)')'; /* other DDnames to free */
    if arg(1,'E')
        then say arg(1); /* Message to end with */
    exit 0;
/* trimListing: Reduce the occurrence of repeated headers, CC and page
   * numbers in the listing, so it appears like one
   * continuous stream. */
trimListing: procedure expose LISTING.
    hdrtypes = 'VERSION SYMBOL MESSAGES STATISTICS';
    j=0; k=LISTING.0; do i = 1 to k;
        parse var LISTING.i 1 cc +1 line;
        if cc='1' & "LEFT"(line,19)='DB2 SQL PRECOMPILER' then do; /*Hdr*/
            key="WORD"(line,4);
            loc="WORDPOS"(key,hdrtypes);
            if loc>0 then do;                        /* First hdr usage.   */
                parse var line line 'PAGE' .           /* Keep, w/o page num.*/
                hdrtypes = "DELWORD"(hdrtypes,loc,1);  /* Header now used. */
            end;
            else iterate;                            /* Skip redundant hdr.*/
        end /*Hdr*/;
        j=j+1; LISTING.j="STRIP"(line,'T');
    end /* do i ... */;
    do i=j+1 to k; drop LISTING.i; end;
    LISTING.0=j;
    return 1+k-j; /* lines trimmed */
    /* ------------------- end DSNTEJ67 -------------------------------- */
@@.//  ENDUP
//*********************************************************************
//*  Step 4:  Run the sample process to extract external SQL SP source
//*           from DB2, convert the source to native SQL PL, save the
//*           modified source in a data set and finish with a source
//*           listing written to the job log.
//*********************************************************************
//* PH067S04 EXEC PGM=IKJEFT01,DYNAMNBR=30
//* SYSEXEC  DD DSN=&&REXXPDS,DISP=(OLD,PASS)
//* SYSTSPRT DD SYSOUT=*
//* DB2SSID  DD DSN=&&PARM0,DISP=(OLD,PASS)
//* SPNAME   DD DSN=&&PARM1,DISP=(OLD,PASS)
//* SOURCEDS DD DSN=&&PARM2,DISP=(OLD,PASS)
//* LISTOUT  DD SYSOUT=*
//* HELPRSP1 DD DSN=DSN!0.SDSNIVPD(DSN8EN1),
//*            DISP=SHR
//* SYSTSIN  DD *
//%DSNTEJ67
/*
Related reference
“Sample programs to help you prepare and run external SQL procedures” on page 307
Db2 provides sample jobs to help you prepare and run external SQL procedures. All samples are in data
set DSN1210.SDSNSAMP. Before you can run the samples, you must customize them for your installation.

Creating multiple versions of external procedures
For native SQL procedures, you can use Db2 to create and maintain multiple versions of the procedure. However, for external procedures including external SQL procedures, if you need multiple versions of a
procedure, you need to maintain them manually.

Before you begin
Deprecated function: External SQL procedures are deprecated and not as fully supported as native
SQL procedures. For best results, create native SQL procedures instead. For more information, see

334  Db2 12 for z/OS: Application Programming and SQL Guide
Procedure

To create multiple versions of external procedures, including external SQL procedures, use one of the following techniques:

- Define multiple procedures with the same name in different schemas. You can subsequently use the SQL path to determine which version of the procedure is to be used by a calling program.
- Define multiple versions of the executable code. You can subsequently use a particular version by specifying the name of the load module for the version that you want to use on the EXTERNAL clause of the CREATE PROCEDURE statement or ALTER PROCEDURE statement.
- Define multiple packages for a procedure. You can subsequently use the COLLID option, the CURRENT PACKAGESET special register, or the CURRENT PACKAGE PATH special register to specify which version of the procedure is to be used by the calling application.
- Set up multiple WLM environments to use different versions of a procedure.

Adding and modifying data in tables from application programs

Your application program can add, modify, or delete data in any Db2 table for which you have the appropriate access.

Inserting data into tables

You can use several different methods to insert data into a table. Decide which method to use based on the amount of data that you need to insert and the other operations that your program needs to perform.

About this task

Besides using stand-alone INSERT statements, you can use the following ways to insert data into a table:

- You can use the MERGE statement to insert new data and update existing data in the same operation.
- You can write an application program to prompt for and enter large amounts of data into a table.
- You can also use the Db2 LOAD utility to enter data from other sources.

Related tasks

Inserting data and updating data in a single operation
You can update existing data and insert new data in a single operation. This operation is useful when you want to update a table with a set of rows, some of which are changes to existing rows and some of which are new rows.

Related reference

LOAD (Db2 Utilities)

Inserting rows by using the INSERT statement

One way to insert data into tables is to use the SQL INSERT statement. This method is useful for inserting small amounts of data or inserting data from another table or view.

About this task

Procedure

Issue an INSERT statement.

By using INSERT statements, you can do the following actions:
• Specify the column values to insert a single row. You can specify constants, host variables, expressions, DEFAULT, or NULL by using the VALUES clause.

• In an application program, specify arrays of column values to insert multiple rows into a table. Use host-variable arrays in the VALUES clause of the INSERT FOR \( n \) ROWS statement to add multiple rows of column values to a table.

• Include a SELECT statement in the INSERT statement to tell Db2 that another table or view contains the data for the new row or rows.

In each case, for every row that you insert, you must provide a value for any column that does not have a default value. For a column that meets one of the following conditions, specify DEFAULT to tell Db2 to insert the default value for that column:

• The column is nullable.

• The column is defined with a default value.

• The column has data type ROWID. ROWID columns always have default values.

• The column is an identity column. Identity columns always have default values.

• The column is a row change timestamp column.

The values that you can insert into a ROWID column, an identity column, or a row change timestamp column depend on whether the column is defined with GENERATED ALWAYS or GENERATED BY DEFAULT.

You can use the VALUES clause of the INSERT statement to insert a single row of column values into a table. You can either name all of the columns for which you are providing values, or you can omit the list of column names. If you omit the column name list, you must specify values for all of the columns.

**Recommendation:** For static INSERT statements, name all of the columns for which you are providing values for the following reasons:

• Your INSERT statement is independent of the table format. (For example, you do not need to change the statement when a column is added to the table.)

• You can verify that you are specifying the values in order.

• Your source statements are more self-descriptive.

If you do not name the columns in a static INSERT statement, and a column is added to the table, an error can occur if the INSERT statement is rebound. An error will occur after any rebind of the INSERT statement unless you change the INSERT statement to include a value for the new column. This is true even if the new column has a default value.

When you list the column names, you must specify their corresponding values in the same order as in the list of column names.

**Examples**

**Example**

The following statement inserts information about a new department into the YDEPT table.

```
INSERT INTO YDEPT (DEPTNO, DEPTNAME, MGRNO, ADMRDEPT, LOCATION)
VALUES ('E31', 'DOCUMENTATION', '000010', 'E01', ' ');
```

After inserting a new department row into your YDEPT table, you can use a SELECT statement to see what you have loaded into the table. The following SQL statement shows you all of the new department rows that you have inserted:

```
SELECT *
FROM YDEPT
WHERE DEPTNO LIKE 'E%'
ORDER BY DEPTNO;
```

The result table looks similar to the following output:

<table>
<thead>
<tr>
<th>DEPTNO</th>
<th>DEPTNAME</th>
<th>MGRNO</th>
<th>ADMRDEPT</th>
<th>LOCATION</th>
</tr>
</thead>
<tbody>
<tr>
<td>336</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Example
The following statement inserts information about a new employee into the YEMP table. Because the
WORKDEPT column is a foreign key, the value that is inserted for that column (E31) must be a value in
the primary key column, which is DEPTNO in the YDEPT table.

```
INSERT INTO YEMP
'MANAGER', 16, 'M', '1970-07-10', 24000, 500, 1900);
```

Example
The following statement also inserts a row into the YEMP table. Because the unspecified columns
allow null values, Db2 inserts null values into the columns that you do not specify.

```
INSERT INTO YEMP
(EMPNO, FIRSTNME, MIDINIT, LASTNAME, WORKDEPT, PHONENO, JOB)
VALUES ('000410', 'MILLARD', 'K', 'FILLMORE', 'D11', '4888', 'MANAGER');
```

Related concepts
Rules for inserting data into an identity column
An identity column contains a unique numeric value for each row in the table. Whether you can insert data
into an identity column and how that data gets inserted depends on how the column is defined.

Rules for inserting data into a ROWID column
A ROWID column contains unique values that identify each row in a table. Whether you can insert data
into a ROWID column and how that data gets inserted depends on how the column is defined.

Related tasks
Inserting multiple rows of data from host-variable arrays
Use host-variable arrays in your INSERT statement when you do not know at least some of the values to
insert until the program runs.

Inserting rows into a table from another table
You can copy one or more rows from one table into another table.

Related reference
INSERT (Db2 SQL)
CREATE TABLE (Db2 SQL)

**Inserting rows into a table from another table**
You can copy one or more rows from one table into another table.

Procedure
Use a fullselect within an INSERT statement.

**Examples**

**Example**
The following SQL statement creates a table named TELE:

```
CREATE TABLE TELE
(NAME2  VARCHAR(15) NOT NULL,
NAME1  VARCHAR(12) NOT NULL,
PHONE  CHAR(4));
```

The following statement copies data from DSN8C10.EMP into the newly created table:

```
INSERT INTO TELE
SELECT LASTNAME, FIRSTNME, PHONENO
```
The two previous statements create and fill a table, TELE, that looks similar to the following table:

<table>
<thead>
<tr>
<th>NAME2</th>
<th>NAME1</th>
<th>PHONE</th>
</tr>
</thead>
<tbody>
<tr>
<td>PULASKI</td>
<td>EVA</td>
<td>7831</td>
</tr>
<tr>
<td>JEFFERSON</td>
<td>JAMES</td>
<td>2094</td>
</tr>
<tr>
<td>MARINO</td>
<td>SALVATORE</td>
<td>3788</td>
</tr>
<tr>
<td>SMITH</td>
<td>DANIEL</td>
<td>0961</td>
</tr>
<tr>
<td>JOHNSON</td>
<td>SYBIL</td>
<td>8953</td>
</tr>
<tr>
<td>PEREZ</td>
<td>MARIA</td>
<td>9001</td>
</tr>
<tr>
<td>MONTEVERDE</td>
<td>ROBERT</td>
<td>3780</td>
</tr>
</tbody>
</table>

The CREATE TABLE statement example creates a table which, at first, is empty. The table has columns for last names, first names, and phone numbers, but does not have any rows.

The INSERT statement fills the newly created table with data that is selected from the DSN8C10.EMP table: the names and phone numbers of employees in department D21.

**Example**

The following CREATE statement creates a table that contains an employee's department name and phone number. The fullselect within the INSERT statement fills the DLIST table with data from rows that are selected from two existing tables, DSN8C10.DEPT and DSN8C10.EMP.

```
CREATE TABLE DLIST
(DEPT    CHAR(3)      NOT NULL,
  DNAME   VARCHAR(36)          ,
  LNAME   VARCHAR(15)  NOT NULL,
  FNAME   VARCHAR(12)  NOT NULL,
  INIT    CHAR                 ,
  PHONE   CHAR(4) );
```

```
INSERT INTO DLIST
SELECT DEPTNO, DEPTNAME, LASTNAME, FIRSTNME, MIDINIT, PHONENO
FROM DSN8C10.DEPT, DSN8C10.EMP
WHERE DEPTNO = WORKDEPT;
```

**Rules for inserting data into a ROWID column**

A *ROWID column* contains unique values that identify each row in a table. Whether you can insert data into a ROWID column and how that data gets inserted depends on how the column is defined.

A *ROWID column* is a column that is defined with a ROWID data type. You must have a column with a ROWID data type in a table that contains a LOB column. The ROWID column is stored in the base table and is used to look up the actual LOB data in the LOB table space. In addition, a ROWID column enables you to write queries that navigate directly to a row in a table. For information about using ROWID columns for direct-row access, see “Specifying direct row access by using row IDs” on page 432.

Before you insert data into a ROWID column, you must know how the ROWID column is defined. ROWID columns can be defined as GENERATED ALWAYS or GENERATED BY DEFAULT. GENERATED ALWAYS means that Db2 generates a value for the column, and you cannot insert data into that column. If the column is defined as GENERATED BY DEFAULT, you can insert a value, and Db2 provides a default value if you do not supply one.

**Example:** Suppose that tables T1 and T2 have two columns: an integer column and a ROWID column. For the following statement to run successfully, ROWIDCOL2 must be defined as GENERATED BY DEFAULT.

```
INSERT INTO T2 (INTCOL2,ROWIDCOL2)
SELECT * FROM T1;
```

If ROWIDCOL2 is defined as GENERATED ALWAYS, you cannot insert the ROWID column data from T1 into T2, but you can insert the integer column data. To insert only the integer data, use one of the following methods:

- Specify only the integer column in your INSERT statement, as in the following statement:
INSERT INTO T2 (INTCOL2)
SELECT INTCOL1 FROM T1;

• Specify the OVERRIDING USER VALUE clause in your INSERT statement to tell Db2 to ignore any values that you supply for system-generated columns, as in the following statement:

INSERT INTO T2 (INTCOL2,ROWIDCOL2) OVERRIDING USER VALUE
SELECT * FROM T1;

Related concepts
Direct row access (PRIMARY_ACCESTYPE='D') (Db2 Performance)
ROWID data type (Introduction to Db2 for z/OS)

Related tasks
Specifying direct row access by using row IDs
For some applications, you can use the value of a ROWID column to navigate directly to a row.

Rules for inserting data into an identity column
An identity column contains a unique numeric value for each row in the table. Whether you can insert data into an identity column and how that data gets inserted depends on how the column is defined.

An identity column is a numeric column, defined in a CREATE TABLE or ALTER TABLE statement, that has ascending or descending values. For an identity column to be as useful as possible, its values should also be unique. The column has a SMALLINT, INTEGER, or DECIMAL(p,0) data type and is defined with the AS IDENTITY clause. The AS IDENTITY clause specifies that the column is an identity column. For information about using identity columns to uniquely identify rows, see “Identity columns” on page 124

Before you insert data into an identity column, you must know how the column is defined. Identity columns are defined with the GENERATED ALWAYS or GENERATED BY DEFAULT clause. GENERATED ALWAYS means that Db2 generates a value for the column, and you cannot insert data into that column. If the column is defined as GENERATED BY DEFAULT, you can insert a value, and Db2 provides a default value if you do not supply one.

Example: Suppose that tables T1 and T2 have two columns: a character column and an integer column that is defined as an identity column. For the following statement to run successfully, IDENTCOL2 must be defined as GENERATED BY DEFAULT.

INSERT INTO T2 (CHARCOL2,IDENTCOL2)
SELECT * FROM T1;

If IDENTCOL2 is defined as GENERATED ALWAYS, you cannot insert the identity column data from T1 into T2, but you can insert the character column data. To insert only the character data, use one of the following methods:

• Specify only the character column in your INSERT statement, as in the following statement:

INSERT INTO T2 (CHARCOL2)
SELECT CHARCOL1 FROM T1;

• Specify the OVERRIDING USER VALUE clause in your INSERT statement to tell Db2 to ignore any values that you supply for system-generated columns, as in the following statement:

INSERT INTO T2 (CHARCOL2,IDENTCOL2) OVERRIDING USER VALUE
SELECT * FROM T1;

Restrictions when assigning values to columns with distinct types
Certain conditions are required when you assign a column value to another column or when you assign a constant to a column of a distinct type. If the conditions are not met, you cannot assign the value.

When assigning a column value to another column or a constant to a column of a distinct type, the type of the value that is to be assigned must match the column type, or you must be able to cast one type to the other. Otherwise, you cannot assign the value.
If you need to assign a value of one distinct type to a column of another distinct type, a function must exist that converts the value from one type to another. Because Db2 provides cast functions only between distinct types and their source types, you must write the function to convert from one distinct type to another.

**Assigning column values to columns with different distinct types**

Suppose tables JAPAN_SALES and JAPAN_SALES_03 are defined like this:

```sql
CREATE TABLE JAPAN_SALES
  (PRODUCT_ITEM  INTEGER,
   MONTH         INTEGER CHECK (MONTH BETWEEN 1 AND 12),
   YEAR          INTEGER CHECK (YEAR > 1990),
   TOTAL         JAPANESE_YEN);

CREATE TABLE JAPAN_SALES_03
  (PRODUCT_ITEM  INTEGER,
   TOTAL         US_DOLLAR);
```

You need to insert values from the TOTAL column in JAPAN_SALES into the TOTAL column of JAPAN_SALES_03. Because INSERT statements follow assignment rules, Db2 does not let you insert the values directly from one column to the other because the columns are of different distinct types. Suppose that a user-defined function called US_DOLLAR has been written that accepts values of type JAPANESE_YEN as input and returns values of type US_DOLLAR. You can then use this function to insert values into the JAPAN_SALES_03 table:

```sql
INSERT INTO JAPAN_SALES_03
  SELECT PRODUCT_ITEM, US_DOLLAR(TOTAL)
  FROM JAPAN_SALES
  WHERE YEAR = 2003;
```

**Assigning column values with distinct types to host variables**

The rules for assigning distinct types to host variables or host variables to columns of distinct types differ from the rules for constants and columns.

You can assign a column value of a distinct type to a host variable if you can assign a column value of the distinct type’s source type to the host variable. In the following example, you can assign SIZECOL1 and SIZECOL2, which has distinct type SIZE, to host variables of type double and short because the source type of SIZE, which is INTEGER, can be assigned to host variables of type double or short.

```sql
EXEC SQL BEGIN DECLARE SECTION;
  double hv1;
  short  hv2;
EXEC SQL END DECLARE SECTION;
CREATE DISTINCT TYPE SIZE AS INTEGER;
CREATE TABLE TABLE1 (SIZECOL1 SIZE, SIZECOL2 SIZE);
⋮
SELECT SIZECOL1, SIZECOL2
  INTO :hv1, :hv2
  FROM TABLE1;
```

**Assigning host variable values to columns with distinct types**

When you assign a value in a host variable to a column with a distinct type, the type of the host variable must be able to cast to the distinct type.

In this example, values of host variable hv2 can be assigned to columns SIZECOL1 and SIZECOL2, because C data type short is equivalent to Db2 data type SMALLINT, and SMALLINT is promotable to data type INTEGER. However, values of hv1 cannot be assigned to SIZECOL1 and SIZECOL2, because C data type double, which is equivalent to Db2 data type DOUBLE, is not promotable to data type INTEGER.

```sql
EXEC SQL BEGIN DECLARE SECTION;
  double hv1;
  short  hv2;
EXEC SQL END DECLARE SECTION;
```
CREATE DISTINCT TYPE SIZE AS INTEGER;
CREATE TABLE TABLE1 (SIZECOL1 SIZE, SIZECOL2 SIZE);

INSERT INTO TABLE1
VALUES (:hv1,:hv1);     /* Invalid statement */
INSERT INTO TABLE1
VALUES (:hv2,:hv2);     /* Valid statement */

Related concepts
Promotion of data types (Db2 SQL)

Inserting data and updating data in a single operation
You can update existing data and insert new data in a single operation. This operation is useful when you want to update a table with a set of rows, some of which are changes to existing rows and some of which are new rows.

About this task
You can update existing data and insert new data in a single operation by using the MERGE statement.

For example, an application might request a set of rows from a database, enable a user to modify the data through a GUI, and then store the modified data in the database. Some of this modified data is updates to existing rows, and some of this data is new rows. You can do these update and insert operations in one step.

Procedure
Issue a MERGE statement.

To update existing data and inserting new data, specify a MERGE statement with the WHEN MATCHED and WHEN NOT MATCHED clauses. These clauses specify how Db2 handles matched and unmatched data. If Db2 finds a matching row, that row is updated. If Db2 does not find a matching row, a new row is inserted.

Example
Suppose that you need to update the inventory at a car dealership. You need to add new car models to the inventory and update information about car models that are already in the inventory.

You could make these changes with the following series of statements:

```
UPDATE INVENTORY
SET QUANTITY = QUANTITY + :hv_delta
WHERE MODEL = :hv_model;

--begin pseudo code
if sqlcode >= 0
then do
  GD
  if rc = 0  then INSERT..
end
-- end pseudo code
GET DIAGNOSTICS :rc = ROW_COUNT;

IF rc = 0 THEN
  INSERT INTO INVENTORY VALUES (:hv_model, :hv_delta);
END IF;
```

The MERGE statement simplifies the update and the insert into a single statement:

```
MERGE INTO INVENTORY
USING ( VALUES (:hv_model, :hv_delta) ) AS SOURCE(MODEL, DELTA)
ON INVENTORY.MODEL = SOURCE.MODEL
WHEN MATCHED THEN UPDATE SET QUANTITY = QUANTITY + SOURCE.DELTA
WHEN NOT MATCHED THEN INSERT VALUES (SOURCE.MODEL, SOURCE.DELTA)
NOT ATOMIC CONTINUE ON SQLEXCEPTION;
```
Selecting values while merging data

When you update existing data and insert new data in a single merge operation, you can select values from those rows at the same time.

Procedure

Specifying the MERGE statement in the FROM clause of the SELECT statement.

When you merge one or more rows into a table, you can retrieve:

• The value of an automatically generated column such as a ROWID or identity column
• Any default values for columns
• All values for a merged row, without specifying individual column names
• Calculated values based on the changes to merged rows

Specify the FINAL TABLE clause with SELECT FROM MERGE statements. The FINAL TABLE consists of the rows of the table or view after the merge occurs.

Example

Suppose that you need to input data into the STOCK table, which contains company stock symbols and stock prices from your stock portfolio. Some of your input data refers to companies that are already in the STOCK table; some of the data refers to companies that you are adding to your stock portfolio. If the stock symbol exists in the SYMBOL column of the STOCK table, you need to update the PRICE column. If the company stock symbol is not yet in the STOCK table, you need to insert a new row with the stock symbol and the stock price. Furthermore, you need to add a new value DELTA to your output to show the change in stock price.

Suppose that the STOCK table contains the data that is shown in Table 62 on page 342.

<table>
<thead>
<tr>
<th>SYMBOL</th>
<th>PRICE</th>
</tr>
</thead>
<tbody>
<tr>
<td>XCOM</td>
<td>95.00</td>
</tr>
<tr>
<td>YCOM</td>
<td>24.50</td>
</tr>
</tbody>
</table>

Now, suppose that :hv_symbol and :hv_price are host-variable arrays that contain updated data that corresponds to the data that is shown in Table 62 on page 342. Table 63 on page 342 shows the host variable data for stock activity.

<table>
<thead>
<tr>
<th>hv_symbol</th>
<th>hv_price</th>
</tr>
</thead>
<tbody>
<tr>
<td>XCOM</td>
<td>97.00</td>
</tr>
<tr>
<td>NEWC</td>
<td>30.00</td>
</tr>
<tr>
<td>XCOM</td>
<td>107.00</td>
</tr>
</tbody>
</table>

NEWC is new to the STOCK table, so its symbol and price need to be inserted into the STOCK table. The rows for XCOM in Table 63 on page 342 represent changed stock prices, so these values need to be updated in the STOCK table. Also, the output needs to show the change in stock prices as a DELTA value.

The following SELECT FROM MERGE statement updates the price of XCOM, inserts the symbol and price for NEWC, and returns an output that includes a DELTA value for the change in stock price.
The INCLUDE clause specifies that an additional column, DELTA, can be returned in the output without adding a column to the STOCK table. The UPDATE portion of the MERGE statement sets the DELTA value to the differential of the previous stock price with the value set for the update operation. The INSERT portion of the MERGE statement sets the DELTA value to the same value as the PRICE column.

After the SELECT FROM MERGE statement is processed, the STOCK table contains the data that is shown in Table 64 on page 343.

<table>
<thead>
<tr>
<th>SYMBOL</th>
<th>PRICE</th>
</tr>
</thead>
<tbody>
<tr>
<td>XCOM</td>
<td>107.00</td>
</tr>
<tr>
<td>YCOM</td>
<td>24.50</td>
</tr>
<tr>
<td>NEWC</td>
<td>30.00</td>
</tr>
</tbody>
</table>

The following output of the SELECT FROM MERGE statement includes both updates to XCOM and a DELTA value for each output row.

<table>
<thead>
<tr>
<th>SYMBOL</th>
<th>PRICE</th>
<th>DELTA</th>
</tr>
</thead>
<tbody>
<tr>
<td>XCOM</td>
<td>97.00</td>
<td>2.00</td>
</tr>
<tr>
<td>NEWC</td>
<td>30.00</td>
<td>30.00</td>
</tr>
<tr>
<td>XCOM</td>
<td>107.00</td>
<td>10.00</td>
</tr>
</tbody>
</table>

Selecting values while inserting data

When you insert rows into a table, you can also select values from the inserted rows at the same time.

About this task

When you insert one or more new rows into a table, you can also retrieve rows, including the following values:

- The value of an automatically generated column such as a ROWID or identity column
- Any default values for columns
- All values for an inserted row, without specifying individual column names
- All values that are inserted by a multiple-row INSERT operation
- Values that are changed by a BEFORE INSERT trigger

Procedure

Specify the INSERT statement in the FROM clause of the SELECT statement.

The rows that are inserted into the target table produce a result table whose columns can be referenced in the SELECT list of the query. The columns of the result table are affected by the columns, constraints, and triggers that are defined for the target table:

- The result table includes Db2-generated values for identity columns, ROWID columns, or row change timestamp columns.
- Before Db2 generates the result table, it enforces any constraints that affect the insert operation (that is, check constraints, unique index constraints, and referential integrity constraints).
• The result table includes any changes that result from a BEFORE trigger that is activated by the insert operation. An AFTER trigger does not affect the values in the result table.

Examples
In addition to examples that use the Db2 sample tables, the examples in this topic use an EMPSAMP table that has the following definition:

```
CREATE TABLE EMPSAMP
(EMPNO     INTEGER GENERATED ALWAYS AS Identity,
 NAME      CHAR(30),
 SALARY    DECIMAL(10,2),
 DEPTNO    SMALLINT,
 LEVEL     CHAR(30),
 HIRETYPE  VARCHAR(30) NOT NULL WITH DEFAULT 'New Hire',
 HIREDATE  DATE NOT NULL WITH DEFAULT);
```

Example 1: Retrieving generated column values
Assume that you need to insert a row for a new employee into the EMPSAMP table. To find out the values for the generated EMPNO, HIRETYPE, and HIREDATE columns, use the following SELECT FROM INSERT statement:

```
SELECT EMPNO, HIRETYPE, HIREDATE
FROM FINAL TABLE (INSERT INTO EMPSAMP (NAME, SALARY, DEPTNO, LEVEL)
VALUES('Mary Smith', 35000.00, 11, 'Associate'));
```

The SELECT statement returns the Db2-generated identity value for the EMPNO column, the default value 'New Hire' for the HIRETYPE column, and the value of the CURRENT DATE special register for the HIREDATE column.

Recommendation: Use the SELECT FROM INSERT statement to insert a row into a parent table and retrieve the value of a primary key that was generated by Db2 (a ROWID or identity column). In another INSERT statement, specify this generated value as a value for a foreign key in a dependent table.

Example 2: Retrieving values updated by triggers
Suppose that a BEFORE INSERT trigger is created on table EMPSAMP to give all new employees at the Associate level a $5000 increase in salary. The trigger has the following definition:

```
CREATE TRIGGER NEW_ASSOC
NO CASCADE BEFORE INSERT ON EMPSAMP
REFERENCING NEW AS NEWSALARY
FOR EACH ROW MODE DB2SQL
WHEN (NEWSALARY.LEVEL = 'ASSOCIATE')
BEGIN ATOMIC
  SET NEWSALARY.SALARY = NEWSALARY.SALARY + 5000.00;
END;
```

The INSERT statement in the FROM clause of the following SELECT statement inserts a new employee into the EMPSAMP table:

```
SELECT NAME, SALARY
FROM FINAL TABLE (INSERT INTO EMPSAMP (NAME, SALARY, DEPTNO, LEVEL)
VALUES('Mary Smith', 35000.00, 11, 'Associate'));
```

The SELECT statement returns a salary of 40000.00 for Mary Smith instead of the initial salary of 35000.00 that was explicitly specified in the INSERT statement.

Selecting values when you insert a single row:
When you insert a new row into a table, you can retrieve any column in the result table of the SELECT FROM INSERT statement. When you embed this statement in an application, you retrieve the row into host variables by using the SELECT ... INTO form of the statement.
Example 4: Retrieving all values for a row inserted into a structure.
You can retrieve all the values for a row that is inserted into a structure. For example, in the following statement: `empstruct` is a host variable structure that is declared with variables for each of the columns in the EMPSAMP table.

```sql
EXEC SQL SELECT * INTO :empstruct
FROM FINAL TABLE (INSERT INTO EMPSAMP (NAME, SALARY, DEPTNO, LEVEL)
VALUES('Mary Smith', 35000.00, 11, 'Associate'));
```

Example 4: Selecting values when inserting data into a view
If the INSERT statement references a view that is defined with a search condition, that view must be defined with the WITH CASCADED CHECK OPTION option. When you insert data into the view, the result table of the SELECT FROM INSERT statement includes only rows that satisfy the view definition.

Because view `V1` is defined with the WITH CASCADED CHECK OPTION option, you can reference `V1` in the INSERT statement:

```sql
CREATE VIEW V1 AS
SELECT C1, I1 FROM T1 WHERE I1 > 10
WITH CASCADED CHECK OPTION;

SELECT C1 FROM
FINAL TABLE (INSERT INTO V1 (I1) VALUES(12));
```

The value 12 satisfies the search condition of the view definition, and the result table consists of the value for `C1` in the inserted row.

If you use a value that does not satisfy the search condition of the view definition, the insert operation fails, and Db2 returns an error.

Example 5: Selecting ROWID values when inserting multiple rows
In an application program, to retrieve values from the insertion of multiple rows, declare a cursor so that the INSERT statement is in the FROM clause of the SELECT statement of the cursor.

To see the values of the ROWID columns that are inserted into the employee photo and resume table, you can declare the following cursor:

```sql
EXEC SQL DECLARE CS1 CURSOR FOR
SELECT EMP_ROWID
FROM FINAL TABLE (INSERT INTO DSN8C10.EMP_PHOTO_RESUME (EMPNO)
SELECT EMPNO FROM DSN8C10.EMP);
```

Example 6: Using the FETCH FIRST clause
To see only the first five rows that are inserted into the employee photo and resume table, use the FETCH FIRST clause:

```sql
EXEC SQL DECLARE CS2 CURSOR FOR
SELECT EMP_ROWID
FROM FINAL TABLE (INSERT INTO DSN8C10.EMP_PHOTO_RESUME (EMPNO)
SELECT EMPNO FROM DSN8C10.EMP)
FETCH FIRST 5 ROWS ONLY;
```

Example 7: Using the INPUT SEQUENCE clause
To retrieve rows in the order in which they are inserted, use the INPUT SEQUENCE clause:

```sql
EXEC SQL DECLARE CS3 CURSOR FOR
SELECT EMP_ROWID
FROM FINAL TABLE (INSERT INTO DSN8C10.EMP_PHOTO_RESUME (EMPNO)
VALUES(:hva_empno)
FOR 5 ROWS)
ORDER BY INPUT SEQUENCE;
```

The INPUT SEQUENCE clause can be specified only if an INSERT statement is in the FROM clause of the SELECT statement. In this example, the rows are inserted from an array of employee numbers.
Example 8: Inserting rows with multiple encoding CCSIDs
Suppose that you want to populate an ASCII table with values from an EBCDIC table and then see selected values from the ASCII table. You can use the following cursor to select the EBCDIC columns, populate the ASCII table, and then retrieve the ASCII values:

```sql
EXEC SQL DECLARE CS4 CURSOR FOR
    SELECT C1, C2
    FROM FINAL TABLE (INSERT INTO ASCII_TABLE
    SELECT * FROM EBCDIC_TABLE);
```

Example 9: Selecting additional columns when inserting data
You can use the INCLUDE clause to introduce a new column to the result table but not add a column to the target table.

Suppose that you need to insert department number data into the project table. Suppose also, that you want to retrieve the department number and the corresponding manager number for each department. Because MGRNO is not a column in the project table, you can use the INCLUDE clause to include the manager number in your result but not in the insert operation. The following SELECT FROM INSERT statement performs the insert operation and retrieves the data.

```sql
DECLARE CS1 CURSOR FOR
    SELECT manager_num, projname FROM FINAL TABLE
    (INSERT INTO PROJ (DEPTNO) INCLUDE(manager_num CHAR(6))
    SELECT DEPTNO, MGRNO FROM DEPT);
```

Example 10: Result table of the cursor when you insert multiple rows
In an application program, when you insert multiple rows into a table, you declare a cursor so that the INSERT statement is in the FROM clause of the SELECT statement of the cursor. The result table of the cursor is determined during OPEN cursor processing. The result table may or may not be affected by other processes in your application.

When you declare a scrollable cursor, the cursor must be declared with the INSENSITIVE keyword if an INSERT statement is in the FROM clause of the cursor specification. The result table is generated during OPEN cursor processing and does not reflect any future changes. You cannot declare the cursor with the SENSITIVE DYNAMIC or SENSITIVE STATIC keywords.

When you declare a non-scrollable cursor, any searched updates or deletes do not affect the result table of the cursor. The rows of the result table are determined during OPEN cursor processing.

For example, assume that your application declares a cursor, opens the cursor, performs a fetch, updates the table, and then fetches additional rows:

```sql
EXEC SQL DECLARE CS1 CURSOR FOR
    SELECT SALARY
    FROM FINAL TABLE (INSERT INTO EMPSAMP (NAME, SALARY, LEVEL)
    SELECT NAME, INCOME, BAND FROM OLD_EMPLOYEE);
EXEC SQL OPEN CS1;
EXEC SQL FETCH CS1 INTO :hv_salary;
/* print fetch result */
...
EXEC SQL UPDATE EMPSAMP SET SALARY = SALARY + 500;
while (SQLCODE == 0) {
    EXEC SQL FETCH CS1 INTO :hv_salary;
    /* print fetch result */
    ...
}
```

The fetches that occur after the updates return the rows that were generated when the cursor was opened. If you use a simple SELECT (with no INSERT statement in the FROM clause), the fetches might return the updated values, depending on the access path that Db2 uses.

Example 11: Effect of WITH HOLD
When you declare a cursor with the WITH HOLD option and open the cursor, all of the rows are inserted into the target table. The WITH HOLD option has no effect on the SELECT FROM INSERT statement of the cursor definition. After your application performs a commit, you can continue to retrieve all of the inserted rows.
Assume that the employee table in the Db2 sample application has five rows. Your application declares a WITH HOLD cursor, opens the cursor, fetches two rows, performs a commit, and then fetches the third row successfully:

```
EXEC SQL DECLARE CS2 CURSOR WITH HOLD FOR
   SELECT EMP_ROWID
   FROM FINAL TABLE (INSERT INTO DSN8C10.EMP_PHOTO_RESUME (EMPNO)
   SELECT EMPNO FROM DSN8C10.EMP);
EXEC SQL OPEN CS2;                                /* Inserts 5 rows */
EXEC SQL FETCH CS2 INTO :hv_rowid;   /* Retrieves ROWID for 1st row */
EXEC SQL FETCH CS2 INTO :hv_rowid;   /* Retrieves ROWID for 2nd row */
EXEC SQL COMMIT;                                  /* Commits 5 rows */
EXEC SQL FETCH CS2 INTO :hv_rowid;   /* Retrieves ROWID for 3rd row */
```

**Example 12: Effect of SAVEPOINT and ROLLBACK**

A savepoint is a point in time within a unit of recovery to which relational database changes can be rolled back. You can set a savepoint with the SAVEPOINT statement.

When you set a savepoint prior to opening the cursor and then roll back to that savepoint, all of the insertions are undone.

Assume that your application declares a cursor, sets a savepoint, opens the cursor, sets another savepoint, rolls back to the second savepoint, and then rolls back to the first savepoint:

```
EXEC SQL DECLARE CS3 CURSOR FOR
   SELECT EMP_ROWID
   FROM FINAL TABLE (INSERT INTO DSN8C10.EMP_PHOTO_RESUME (EMPNO)
   SELECT EMPNO FROM DSN8C10.EMP);
EXEC SQL SAVEPOINT A ON ROLLBACK RETAIN CURSORS;       /* Sets 1st savepoint */
EXEC SQL OPEN CS3;
EXEC SQL SAVEPOINT B ON ROLLBACK RETAIN CURSORS;       /* Sets 2nd savepoint */
...
EXEC SQL ROLLBACK TO SAVEPOINT B;  /* Rows still in DSN8C10.EMP_PHOTO_RESUME */
...
EXEC SQL ROLLBACK TO SAVEPOINT A;            /* All inserted rows are undone */
```

**Example 13: Errors during SELECT INTO processing**

In an application program, when you insert one or more rows into a table by using the SELECT FROM INSERT statement, the result table of the insert operation may or may not be affected, depending on where the error occurred in the application processing.

If the insert processing or the select processing fails during a SELECT INTO statement, no rows are inserted into the target table, and no rows are returned from the result table of the insert operation. For example, assume that the employee table of the Db2 sample application has one row, and that the SALARY column has a value of 9999000.00.

```
EXEC SQL SELECT EMPNO INTO :hv_empno
   FROM FINAL TABLE (INSERT INTO EMPSAMP (NAME, SALARY)
   SELECT FIRSTNAME || MIDINIT || LASTNAME,
   SALARY + 10000.00
   FROM DSN8C10.EMP)
```

The addition of 10000.00 causes a decimal overflow to occur, and no rows are inserted into the EMPSAMP table.

**Example 14: Errors during OPEN cursor processing**

If the insertion of any row fails during the OPEN cursor processing, all previously successful insertions are undone. The result table of the insert is empty.

**Example 15: Errors during FETCH processing**

If the FETCH statement fails while retrieving rows from the result table of the insert operation, a negative SQLCODE is returned to the application, but the result table still contains the original number of rows that was determined during the OPEN cursor processing. At this point, you can undo all of the inserts.
Assume that the result table contains 100 rows and the 90th row that is being fetched from the cursor returns a negative SQLCODE:

```sql
EXEC SQL DECLARE CS1 CURSOR FOR
SELECT EMPNO
FROM FINAL TABLE (INSERT INTO EMPSAMP (NAME, SALARY)
SELECT FIRSTNAME || MIDINIT || LASTNAME, SALARY + 10000.00
FROM DSN8C10.EMP);
EXEC SQL OPEN CS1; /* Inserts 100 rows */
while (SQLCODE == 0)
EXEC SQL FETCH CS1 INTO :hv_empno;
if (SQLCODE == -904) /* If SQLCODE is -904, undo all inserts */
EXEC SQL ROLLBACK;
else
EXEC SQL COMMIT; /* Else, commit inserts */
```

Related concepts

Held and non-held cursors

A held cursor does not close after a commit operation. A cursor that is not held closes after a commit operation. You specify whether you want a cursor to be held or not held by including or omitting the WITH HOLD clause when you declare the cursor.

Using host variables in SQL statements

Use scalar host variables in embedded SQL statements to represent a single value. Host variables are useful for storing retrieved data or for passing values that are to be assigned or used for comparisons.

Identity columns

An identity column contains a unique numeric value for each row in the table. Db2 can automatically generate sequential numeric values for this column as rows are inserted into the table. Thus, identity columns are ideal for primary key values, such as employee numbers or product numbers.

Types of cursors

You can declare row-positioned or rowset-positioned cursors in a number of ways. These cursors can be scrollable or not scrollable, held or not held, or returnable or not returnable.

Related tasks

Inserting multiple rows of data from host-variable arrays

Use host-variable arrays in your INSERT statement when you do not know at least some of the values to insert until the program runs.

Retrieving a set of rows by using a cursor

In an application program, you can retrieve a set of rows from a table or a result table that is returned by a stored procedure. You can retrieve one or more rows at a time.

Undoing selected changes within a unit of work by using savepoints

Savepoints enable you to undo selected changes within a unit of work. Your application can set any number of savepoints and then specify a specific savepoint to indicate which changes to undo within the unit of work.

Related reference

Command line processor BIND command

Use the command line processor BIND command to bind DBRMs that are in z/OS UNIX HFS files to packages.

Preserving the order of a derived table

When you specify SELECT FROM INSERT, SELECT FROM UPDATE, SELECT FROM DELETE, or SELECT FROM MERGE, you can preserve the order of the derived table. This action ensures that the result rows of a fullselect follow the same order as the result table of a subquery within the fullselect.

Procedure

To preserve the order of the derived table specify the ORDER OF clause with the ORDER BY clause.

These two clauses ensure that the result rows of a fullselect follow the same order as the result table of a subquery within the fullselect.
You can use the ORDER OF clause in any query that uses an ORDER BY clause, but the ORDER OF clause is most useful with queries that contain a set operator, such as UNION.

**Examples**

**Example**
The following example retrieves the following rows:

- Rows of table T1 in no specified order
- Rows of table T2 in the order of the first column in table T2

The example query then performs a UNION ALL operation on the results of the two subqueries. The ORDER BY ORDER OF UTABLE clause in the query specifies that the fullselect result rows are to be returned in the same order as the result rows of the UNION ALL statement.

```sql
SELECT * FROM
(SELECT * FROM T1
UNION ALL
(SELECT * FROM T2 ORDER BY 1)
) AS UTABLE
ORDER BY ORDER OF UTABLE;
```

**Example**
The following example joins data from table T1 to the result table of a nested table expression. The nested table expression is ordered by the second column in table T2. The ORDER BY ORDER OF TEMP clause in the query specifies that the fullselect result rows are to be returned in the same order as the nested table expression.

```sql
SELECT T1.C1, T1.C2, TEMP.Cy, TEMP.Cx
FROM T1, (SELECT T2.C1, T2.C2 FROM T2 ORDER BY 2) as TEMP(Cx, Cy)
WHERE Cy = T1.C1
ORDER BY ORDER OF TEMP;
```

Alternatively, you can produce the same result by explicitly stating the ORDER BY column TEMP.Cy in the fullselect instead of using the ORDER OF syntax.

```sql
SELECT T1.C1, T1.C2, TEMP.Cy, TEMP.Cx
FROM T1, (SELECT T2.C1, T2.C2 FROM T2 ORDER BY 2) as TEMP(Cx, Cy)
WHERE Cy = T1.C1
ORDER BY TEMP.Cy;
```

**Related reference**
- fullselect (Db2 SQL)
- order-by-clause (Db2 SQL)

### Adding data to the end of a table

In a relational database, the rows of a table are not ordered, and thus, the table has no "end." However, depending on your goal, you can perform several actions to simulate adding data to the end of a table.

**About this task**

**Question:** How can I add data to the end of a table?

**Answer:** Though the question is often asked, it has no meaning in a relational database. The rows of a base table are not ordered; hence, the table does not have an "end".

However, depending on your goal, you can perform one of the following actions to simulate adding data to the end of a table:

- If your goal is to get a result table that is ordered according to when the rows were inserted, define a unique index on a TIMESTAMP column in the table definition. Then, when you retrieve data from the table, use an ORDER BY clause that names that column. The newest insert appears last.
• If your goal is for Db2 to insert rows in the next available free space, without preserving clustering order, specify the APPEND YES option when you create or alter the table. Specifying this option might reduce the time it takes to insert rows, because Db2 does not spend time searching for free space.

### Storing data that does not have a tabular format

Db2 provides several options for you to store large volumes of data that is not defined as a set of columns in a table.

#### About this task

**Question:** How can I store a large volume of data that is not defined as a set of columns in a table?

**Answer:** You can store the data in a table in a binary string, a LOB, or an XML column.

### Updating table data

You can change a column value to another value or remove the column value altogether.

#### Procedure

To change the data in a table, use the UPDATE statement. For example, suppose that an employee relocates. To update several items of the employee's data in the YEMP work table to reflect the move, you can execute the following statement:

```sql
UPDATE YEMP
SET JOB = 'MANAGER ',
PHONENO = '5678'
WHERE EMPNO = '000400';
```

You can also use the UPDATE statement to remove a value from a column (without removing the row) by changing the column value to null.

You cannot update rows in a created temporary table, but you can update rows in a declared temporary table.

The SET clause names the columns that you want to update and provides the values that you want to assign to those columns. You can replace a column value in the SET clause with any of the following items:

- A null value
  - The column to which you assign the null value must not be defined as NOT NULL.
- An expression, which can be any of the following items:
  - A column
  - A constant
  - A scalar fullselect
  - A host variable
  - A special register
- A default value
  - If you specify DEFAULT, Db2 determines the value based on how the corresponding column is defined in the table.

In addition, you can replace one or more column values in the SET clause with the column values in a row that is returned by a fullselect.

Next, identify the rows to update:

- To update a single row, use a WHERE clause that locates one, and only one, row.
- To update several rows, use a WHERE clause that locates only the rows that you want to update.
If you omit the WHERE clause, Db2 updates **every row** in the table or view with the values that you supply.

If Db2 finds an error while executing your UPDATE statement (for example, an update value that is too large for the column), it stops updating and returns an error. No rows in the table change. Rows that were already changed, if any, are restored to their previous values. If the UPDATE statement is successful, SQLERRD(3) is set to the number of rows that are updated.

**Example**

The following statement supplies a missing middle initial and changes the job for employee 000200.

```
UPDATE YEMP
SET MIDINIT = 'H', JOB = 'FIELDREP'
WHERE EMPNO = '000200';
```

The following statement gives everyone in department D11 a raise of 400.00. The statement can update several rows.

```
UPDATE YEMP
SET SALARY = SALARY + 400.00
WHERE WORKDEPT = 'D11';
```

The following statement sets the salary for employee 000190 to the average salary and sets the bonus to the minimum bonus for all employees.

```
UPDATE YEMP
SET (SALARY, BONUS) =
(SELECT AVG(SALARY), MIN(BONUS)
FROM EMP)
WHERE EMPNO = '000190';
```

**Related reference**

UPDATE (Db2 SQL)

**Selecting values while updating data**

When you update rows in a table, you can select the updated values from those rows at the same time.

**Procedure**

Specify the UPDATE statement in the FROM clause of the SELECT statement.

When you update one or more rows in a table, you can retrieve:

- The value of an automatically generated column such as a ROWID or identity column
- Any default values for columns
- All values for an updated row, without specifying individual column names

In most cases, you want to use the FINAL TABLE clause with SELECT FROM UPDATE statements. The FINAL TABLE consists of the rows of the table or view after the update occurs.

**Examples**

**Example: SELECT FROM FINAL TABLE**

Suppose that all clerks for a company are receiving 5 percent raises. You can use the following SELECT FROM UPDATE statement to increase the salary of each designer by 5 percent and to retrieve the total increase in salary for the company.

```
SELECT SUM(SALARY) INTO :salary FROM FINAL TABLE
  (UPDATE EMP SET SALARY = SALARY * 1.05
   WHERE JOB = 'DESIGNER');
```
Example: retrieving data row-by-row from updated data

To retrieve row-by-row output of updated data, use a cursor with a SELECT FROM UPDATE statement. For example, suppose that all designers for a company are receiving a 30 percent increase in their bonus. You can use the following SELECT FROM UPDATE statement to increase the bonus of each clerk by 30 percent and to retrieve the bonus for each clerk.

```sql
DECLARE CS1 CURSOR FOR
    SELECT LASTNAME, BONUS FROM FINAL TABLE
    (UPDATE EMP SET BONUS = BONUS * 1.3
     WHERE JOB = 'CLERK');
FETCH CS1 INTO :lastname, :bonus;
```

Example: INCLUDE a new column in the result table but not the target table

You can use the INCLUDE clause to introduce a new column to the result table but not add the column to the target table. For example, suppose that sales representatives received a 20 percent increase in their commission. You need to update the commission (COMM) of sales representatives (SALESREP) in the EMP table and that you need to retrieve the old commission and the new commission for each sales representative. You can use the following SELECT FROM UPDATE statement to perform the update and to retrieve the required data.

```sql
DECLARE CS2 CURSOR FOR
    SELECT LASTNAME, COMM, old_comm FROM FINAL TABLE
    (UPDATE EMP INCLUDE(old_comm DECIMAL (7,2))
     SET COMM = COMM * 1.2, old_comm = COMM
     WHERE JOB = 'SALESREP');
```

Related reference

- `table-reference (Db2 SQL)`
- `UPDATE (Db2 SQL)`

Updating thousands of rows

When you update large volumes of data, consider certain recommended actions to increase concurrency.

About this task

**Question:** Are there any special techniques for updating large volumes of data?

**Answer:** Yes. When updating large volumes of data using a cursor, you can minimize the amount of time that you hold locks on the data by declaring the cursor with the HOLD option and by issuing commits frequently.

Deleting data from tables

You can delete data from a table by deleting one or more rows from the table, by deleting all rows from the table, or by dropping columns from the table.

**Procedure**

To delete one or more rows in a table:

- Use the DELETE statement with a WHERE clause to specify a search condition.

  The DELETE statement removes zero or more rows of a table, depending on how many rows satisfy the search condition that you specify in the WHERE clause.

  You can use DELETE with a WHERE clause to remove only selected rows from a declared temporary table, but not from a created temporary table.

  The following DELETE statement deletes each row in the YEMP table that has an employee number '000060'.

```sql
DELETE FROM YEMP
WHERE EMPNO = '000060';
```
When this statement executes, Db2 deletes any row from the **YEMP** table that meets the search condition.

If Db2 finds an error while executing your DELETE statement, it stops deleting data and returns error codes in the SQLCODE and SQLSTATE variables or related fields in the SQLCA. The data in the table does not change.

If the DELETE is successful, SQLERRD(3) in the SQLCA contains the number of deleted rows. This number includes only the number of deleted rows in the table that is specified in the DELETE statement. Rows that are deleted (in other tables) according to the CASCADE rule are not included in SQLERRD(3).

To delete every row in a table:

- Use the DELETE statement without specifying a WHERE clause.

  With segmented table spaces, deleting all rows of a table is very fast.

  The following DELETE statement deletes every row in the **YDEPT** table:

  ```sql
  DELETE FROM YDEPT;
  ```

  If the statement executes, the table continues to exist (that is, you can insert rows into it), but it is empty. All existing views and authorizations on the table remain intact when using DELETE.

- Use the TRUNCATE statement.

  The TRUNCATE statement can provide the following advantages over a DELETE statement:

  - The TRUNCATE statement can ignore delete triggers
  - The TRUNCATE statement can perform an immediate commit
  - The TRUNCATE statement can keep storage allocated for the table

  The TRUNCATE statement does not, however, reset the count for an automatically generated value for an identity column on the table. If 14872 was the next identity column value to be generated before a TRUNCATE statement, 14872 would be the next value generated after the TRUNCATE statement.

  Suppose that you need to empty the data from an old inventory table, regardless of any existing delete triggers, and you need to make the space that is allocated for the table available for other uses. Use the following TRUNCATE statement.

  ```sql
  TRUNCATE INVENTORY_TABLE
  IGNORE DELETE TRIGGERS
  DROP STORAGE;
  ```

  Suppose that you need to empty the data from an old inventory table permanently, regardless of any existing delete triggers, and you need to preserve the space that is allocated for the table. You need the emptied data to be completely unavailable, so that a ROLLBACK statement cannot return the data. Use the following TRUNCATE statement.

  ```sql
  TRUNCATE INVENTORY_TABLE
  REUSE STORAGE
  IGNORE DELETE TRIGGERS
  IMMEDIATE;
  ```

- Use the DROP TABLE statement.

  DROP TABLE drops the specified table and all related views and authorizations, which can invalidate plans and packages.

To drop columns from a table:

- Use the ALTER TABLE statement with the DROP COLUMN clause.

  Because dropping a column from a table is a pending change to the definition of the table, the table space is placed in advisory REORG-pending status (AREOR). When the pending change is applied (by running the REORG utility with the SHRLEVEL CHANGE or REFERENCE options), the column is dropped
from the table, and any dependent packages and statements in the dynamic statement cache are invalidated.

**Related concepts**
SQL communication area (SQLCA) (Db2 SQL)

**Related tasks**
Dropping tables
When you drop a table, you delete the data and the table definition. You also delete all synonyms, views, indexes, referential constraints, and check constraints that are associated with that table.

**Related reference**
DELETE (Db2 SQL)
DROP (Db2 SQL)
TRUNCATE (Db2 SQL)
ALTER TABLE (Db2 SQL)

**Selecting values while deleting data**
When you delete rows from a table, you can select the values from those rows at the same time.

**Procedure**
Specify the DELETE statement in the FROM clause of the SELECT statement.
When you delete one or more rows in a table, you can retrieve:
- Any default values for columns
- All values for a deleted row, without specifying individual column names
- Calculated values based on deleted rows

**Example**

**Example: FROM OLD TABLE clause**
When you use a SELECT FROM DELETE statement, you must use the FROM OLD TABLE clause to retrieve deleted values. The OLD TABLE consists of the rows of the table or view before the delete occurs. For example, suppose that a company is eliminating all operator positions and that the company wants to know how much salary money it will save by eliminating these positions. You can use the following SELECT FROM DELETE statement to delete operators from the EMP table and to retrieve the sum of operator salaries.

```sql
SELECT SUM(SALARY) INTO :salary FROM OLD TABLE
(DELETE FROM EMP
WHERE JOB = 'OPERATOR');
```

**Example: retrieving row-by-row output of deleted data**
To retrieve row-by-row output of deleted data, use a cursor with a SELECT FROM DELETE statement. For example, suppose that a company is eliminating all analyst positions and that the company wants to know how many years of experience each analyst had with the company. You can use the following SELECT FROM DELETE statement to delete analysts from the EMP table and to retrieve the experience of each analyst.

```sql
DECLARE CS1 CURSOR FOR
SELECT YEAR(CURRENT DATE - HIREDATE) FROM OLD TABLE
(DELETE FROM EMP
WHERE JOB = 'ANALYST');
FETCH CS1 INTO :years_of_service;
```

**Example: retrieving calculated data based on deleted table**
If you need to retrieve calculated data, based on the data that you delete but not add that column to the target table. For example, suppose that you need to delete managers from the EMP table and that you need to retrieve the salary and the years of employment for each manager. You can use
the following SELECT FROM DELETE statement to perform the delete operation and to retrieve the required data.

```
DECLARE CS2 CURSOR FOR
SELECT LASTNAME, SALARY, years_employed FROM OLD TABLE
  (DELETE FROM EMP INCLUDE(years_employed INTEGER)
   SET years_employed = YEAR(CURRENT DATE - HIREDATE)
   WHERE JOB = 'MANAGER');
```

**Related reference**
- table-reference (Db2 SQL)
- DELETE (Db2 SQL)

## Accessing data in tables from application programs

Your program can use a number of different techniques to read data from any Db2 tables for which you have read access. The simplest technique is to use basic SQL SELECT statements. However, you should choose the technique that works best for your situation and performs well.

**About this task**

**Tip:** You can use tools such as IBM Data Server Manager and related Db2 Tools for z/OS solutions to simplify this task.

- IBM Data Server Manager (Introduction to Db2 for z/OS)
- Db2 Tools for z/OS

**Related concepts**
- Investigating SQL performance by using EXPLAIN (Db2 Performance)
- Interpreting data access by using EXPLAIN (Db2 Performance)

**Related tasks**
- Writing efficient SQL queries (Db2 Performance)

### Determining which tables you have access to

You can ask Db2 to list the tables that a specific authorization ID has access to.

**About this task**

The contents of the Db2 catalog tables can be a useful reference tool when you begin to develop an SQL statement or an application program.

The catalog table, SYSIBM.SYSTABAUTH, lists table privileges that are granted to authorization IDs. To display the tables that you have authority to access (by privileges granted either to your authorization ID or to PUBLIC), you can execute an SQL statement similar to the one shown in the following example. To do this, you must have the SELECT privilege on SYSIBM.SYSTABAUTH.

**Procedure**

Issue a SELECT statement similar to the following example. To do this, you must have the SELECT privilege on SYSIBM.SYSTABAUTH.

```
SELECT DISTINCT TCREATOR, TTNAME
FROM SYSIBM.SYSTABAUTH
WHERE GRANTEE IN (USER, 'PUBLIC', 'PUBLIC*') AND GRANTEETYPE = ' ';
```

In this query, the predicate GRANTEETYPE = ' ' selects authorization IDs.

**Exception:** If your Db2 subsystem uses an exit routine for access control authorization, you cannot rely on catalog queries to tell you the tables that you can access. When such an exit routine is installed, both RACF and Db2 control table access.

Chapter 3. Db2 SQL programming 355
Displaying information about the columns for a given table

You can ask Db2 to list the columns in a particular table and certain information about those columns.

About this task

The catalog table, SYSIBM.SYSCOLUMNS, describes every column of every table.

Procedure

Query the SYSIBM.SYSCOLUMNS catalog table.

Examples

Example

Suppose that you want to display information about table DSN8C10.DEPT. If you have the SELECT privilege on SYSIBM.SYSCOLUMNS, you can use the following statement:

```
SELECT NAME, COLTYPE, SCALE, LENGTH
FROM SYSIBM.SYSCOLUMNS
WHERE TBNAME = 'DEPT'
AND TBCREATOR = 'DSN8C10';
```

Example

If you display column information about a table that includes LOB or ROWID columns, the LENGTH field for those columns contains the number of bytes that those column occupy in the base table. The LENGTH field does not contain the length of the LOB or ROWID data.

To determine the maximum length of data for a LOB or ROWID column, include the LENGTH2 column in your query:

```
SELECT NAME, COLTYPE, LENGTH, LENGTH2
FROM SYSIBM.SYSCOLUMNS
WHERE TBNAME = 'EMP_PHOTO_RESUME'
AND TBCREATOR = 'DSN8C10';
```

Related reference

SYSCOLUMNS catalog table (Db2 SQL)

Retrieving data by using the SELECT statement

The simplest way to retrieve data is to use the SQL statement SELECT to specify a result table. You can specify the columns and rows that you want to retrieve.

Before you begin

Consider developing your own SQL statements similar to the examples in this section, and then run them dynamically using SPUFI. For a tutorial see Lesson 1.1: Querying data interactively (Introduction to Db2 for z/OS).

You can also use the command line processor, or Db2 Query Management Facility (QMF).

Procedure

Issue a SELECT statement.
Examples

Example 1: Selecting all columns with SELECT *
You do not need to know the column names to select Db2 data. Use an asterisk (*) in the SELECT clause to indicate that you want to retrieve all columns of each selected row of the named table. Implicitly hidden columns, such as ROWID columns and XML document ID columns, are not included in the result of the SELECT * statement. To view the values of these columns, you must specify the column name.

The following SQL statement selects all columns from the department table:

```
SELECT *
FROM DSN8C10.DEPT;
```

The result table looks similar to the following output:

```
DEPTNO  DEPTNAME                         MGRNO      ADMRDEPT  LOCATION
======  ==============================   ======     ========  ========
A00     SPIFFY COMPUTER SERVICES DIV.    000010     A00       --------
B01     PLANNING                         000020     A00       --------
C01     INFORMATION CENTER               000030     A00       --------
D01     DEVELOPMENT CENTER               --------  A00       --------
D11     MANUFACTURING CENTER             000060     D01       --------
D21     ADMINISTRATION SYSTEMS           000070     D01       --------
E00     SUPPORT SERVICES                 000050     A00       --------
E11     OPERATIONS                       000090     E01       --------
E21     SOFTWARE SUPPORT                 000100     E01       --------
F22     BRANCH OFFICE F2                 --------  E01       --------
G22     BRANCH OFFICE G2                 --------  E01       --------
H22     BRANCH OFFICE H2                 --------  E01       --------
J22     BRANCH OFFICE J2                 --------  E01       --------
```

Because the example does not specify a WHERE clause, the statement retrieves data from all rows. The dashes for MGRNO and LOCATION in the result table indicate null values.

SELECT * is recommended mostly for use with dynamic SQL and view definitions. You can use SELECT * in static SQL, but doing so is not recommended because of possible host variable compatibility and performance implications. Suppose that you add a column to the table to which SELECT * refers. If you have not defined a receiving host variable for that column, an error might occur, or the data from the added column might not be retrieved.

If you list the column names in a static SELECT statement instead of using an asterisk, you can avoid problems that might occur with SELECT *. You can also see the relationship between the receiving host variables and the columns in the result table.

Example 2: selecting specific columns with SELECT column-name
Select the column or columns you want to retrieve by naming each column. With a single SELECT statement, you can select data from one column or as many as 750 columns. All columns appear in the order you specify, not in their order in the table.

For example, the following SQL statement retrieves only the MGRNO and DEPTNO columns from the department table:

```
SELECT MGRNO, DEPTNO
FROM DSN8C10.DEPT;
```

The result table looks similar to the following output:

```
MGRNO     DEPTNO
-------    -------
000010     A00
000020     B01
000030     C01
--------   D01
000050     E01
000060     D11
000070     D21
000090     E11
```

Chapter 3. Db2 SQL programming 357
**Example 3: Selecting data from implicitly hidden columns**

To SELECT data from implicitly hidden columns, such as ROWID and XML document ID, look up the column names in SYSIBM.SYSCOLUMNS and specify these names in the SELECT list. For example, suppose that you create and populate the following table:

```
CREATE TABLE MEMBERS (MEMBERID INTEGER, 
                        BIO          XML, 
                        REPORT      XML, 
                        RECOMMENDATIONS     XML);
```

Db2 generates one additional implicitly hidden XML document ID column. To retrieve data in all columns, including the generated XML document ID column, first look up the name of the generated column in SYSIBM.SYSCOLUMNS. Suppose the name is DB2_GENERATED_DOCID_FOR_XML. Then, specify the following statement:

```
SELECT DB2_GENERATED_DOCID_FOR_XML, MEMBERID, BIO, 
       REPORT, RECOMMENDATIONS FROM MEMBERS
```

**Related concepts**

- **Host variables**
  Use host variables to pass a single data item between Db2 and your application.

- **Remote servers and distributed data**
  *Distributed data* is data that resides on a database management system (DBMS) other than your local system. Your local DBMS is the one on which you bind your application plan. All other DBMSs are remote.

- **Predicates (Db2 SQL)**

**Related reference**

- select-statement (Db2 SQL)

**Specifying search conditions with a WHERE clause**

You can use a WHERE clause to select the rows that meet certain conditions. A WHERE clause specifies a search condition. A *search condition* consists of one or more predicates. A *predicate* specifies a test that you want Db2 to apply to each table row.

**About this task**

A WHERE clause specifies a search condition. A *search condition* consists of one or more predicates. A *predicate* specifies a test that you want Db2 to apply to each table row.

**Procedure**

Specify a WHERE clause with one or more predicates.

Db2 evaluates a predicate for each row as true, false, or unknown. Results are unknown only if an operand is null.

If a search condition contains a column of a distinct type, the value to which that column is compared must be of the same distinct type, or you must cast the value to the distinct type.

The following table lists the type of comparison, the comparison operators, and an example of each type of comparison that you can use in a predicate in a WHERE clause.
### Table 65. Comparison operators used in conditions

<table>
<thead>
<tr>
<th>Type of comparison</th>
<th>Comparison operator</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Equal to</td>
<td>=</td>
<td>DEPTNO = 'X01'</td>
</tr>
<tr>
<td>Not equal to</td>
<td>&lt;&gt;</td>
<td>DEPTNO &lt;&gt; 'X01'</td>
</tr>
<tr>
<td>Less than</td>
<td>&lt;</td>
<td>AVG(SALARY) &lt; 30000</td>
</tr>
<tr>
<td>Less than or equal to</td>
<td>&lt;=</td>
<td>AGE &lt;= 25</td>
</tr>
<tr>
<td>Not less than</td>
<td>=&gt;</td>
<td>AGE &gt;= 21</td>
</tr>
<tr>
<td>Greater than</td>
<td>&gt;</td>
<td>SALARY &gt; 2000</td>
</tr>
<tr>
<td>Greater than or equal to</td>
<td>=&gt;</td>
<td>SALARY &gt;= 5000</td>
</tr>
<tr>
<td>Not greater than</td>
<td>&lt;=</td>
<td>SALARY &lt;= 5000</td>
</tr>
<tr>
<td>Equal to null</td>
<td>IS NULL</td>
<td>PHONENO IS NULL</td>
</tr>
<tr>
<td>Not equal to another value or one value is equal to null</td>
<td>IS DISTINCT FROM</td>
<td>PHONENO IS DISTINCT FROM :PHONEHV</td>
</tr>
<tr>
<td>Similar to another value</td>
<td>LIKE</td>
<td>NAME LIKE ' or STATUS LIKE 'N_'</td>
</tr>
<tr>
<td>At least one of two conditions</td>
<td>OR</td>
<td>HIREDATE &lt; '1965-01-01' OR SALARY &lt; 16000</td>
</tr>
<tr>
<td>Both of two conditions</td>
<td>AND</td>
<td>HIREDATE &lt; '1965-01-01' AND SALARY &lt; 16000</td>
</tr>
<tr>
<td>Between two values</td>
<td>BETWEEN</td>
<td>SALARY BETWEEN 20000 AND 40000</td>
</tr>
<tr>
<td>Equals a value in a set</td>
<td>IN (X, Y, Z)</td>
<td>DEPTNO IN ('B01', 'C01', 'D01')</td>
</tr>
</tbody>
</table>

**Note:** SALARY BETWEEN 20000 AND 40000 is equivalent to SALARY >= 20000 AND SALARY <= 40000.

You can also search for rows that do not satisfy one of the preceding conditions by using the NOT keyword before the specified condition.

You can search for rows that do not satisfy the IS DISTINCT FROM predicate by using either of the following predicates:

- `value 1 IS NOT DISTINCT FROM value 2`
- `NOT(value 1 IS DISTINCT FROM value 2)`

Both of these forms of the predicate create an expression for which one value is equal to another value or both values are equal to null.

**Related concepts**

Subqueries
Subqueries are used when you need to narrow your search condition based on information in an interim table, you can use a subquery. For example, you might want to find all employee numbers in one table that also exist for a given project in a second table.

**Predicates (Db2 SQL)**

<table>
<thead>
<tr>
<th>Related tasks</th>
<th>Coding SQL statements to avoid unnecessary processing (Db2 Performance)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Related reference</td>
<td>where-clause (Db2 SQL)</td>
</tr>
</tbody>
</table>
Handling null values

A null value indicates the absence of a column value in a row. A null value is an unknown value; it is not the same as zero or all blanks.

About this task

Null values can be used as a condition in the WHERE and HAVING clauses. For example, a WHERE clause can specify a column that, for some rows, contains a null value. A basic comparison predicate using a column that contains null values does not select a row that has a null value for the column. This is because a null value is not less than, equal to, or greater than the value specified in the condition. The IS NULL predicate is used to check for null values.

Examples

Example 1: Selecting rows that contain null in a column
To select the values for all rows that contain a null value for the manager number, you can issue the following statement:

```
SELECT DEPTNO, DEPTNAME, ADMRDEPT
FROM DSN8C10.DEPT
WHERE MGRNO IS NULL
```

The following table shows the result.

<table>
<thead>
<tr>
<th>DEPTNO</th>
<th>DEPTNAME</th>
<th>ADMRDEPT</th>
</tr>
</thead>
<tbody>
<tr>
<td>D01</td>
<td>DEVELOPMENT CENTER</td>
<td>A00</td>
</tr>
<tr>
<td>F22</td>
<td>BRANCH OFFICE F2</td>
<td>E01</td>
</tr>
<tr>
<td>G22</td>
<td>BRANCH OFFICE G2</td>
<td>E01</td>
</tr>
<tr>
<td>H22</td>
<td>BRANCH OFFICE H2</td>
<td>E01</td>
</tr>
<tr>
<td>I22</td>
<td>BRANCH OFFICE I2</td>
<td>E01</td>
</tr>
<tr>
<td>J22</td>
<td>BRANCH OFFICE J2</td>
<td>E01</td>
</tr>
</tbody>
</table>

Example 2: Selecting rows that do not contain a null value
To get the rows that do not have a null value for the manager number, you can change the WHERE clause in the previous example like this:

```
WHERE MGRNO IS NOT NULL
```

Example 3: Comparing values that contain the NULL value
Another predicate that is useful for comparing values that can contain the NULL value is the DISTINCT predicate. Comparing two columns using a normal equal comparison (COL1 = COL2) will be true if both columns contain an equal non-null value. If both columns are null, the result will be false because null is never equal to any other value, not even another null value. Using the DISTINCT predicate, null values are considered equal. So COL1 is NOT DISTINCT from COL2 will be true if both columns contain an equal non-null value and also when both columns are the null value.

For example, suppose that you want to select information from two tables that contain null values. The first table T1 has a column C1 with the following values.

<table>
<thead>
<tr>
<th>C1</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>null</td>
</tr>
</tbody>
</table>
The second table has column C2 with the following values.

<table>
<thead>
<tr>
<th>C2</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
</tr>
<tr>
<td>null</td>
</tr>
</tbody>
</table>

Assume that you issue the following SELECT statement:

```sql
SELECT * FROM T1, T2 WHERE C1 IS DISTINCT FROM C2
```

The result follows.

<table>
<thead>
<tr>
<th>C1</th>
<th>C2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>-</td>
</tr>
<tr>
<td>2</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>2</td>
</tr>
</tbody>
</table>

**Related concepts**

- Nulls (Db2 SQL)
- Null values in table columns (Introduction to Db2 for z/OS)

**How to check for null values**

Before you retrieve a column value, you might first want to determine if the column value is null.

Applications frequently need to check two values to see if they are equal or not equal. You can use a basic predicate to do an equal or not equal comparison. An equal comparison or a not equal comparison can return true, false, or unknown. The normal rule in SQL, except for the DISTINCT predicate, is that one null value is never equal to another null value. If either or both operands of a basic predicate are the null value, the result is unknown.

Depending on your application, you might want to include or exclude rows that have a NULL value in a column. You can use the NULL predicate to do that.

MY_EMP is a table that has a row with the last name and phone number for each of the employees in a company. In this company, no employees share a phone number, but some employees might not have a phone number. The LASTNAME column contains the last name of each employee. The PHONENO column contains the phone number for each employee. If an employee does not have a phone, the PHONENO column value is NULL. The table might look like this:

<table>
<thead>
<tr>
<th>LASTNAME</th>
<th>PHONENO</th>
</tr>
</thead>
<tbody>
<tr>
<td>HAAS</td>
<td>-------</td>
</tr>
<tr>
<td>THOMPSON</td>
<td>3476</td>
</tr>
<tr>
<td>KWAN</td>
<td>4738</td>
</tr>
<tr>
<td>GEYER</td>
<td>6789</td>
</tr>
<tr>
<td>STERN</td>
<td>6423</td>
</tr>
</tbody>
</table>
Suppose that you want to know the last name of the employee who has no phone number. Using a query like this one does not work, because if the PHONENO column value is NULL, the WHERE clause compares a NULL column value to a null host variable value. The result of that comparison is unknown.

```sql
MOVE -1 TO PHONENO-IND.
EXEC SQL
SELECT LASTNAME
  INTO :LASTNAME-HV
FROM MY_EMP
WHERE PHONENO = :PHONENO-HV :PHONENO-IND
END-EXEC.
```

To find the employee with a NULL value for the phone number, you need to use a NULL predicate:

```sql
EXEC SQL
SELECT LASTNAME
  INTO :LASTNAME-HV
FROM MY_EMP
WHERE PHONENO IS NULL
END-EXEC.
```

The SELECT statement returns a LASTNAME value of 'HAAS'.

Now suppose that you want to select the last name of an employee whose phone number matches a certain value or whose phone number is NULL. To do that, you need to code two search conditions: one to handle the case where the phone number is not NULL, and another to handle the case where the phone number is NULL. The SELECT statement might look like this:

```sql
EXEC SQL
SELECT LASTNAME
  INTO :LASTNAME-HV
FROM MY_EMP
WHERE (PHONENO IS NOT NULL AND :PHONENO-HV :PHONENO-IND IS NOT NULL
  AND PHONENO = :PHONENO-HV ) -- Search condition for non-NULL
    -- phone number
OR
  (PHONENO IS NULL AND :PHONENO-HV :PHONENO-IND IS NULL)
    -- Search condition for NULL
    -- phone number
END-EXEC.
```

If you set :PHONENO-HV to '3476' and :PHONENO-IND to 0, the SELECT statement returns 'THOMPSON' because the search condition for a non-NULL phone number is used. If you set :PHONENO-HV to any value, and set :PHONENO-IND to -1, the SELECT statement returns 'HAAS' because the search condition for a NULL phone number is used.

**Related tasks**
Declaring host variables and indicator variables
You can use host variables and indicator variables in SQL statements in your program to pass data between Db2 and your application.

**Related reference**
DISTINCT predicate (Db2 SQL)
NULL predicate (Db2 SQL)
Selecting derived columns

In a SELECT statement, you can select columns that are not actual columns in a table. Instead, you can specify "columns" that are derived from a constant, an expression, or a function.

Example: SELECT with an expression

This SQL statement generates a result table in which the second column is a derived column that is generated by adding the values of the SALARY, BONUS, and COMM columns.

```sql
SELECT EMPNO, (SALARY + BONUS + COMM) FROM DSN8C10.EMP;
```

Derived columns in a result table, such as (SALARY + BONUS + COMM), do not have names. You can use the AS clause to give a name to an unnamed column of the result table. For information about using the AS clause, see "Naming result columns" on page 365.

What to do next

To order the rows in a result table by the values in a derived column, specify a name for the column by using the AS clause, and specify that name in the ORDER BY clause. For information about using the ORDER BY clause, see “Ordering the result table rows” on page 366.

Selecting XML data

You can select all XML data that is stored in a particular column or only a subset of data from an XML column.

Procedure

- You can select all XML data that is stored in a particular column by specifying SELECT column name or SELECT *, just as you would for columns of any other data type.
- Alternatively, you can select only a subset of data from an XML column by using an XPath expression in a SELECT statement. XPath expressions identify specific nodes in an XML document.

To select a subset of data in an XML column, specify the XMLQUERY function in your SELECT statement with the following parameters:

- An XPath expression that is embedded in a character string constant. Specify an XPath expression that identifies which XML data to return.
- Any additional values to pass to the XPath expression, including the XML column name. Specify these values after the PASSING keyword.

Example

Suppose that you store purchase orders as XML documents in the POrder column in the PurchaseOrders table. You need to find in each purchase order the items whose product name is equal to a name in the Product table. You can use the following statement to find these values:

```sql
SELECT XMLQUERY('//item[productName = $n]'
PASSING PO.POrder, 
P.name AS "n")
FROM PurchaseOrders PO, Product P;
```

This statement returns the item elements in the POrder column that satisfy the criteria in the XPath expression.

Related concepts
Overview of XQuery (Db2 Programming for XML)

Related reference
XMLQUERY (Db2 SQL)
Formatting the result table

An SQL statement returns data in a table called a result table. You can specify certain attributes of the result table, such as the column names, how the rows are ordered, and whether the rows are numbered.

Result tables

The data that is retrieved by an SQL statement is always in the form of a table, which is called a result table. Like the tables from which you retrieve the data, a result table has rows and columns. A program fetches this data one row at a time.

Example result table: Assume that you issue the following SELECT statement, which retrieves the last name, first name, and phone number of employees in department D11 from the sample employee table:

```
SELECT LASTNAME, FIRSTNAME, PHONENO
FROM DSN8C10.EMP
WHERE WORKDEPT = 'D11'
ORDER BY LASTNAME;
```

The result table looks similar to the following output:

<table>
<thead>
<tr>
<th>LASTNAME</th>
<th>FIRSTNAME</th>
<th>PHONENO</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADAMSON</td>
<td>BRUCE</td>
<td>4510</td>
</tr>
<tr>
<td>BROWN</td>
<td>DAVID</td>
<td>4561</td>
</tr>
<tr>
<td>JOHN</td>
<td>REBA</td>
<td>0672</td>
</tr>
<tr>
<td>JONES</td>
<td>WILLIAM</td>
<td>0942</td>
</tr>
<tr>
<td>LUTZ</td>
<td>JENNIFER</td>
<td>0672</td>
</tr>
<tr>
<td>PIANKA</td>
<td>ELIZABETH</td>
<td>3782</td>
</tr>
<tr>
<td>SCOUTTEN</td>
<td>MARILYN</td>
<td>1682</td>
</tr>
<tr>
<td>STERN</td>
<td>IRVING</td>
<td>6432</td>
</tr>
<tr>
<td>WALKER</td>
<td>JAMES</td>
<td>2986</td>
</tr>
<tr>
<td>YAMAMOTO</td>
<td>KIYOSHI</td>
<td>2890</td>
</tr>
<tr>
<td>YOSHIMURA</td>
<td>MASATOSHI</td>
<td>2890</td>
</tr>
</tbody>
</table>

Excluding duplicate rows from the result table of a query

You can ask Db2 to exclude multiple identical rows from a query result table. For example, a query might return multiple rows for each employee when one row per employee is sufficient for your program.

Procedure

Specify the DISTINCT keyword in the query.

The DISTINCT keyword excludes duplicate rows from your query result table, so that each row contains unique data.

Restriction: You cannot use the DISTINCT keyword with LOB columns or XML columns.

Example

The following SELECT statement lists unique department numbers for administrative departments:

```
SELECT DISTINCT ADMRDEPT
FROM DSN8C10.DEPT;
```

The result table looks similar to the following output:

<table>
<thead>
<tr>
<th>ADMRDEPT</th>
</tr>
</thead>
<tbody>
<tr>
<td>A00</td>
</tr>
<tr>
<td>D01</td>
</tr>
<tr>
<td>E01</td>
</tr>
</tbody>
</table>

Related tasks

Coding SQL statements to avoid unnecessary processing (Db2 Performance)

Related reference

select-clause (Db2 SQL)
**Naming result columns**

You can provide your own names for the result table columns for a SELECT statement. This capability is particularly useful for a column that is derived from an expression or a function.

**Procedure**

Use the AS clause to name result columns in a SELECT statement.

**Examples**

The following examples show different ways to use the AS clause.

**Example: SELECT with AS CLAUSE**

The following example of the SELECT statement gives the expression SALARY+BONUS+COMM the name TOTAL_SAL.

```sql
SELECT SALARY+BONUS+COMM AS TOTAL_SAL
FROM DSN8C10.EMP
ORDER BY TOTAL_SAL;
```

**Example: CREATE VIEW with AS clause**

You can specify result column names in the select-clause of a CREATE VIEW statement. You do not need to supply the column list of CREATE VIEW, because the AS keyword names the derived column. The columns in the view EMP_SAL are EMPNO and TOTAL_SAL.

```sql
CREATE VIEW EMP_SAL AS
SELECT EMPNO, SALARY+BONUS+COMM AS TOTAL_SAL
FROM DSN8C10.EMP;
```

**Example: set operator with AS clause**

You can use the AS clause with set operators, such as UNION. In this example, the AS clause is used to give the same name to corresponding columns of tables in a UNION. The third result column from the union of the two tables has the name TOTAL_VALUE, even though it contains data that is derived from columns with different names:

```sql
SELECT 'On hand' AS STATUS, PARTNO, QOH * COST AS TOTAL_VALUE
FROM PART_ON_HAND
UNION ALL
SELECT 'Ordered' AS STATUS, PARTNO, QORDER * COST AS TOTAL_VALUE
FROM ORDER_PART
ORDER BY PARTNO, TOTAL_VALUE;
```

The column STATUS and the derived column TOTAL_VALUE have the same name in the first and second result tables. They are combined in the union of the two result tables, which is similar to the following partial output:

```
<table>
<thead>
<tr>
<th>STATUS</th>
<th>PARTNO</th>
<th>TOTAL_VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>On hand</td>
<td>00557</td>
<td>345.60</td>
</tr>
<tr>
<td>Ordered</td>
<td>00557</td>
<td>150.50</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
```

**Example: GROUP BY derived column**

You can use the AS clause in a FROM clause to assign a name to a derived column that you want to refer to in a GROUP BY clause. This SQL statement names HIREYEAR in the nested table expression, which lets you use the name of that result column in the GROUP BY clause:

```sql
SELECT HIREYEAR, AVG(SALARY)
FROM (SELECT YEAR(HIREDATE) AS HIREYEAR, SALARY
FROM DSN8C10.EMP) AS NEWEMP
GROUP BY HIREYEAR;
```

You cannot use GROUP BY with a name that is defined with an AS clause for the derived column YEAR(HIREDATE) in the outer SELECT, because that name does not exist when the GROUP BY runs.
However, you can use GROUP BY with a name that is defined with an AS clause in the nested table expression, because the nested table expression runs before the GROUP BY that references the name.

Related tasks
Combining result tables from multiple SELECT statements
When you combine the results of multiple SELECT statements, you can choose what to include in the result table. You can include all rows, only rows that are in the result table of both SELECT statements, or only rows that are unique to the result table of the first SELECT statement.

Defining a view
A view is a named specification of a result table. Use views to control which users have access to certain data or to simplify writing SQL statements.

Summarizing group values
You can group rows in the result table by the values of one or more columns or by the results of an expression. You can then apply aggregate functions to each group.

Related reference
select-clause (Db2 SQL)

Ordering the result table rows
If you want to guarantee that the rows in your result table are ordered in a particular way, you must specify the order in the SELECT statement. Otherwise, Db2 can return the rows in any order.

About this task
Using ORDER BY is the only way to guarantee that your rows are ordered as you want them.

Procedure
To retrieve rows in a specific order, use the ORDER BY clause

Examples
Example: Specifying the sort key in the ORDER BY clause
The order of the selected rows depends on the sort keys that you identify in the ORDER BY clause. A sort key can be a column name, an integer that represents the number of a column in the result table, or an expression. Db2 orders the rows by the first sort key, followed by the second sort key, and so on.

You can list the rows in ascending or descending order. Null values appear last in an ascending sort and first in a descending sort.

Db2 sorts strings in the collating sequence associated with the encoding scheme of the table. Db2 sorts numbers algebraically and sorts datetime values chronologically.

Restriction: You cannot use the ORDER BY clause with LOB or XML columns.

Example: ORDER BY clause with a column name as the sort key
Retrieve the employee numbers, last names, and hire dates of employees in department A00 in ascending order of hire dates:

```
SELECT EMPNO, LASTNAME, HIREDATE
FROM DSN8C10.EMP
WHERE WORKDEPT = 'A00'
ORDER BY HIREDATE ASC;
```

The result table looks similar to the following output:

<table>
<thead>
<tr>
<th>EMPNO</th>
<th>LASTNAME</th>
<th>HIREDATE</th>
</tr>
</thead>
<tbody>
<tr>
<td>000110</td>
<td>LUCCHESI</td>
<td>1958-05-16</td>
</tr>
<tr>
<td>000120</td>
<td>O'CONNELL</td>
<td>1963-12-05</td>
</tr>
<tr>
<td>000010</td>
<td>HAAS</td>
<td>1965-01-01</td>
</tr>
</tbody>
</table>
Example: ORDER BY clause with an expression as the sort key

The following subselect retrieves the employee numbers, salaries, commissions, and total compensation (salary plus commission) for employees with a total compensation greater than 40000. Order the results by total compensation:

```
SELECT EMPNO, SALARY, COMM, SALARY+COMM AS "TOTAL COMP"
FROM DSN8C10.EMP
WHERE SALARY+COMM > 40000
ORDER BY SALARY+COMM;
```

The intermediate result table looks similar to the following output:

<table>
<thead>
<tr>
<th>EMPNO</th>
<th>SALARY</th>
<th>COMM</th>
<th>TOTAL COMP</th>
</tr>
</thead>
<tbody>
<tr>
<td>000030</td>
<td>38250.00</td>
<td>3060.00</td>
<td>41310.00</td>
</tr>
<tr>
<td>000050</td>
<td>40175.00</td>
<td>3214.00</td>
<td>43389.00</td>
</tr>
<tr>
<td>000020</td>
<td>41250.00</td>
<td>3300.00</td>
<td>44550.00</td>
</tr>
<tr>
<td>000110</td>
<td>46500.00</td>
<td>3720.00</td>
<td>50220.00</td>
</tr>
<tr>
<td>200010</td>
<td>46500.00</td>
<td>4220.00</td>
<td>50720.00</td>
</tr>
<tr>
<td>000010</td>
<td>52750.00</td>
<td>4220.00</td>
<td>56970.00</td>
</tr>
</tbody>
</table>

Referencing derived columns in the ORDER BY clause

If you use the AS clause to name an unnamed column in a SELECT statement, you can use that name in the ORDER BY clause. The following SQL statement orders the selected information by total salary:

```
SELECT EMPNO, (SALARY + BONUS + COMM) AS TOTAL_SAL
FROM DSN8C10.EMP
ORDER BY TOTAL_SAL;
```

Related reference

fullselect (Db2 SQL)

Numbering the rows in a result table

Db2 does not number the rows in the result table for a query unless you explicitly request that the rows be numbered.

About this task

To number the rows in a result table, include the ROW_NUMBER specification in your query. If you want to ensure that the rows are in a particular order, include an ORDER BY clause after the OVER keyword. Otherwise, the rows are numbered in an arbitrary order.

Example

Suppose that you want a list of employees and salaries from department D11 in the sample EMP table. You can return a numbered list that is ordered by last name by submitting the following query:

```
SELECT ROW_NUMBER() OVER (ORDER BY LASTNAME) AS NUMBER,
WORKDEPT, LASTNAME, SALARY
FROM DSN8910.EMP
WHERE WORKDEPT='D11'
```

This query returns the following result:

<table>
<thead>
<tr>
<th>NUMBER</th>
<th>WORKDEPT</th>
<th>LASTNAME</th>
<th>SALARY</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>D11</td>
<td>ADAMSON</td>
<td>25280.00</td>
</tr>
<tr>
<td>2</td>
<td>D11</td>
<td>BROWN</td>
<td>27740.00</td>
</tr>
<tr>
<td>3</td>
<td>D11</td>
<td>JOHN</td>
<td>29840.00</td>
</tr>
<tr>
<td>4</td>
<td>D11</td>
<td>JONES</td>
<td>18270.00</td>
</tr>
<tr>
<td>5</td>
<td>D11</td>
<td>LUTZ</td>
<td>29840.00</td>
</tr>
<tr>
<td>6</td>
<td>D11</td>
<td>PIANKA</td>
<td>22250.00</td>
</tr>
<tr>
<td>7</td>
<td>D11</td>
<td>SCOUTTEN</td>
<td>21340.00</td>
</tr>
<tr>
<td>8</td>
<td>D11</td>
<td>STERN</td>
<td>32250.00</td>
</tr>
</tbody>
</table>
Ranking the rows

You can request that Db2 calculate the ordinal rank of each row in the result set based on a particular column. For example, you can rank finishing times for a marathon to determine the first, second, and third place finishers.

Procedure

To rank rows, use one of the following ranking specifications in an SQL statement:

- **Use RANK to return a rank number for each row value.**
  
  Use this specification if you want rank numbers to be skipped when duplicate row values exist.
  
  For example, suppose the top five finishers in a marathon have the following times:
  - 2:31:57
  - 2:34:52
  - 2:34:52
  - 2:37:26
  - 2:38:01

  When you use the RANK specification, Db2 returns the following rank numbers:

<table>
<thead>
<tr>
<th>Value</th>
<th>Rank number</th>
</tr>
</thead>
<tbody>
<tr>
<td>2:31:57</td>
<td>1</td>
</tr>
<tr>
<td>2:34:52</td>
<td>2</td>
</tr>
<tr>
<td>2:34:52</td>
<td>2</td>
</tr>
<tr>
<td>2:37:26</td>
<td>4</td>
</tr>
<tr>
<td>2:38:01</td>
<td>5</td>
</tr>
</tbody>
</table>

- **Use DENSE_RANK to return a rank number for each row value.**
  
  Use this specification if you do not want rank numbers to be skipped when duplicate row values exist.
  
  For example, when you specify DENSE_RANK with the same times that are listed in the description of RANK, Db2 returns the following rank numbers:

<table>
<thead>
<tr>
<th>Value</th>
<th>Rank number</th>
</tr>
</thead>
<tbody>
<tr>
<td>2:31:57</td>
<td>1</td>
</tr>
<tr>
<td>2:34:52</td>
<td>2</td>
</tr>
<tr>
<td>2:34:52</td>
<td>2</td>
</tr>
<tr>
<td>2:37:26</td>
<td>3</td>
</tr>
<tr>
<td>2:38:01</td>
<td>4</td>
</tr>
</tbody>
</table>
Examples
Suppose that you had the following values in the DATA column of table T1:

<table>
<thead>
<tr>
<th>DATA</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
</tr>
<tr>
<td>35</td>
</tr>
<tr>
<td>23</td>
</tr>
<tr>
<td>8</td>
</tr>
<tr>
<td>8</td>
</tr>
<tr>
<td>6</td>
</tr>
</tbody>
</table>

Example: RANK
Suppose that you use the following RANK specification:

```sql
SELECT DATA,
    RANK() OVER (ORDER BY DATA DESC) AS RANK_DATA
FROM T1
ORDER BY RANK_DATA;
```

Db2 returns the following ranked data:

<table>
<thead>
<tr>
<th>DATA</th>
<th>RANK_DATA</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>1</td>
</tr>
<tr>
<td>35</td>
<td>2</td>
</tr>
<tr>
<td>23</td>
<td>3</td>
</tr>
<tr>
<td>8</td>
<td>4</td>
</tr>
<tr>
<td>8</td>
<td>4</td>
</tr>
<tr>
<td>6</td>
<td>5</td>
</tr>
</tbody>
</table>

Example: DENSE RANK
Suppose that you use the following DENSE_RANK specification on the same data:

```sql
SELECT DATA,
    DENSE_RANK() OVER (ORDER BY DATA DESC) AS RANK_DATA
FROM T1
ORDER BY RANK_DATA;
```

Db2 returns the following ranked data:

<table>
<thead>
<tr>
<th>DATA</th>
<th>RANK_DATA</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>1</td>
</tr>
<tr>
<td>35</td>
<td>2</td>
</tr>
<tr>
<td>23</td>
<td>3</td>
</tr>
<tr>
<td>8</td>
<td>4</td>
</tr>
<tr>
<td>8</td>
<td>4</td>
</tr>
<tr>
<td>6</td>
<td>5</td>
</tr>
</tbody>
</table>

In the example with the RANK specification, two equal values are both ranked as 4. The next rank number is 6. Number 5 is skipped.

In the example with the DENSE_RANK option, those two equal values are also ranked as 4. However, the next rank number is 5. With DENSE_RANK, no gaps exist in the sequential rank numbering.

Related reference
OLAP specification (Db2 SQL)
Accessing part of a result set based on data position

Data-dependent or numeric-based pagination can be used to retrieve a subset of data from a result set based on the position of the data.

About this task

To retrieve a subset of data from a result set based on the position of the data in the result set, you can use either data-dependent pagination or numeric-based pagination.

Procedure

• For **data dependent pagination**: Use *row-value expressions* with the `<`, `<=`, `>`, or `>=` comparison operators in a SELECT statement to retrieve only part of a result set.
  When used with a basic predicate, row-value expressions enable an application to access only part of a Db2 result table based on a logical key value.
  The following SELECT statement returns information from the table where the value of the `LASTNAME` column is greater than or equal to ‘SMITH’ and the value of the `FIRSTNAME` column is greater than ‘JOHN’:

  ```sql
  SELECT EMPNO, LASTNAME, HIREDATE
  FROM DSN8C10.EMP
  WHERE (LASTNAME, FIRSTNAME) >= ('SMITH', 'JOHN')
  ORDER BY HIREDATE ASC;
  ```

• For **numeric based pagination**: Use the OFFSET clause (either by itself, or with the FETCH clause) to skip a specified number of rows from the result set.
  To access part of Db2 result set based on an absolute position, the OFFSET clause can be specified as part of the SELECT statement. The OFFSET clause specifies the number of rows to skip from the beginning of a result set, which can be a more efficient way to filter unneeded rows. The OFFSET clause can be used with the FETCH clause to further limit the number of rows returned from the result set.
  The following SELECT statement skips the first 100 rows from the T1 table before it returns rows for the query:

  ```sql
  SELECT * FROM T1
  OFFSET 100 ROWS;
  ```
  Using the OFFSET clause with the FETCH clause specifies the number of rows to skip from the beginning of the table before returning the number of rows specified in the FETCH clause:

  ```sql
  SELECT * FROM T1
  OFFSET 10 ROWS
  FETCH FIRST 10 ROWS ONLY;
  ```

  To return three "pages" of 10 rows each, you might use statements similar to the following SQL statements:

  ```sql
  SELECT * FROM T1
  OFFSET 0 ROWS
  FETCH FIRST 10 ROWS ONLY;

  SELECT * FROM T1
  OFFSET 10 ROWS
  FETCH NEXT 10 ROWS ONLY;

  SELECT * FROM T1
  OFFSET 20 ROWS
  FETCH NEXT 10 ROWS ONLY;
  ```

  This example is three separate SQL statements, each with different values for the OFFSET clause. Each SELECT statement is processed as a new SQL statement.
Combining result tables from multiple SELECT statements

When you combine the results of multiple SELECT statements, you can choose what to include in the result table. You can include all rows, only rows that are in the result table of both SELECT statements, or only rows that are unique to the result table of the first SELECT statement.

About this task

Assume that you want to combine the results of two SELECT statements that return the following result tables:

**Example: R1 result table**

<table>
<thead>
<tr>
<th>COL1</th>
<th>COL2</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>a</td>
</tr>
<tr>
<td>a</td>
<td>b</td>
</tr>
<tr>
<td>a</td>
<td>c</td>
</tr>
</tbody>
</table>

**Example: R2 result table**

<table>
<thead>
<tr>
<th>COL1</th>
<th>COL2</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>b</td>
</tr>
<tr>
<td>a</td>
<td>c</td>
</tr>
<tr>
<td>a</td>
<td>d</td>
</tr>
</tbody>
</table>

You can use the set operators to combine two or more SELECT statements to form a single result table:

**UNION**

UNION returns all of the values from the result table of each SELECT statement. If you want all duplicate rows to be repeated in the result table, specify UNION ALL. If you want redundant duplicate rows to be eliminated from the result table, specify UNION or UNION DISTINCT.

For example, the following example is the result of specifying UNION for R1 and R2.

<table>
<thead>
<tr>
<th>COL1</th>
<th>COL2</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>a</td>
</tr>
<tr>
<td>a</td>
<td>b</td>
</tr>
<tr>
<td>a</td>
<td>c</td>
</tr>
<tr>
<td>a</td>
<td>d</td>
</tr>
</tbody>
</table>

**EXCEPT**

Returns all rows from the first result table (R1) that are not also in the second result table (R2). If you want all duplicate rows from R1 to be contained in the result table, specify EXCEPT ALL. If you want redundant duplicate rows in R1 to be eliminated from the result table, specify EXCEPT or EXCEPT DISTINCT.

The result of the EXCEPT operation depends on which SELECT statement is included before the EXCEPT keyword in the SQL statement. For example, if the SELECT statement that returns the R1 result table is listed first, the result is a single row:

<table>
<thead>
<tr>
<th>COL1</th>
<th>COL2</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>a</td>
</tr>
</tbody>
</table>

If the SELECT statement that returns the R2 result table is listed first, the final result is a different row:

<table>
<thead>
<tr>
<th>COL1</th>
<th>COL2</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>d</td>
</tr>
</tbody>
</table>
INTERSECT

Returns rows that are in the result table of both SELECT statements. If you want all duplicate rows to be contained in the result table, specify INTERSECT ALL. If you want redundant duplicate rows to be eliminated from the result table, specify INTERSECT or INTERSECT DISTINCT.

For example, the following example is the result of specifying UNION for R1 and R2.

<table>
<thead>
<tr>
<th>COL1</th>
<th>COL2</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>b</td>
</tr>
<tr>
<td>a</td>
<td>c</td>
</tr>
</tbody>
</table>

When you specify one of the set operators, Db2 processes each SELECT statement to form an interim result table, and then combines the interim result table of each statement. If the nth column of the first result table (R1) and the nth column of the second result table (R2) have the same result column name, the nth column of the result table has that same result column name. If the nth column of R1 and the nth column of R2 do not have the same names, the result column is unnamed.

Procedure

- To combine two or more SELECT statements to form a single result table, use the set operators: UNION, EXCEPT or INTERSECT.

For example, assume that you have the following tables to manage stock at two book stores.

<table>
<thead>
<tr>
<th>Table 68. STOCKA</th>
</tr>
</thead>
<tbody>
<tr>
<td>ISBN</td>
</tr>
<tr>
<td>------</td>
</tr>
<tr>
<td>8778997709</td>
</tr>
<tr>
<td>4599877699</td>
</tr>
<tr>
<td>9228736278</td>
</tr>
<tr>
<td>1002387872</td>
</tr>
<tr>
<td>4599877699</td>
</tr>
<tr>
<td>0087873532</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Table 69. STOCKB</th>
</tr>
</thead>
<tbody>
<tr>
<td>ISBN</td>
</tr>
<tr>
<td>------</td>
</tr>
<tr>
<td>6689038367</td>
</tr>
<tr>
<td>2909788445</td>
</tr>
<tr>
<td>1182983745</td>
</tr>
<tr>
<td>9228736278</td>
</tr>
<tr>
<td>1002387872</td>
</tr>
</tbody>
</table>

Example: UNION clause

Suppose that you want a list of books whose authors have won the Nobel Prize and that are in stock at either store. The following SQL statement returns these books in order by author name without redundant duplicate rows:

```sql
SELECT TITLE, AUTHOR
FROM STOCKA
WHERE NOBELPRIZE = 'Y'
UNION
SELECT TITLE, AUTHOR
FROM STOCKB
WHERE NOBELPRIZE = 'Y'
ORDER BY AUTHOR
```
This statement returns the following final result table:

<table>
<thead>
<tr>
<th>Table 70. Result of UNION</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>TITLE</strong></td>
</tr>
<tr>
<td>The Good Earth</td>
</tr>
<tr>
<td>Light in August</td>
</tr>
<tr>
<td>Beloved</td>
</tr>
<tr>
<td>The Silent Cry</td>
</tr>
<tr>
<td>The Labyrinth of Solitude</td>
</tr>
<tr>
<td>The Grapes of Wrath</td>
</tr>
</tbody>
</table>

**Example: EXCEPT clause**

Suppose that you want a list of books that are only in STOCKA. The following SQL statement returns the book names that are in STOCKA only without any redundant duplicate rows:

```
SELECT TITLE
FROM STOCKA
EXCEPT
SELECT TITLE
FROM STOCKB
ORDER BY TITLE;
```

This statement returns the following result table:

<table>
<thead>
<tr>
<th>Table 71. Result of EXCEPT</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>TITLE</strong></td>
</tr>
<tr>
<td>For Whom the Bell Tolls</td>
</tr>
<tr>
<td>The Good Earth</td>
</tr>
<tr>
<td>The Labyrinth of Solitude</td>
</tr>
</tbody>
</table>

**Example: INTERSECT clause**

Suppose that you want a list of books that are in both STOCKA and in STOCKB. The following statement returns a list of all books from both of these tables with redundant duplicate rows are removed.

```
SELECT TITLE
FROM STOCKA
INTERSECT
SELECT TITLE
FROM STOCKB
ORDER BY TITLE;
```

This statement returns the following result table:

<table>
<thead>
<tr>
<th>Table 72. Result of INTERSECT</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>TITLE</strong></td>
</tr>
<tr>
<td>A Tale of Two Cities</td>
</tr>
<tr>
<td>Beloved</td>
</tr>
</tbody>
</table>

- To keep all duplicate rows when combining result tables, specify the ALL keyword with the set operator clause.

The following examples use the STOCKA and STOCK B tables from the previous step.
Example: UNION ALL
The following SQL statement returns a list of books that won Nobel prizes and are in stock at either store, with duplicates included.

```
SELECT TITLE, AUTHOR
FROM STOCKA
WHERE NOBELPRIZE = 'Y'
UNION ALL
SELECT TITLE, AUTHOR
FROM STOCKB
WHERE NOBELPRIZE = 'Y'
ORDER BY AUTHOR
```

This statement returns the following result table:

<table>
<thead>
<tr>
<th>Title</th>
<th>Author</th>
</tr>
</thead>
<tbody>
<tr>
<td>The Good Earth</td>
<td>Buck</td>
</tr>
<tr>
<td>The Good Earth</td>
<td>Buck</td>
</tr>
<tr>
<td>Light in August</td>
<td>Faulkner</td>
</tr>
<tr>
<td>Beloved</td>
<td>Morrison</td>
</tr>
<tr>
<td>Beloved</td>
<td>Morrison</td>
</tr>
<tr>
<td>The Silent Cry</td>
<td>Oe</td>
</tr>
<tr>
<td>The Labyrinth of Solitude</td>
<td>Paz</td>
</tr>
<tr>
<td>The Grapes of Wrath</td>
<td>Steinbeck</td>
</tr>
</tbody>
</table>

Example: EXCEPT ALL
Suppose that you want a list of books that are only in STOCKA. The following SQL statement returns the book names that are in STOCKA only with all duplicate rows:

```
SELECT TITLE
FROM STOCKA
EXCEPT ALL
SELECT TITLE
FROM STOCKB
ORDER BY TITLE;
```

This statement returns the following result table:

<table>
<thead>
<tr>
<th>Title</th>
</tr>
</thead>
<tbody>
<tr>
<td>For Whom the Bell Tolls</td>
</tr>
<tr>
<td>The Good Earth</td>
</tr>
<tr>
<td>The Good Earth</td>
</tr>
<tr>
<td>The Labyrinth of Solitude</td>
</tr>
</tbody>
</table>

Example: INTERSECT ALL
Suppose that you want a list of books that are in both STOCKA and in STOCKB, including any duplicate matches. The following statement returns a list of titles that are in both stocks, including duplicate matches. In this case, one match exists for "A Tale of Two Cities" and one match exists for "Beloved."

```
SELECT TITLE
FROM STOCKA
INTERSECT ALL
FROM STOCKB
```

374 Db2 12 for z/OS: Application Programming and SQL Guide
This statement returns the following result table:

<table>
<thead>
<tr>
<th>Title 75. Result of INTERSECT ALL</th>
</tr>
</thead>
<tbody>
<tr>
<td>TITLE</td>
</tr>
<tr>
<td>A Tale of Two Cities</td>
</tr>
<tr>
<td>Beloved</td>
</tr>
</tbody>
</table>

• To eliminate redundant duplicate rows when combining result tables, specify one of the following keywords:
  – UNION or UNION DISTINCT
  – EXCEPT or EXCEPT DISTINCT
  – INTERSECT or INTERSECT DISTINCT

• To order the entire result table, specify the ORDER BY clause at the end.

Related tasks
Ordering the result table rows
If you want to guarantee that the rows in your result table are ordered in a particular way, you must specify the order in the SELECT statement. Otherwise, Db2 can return the rows in any order.

Related reference
fullselect (Db2 SQL)

Summarizing group values
You can group rows in the result table by the values of one or more columns or by the results of an expression. You can then apply aggregate functions to each group.

Procedure
Use the GROUP BY clause.

When it is used, the GROUP BY clause follows the FROM clause and any WHERE clause, and it precedes the ORDER BY clause.

Except for the columns that are named in the GROUP BY clause, the SELECT statement must specify any other selected columns as an operand of one of the aggregate functions.

If a column that you specify in the GROUP BY clause contains null values, Db2 considers those null values to be equal. Thus, all nulls form a single group.

Examples
Example: GROUP BY clause using one column
The following SQL statement lists, for each department, the lowest and highest education level within that department:

```sql
SELECT WORKDEPT, MIN(EDLEVEL), MAX(EDLEVEL)
FROM DSN8C10.EMP
GROUP BY WORKDEPT;
```

Example: GROUP BY clause using more than one column
You can group the rows by the values of more than one column. For example, The following statement finds the average salary for men and women in departments A00 and C01:

```sql
SELECT WORKDEPT, SEX, AVG(SALARY) AS AVG_SALARY
FROM DSN8C10.EMP
```
WHERE WORKDEPT IN ('A00', 'C01')
GROUP BY WORKDEPT, SEX;

The result table looks similar to the following output:

<table>
<thead>
<tr>
<th>WORKDEPT</th>
<th>SEX</th>
<th>AVG_SALARY</th>
</tr>
</thead>
<tbody>
<tr>
<td>A00</td>
<td>F</td>
<td>49625.0000000000</td>
</tr>
<tr>
<td>A00</td>
<td>M</td>
<td>35000.0000000000</td>
</tr>
<tr>
<td>C01</td>
<td>F</td>
<td>29722.5000000000</td>
</tr>
</tbody>
</table>

Db2 groups the rows first by department number and then (within each department) by sex before it derives the average SALARY value for each group.

**Example: GROUP BY clause using a expression**

You can also group the rows by the results of an expression. For example, the following statement groups departments by their leading characters, and lists the lowest and highest education level for each group:

```sql
SELECT SUBSTR(WORKDEPT,1,1), MIN(EDLEVEL), MAX(EDLEVEL)
FROM DSN8C10.EMP
GROUP BY SUBSTR(WORKDEPT,1,1);
```

**Related reference**

group-by-clause (Db2 SQL)

**Filtering groups**

If you group rows in the result table, you can also specify a search condition that each retrieved group must satisfy. The search condition tests properties of each group rather than properties of individual rows in the group.

**Procedure**

Use the HAVING clause to specify a search condition.

The HAVING clause acts like a WHERE clause for groups, and it contains the same kind of search conditions that you specify in a WHERE clause.

**Example**

**Example: HAVING clause**

The following SQL statement includes a HAVING clause that specifies a search condition for groups of work departments in the employee table:

```sql
SELECT WORKDEPT, AVG(SALARY) AS AVG_SALARY
FROM DSN8C10.EMP
GROUP BY WORKDEPT
HAVING COUNT(*) > 1
ORDER BY WORKDEPT;
```

The result table looks similar to the following output:

<table>
<thead>
<tr>
<th>WORKDEPT</th>
<th>AVG_SALARY</th>
</tr>
</thead>
<tbody>
<tr>
<td>A00</td>
<td>40850.0000000000</td>
</tr>
<tr>
<td>C01</td>
<td>29722.5000000000</td>
</tr>
<tr>
<td>D11</td>
<td>25147.2727272727</td>
</tr>
<tr>
<td>D21</td>
<td>25668.5714285714</td>
</tr>
<tr>
<td>E11</td>
<td>21020.0000000000</td>
</tr>
<tr>
<td>E21</td>
<td>24886.6666666666</td>
</tr>
</tbody>
</table>

Compare the preceding example with the second example shown in “Summarizing group values” on page 375. The clause, HAVING COUNT(*) > 1, ensures that only departments with more than one member are displayed. In this case, departments B01 and E01 do not display because the HAVING clause tests a property of the group.
**Example: HAVING clause used with a GROUP BY clause**

Use the HAVING clause to retrieve the average salary and minimum education level of women in each department for which all female employees have an education level greater than or equal to 16. Assuming that you want results from only departments A00 and D11, the following SQL statement tests the group property, MIN(EDLEVEL):

```
SELECT WORKDEPT, AVG(SALARY) AS AVG_SALARY, MIN(EDLEVEL) AS MIN_EDLEVEL
FROM DSN8C10.EMP
WHERE SEX = 'F' AND WORKDEPT IN ('A00', 'D11')
GROUP BY WORKDEPT
HAVING MIN(EDLEVEL) >= 16;
```

The result table looks similar to the following output:

<table>
<thead>
<tr>
<th>WORKDEPT</th>
<th>AVG_SALARY</th>
<th>MIN_EDLEVEL</th>
</tr>
</thead>
<tbody>
<tr>
<td>A00</td>
<td>49625.00000000</td>
<td>18</td>
</tr>
<tr>
<td>D11</td>
<td>25817.50000000</td>
<td>17</td>
</tr>
</tbody>
</table>

When you specify both GROUP BY and HAVING, the HAVING clause must follow the GROUP BY clause. A function in a HAVING clause can include DISTINCT if you have not used DISTINCT anywhere else in the same SELECT statement. You can also connect multiple predicates in a HAVING clause with AND or OR, and you can use NOT for any predicate of a search condition.

**Related reference**
- [where-clause (Db2 SQL)](https://www.ibm.com/docs/en/db2?topic=select-where-clause)

**Finding rows that were changed within a specified period of time**

You can filter rows based on the time that they were updated. For example, you might want to find all rows in a particular table that have been changed in the last 7 days.

**Procedure**

Specify the ROW CHANGE TIMESTAMP expression in the predicate of your SQL statement.

**Recommendation:** Ensure that the table has a ROW CHANGE TIMESTAMP column that was defined prior to the time period that you want to query. This column ensures that Db2 returns only those rows that were updated in the given time period.

If the table does not have a ROW CHANGE TIMESTAMP column, Db2 returns all rows on each page that has had any changes within the given time period. In this case, your result set can contain rows that have not been updated in the given time period, if other rows on that page have been updated or inserted.

**Examples**

**Example**

Suppose that the TAB table has a ROW CHANGE TIMESTAMP column and that you want to return all of the records that have changed in the last 30 days. The following query returns all of those rows.

```
SELECT * FROM TAB
WHERE ROW CHANGE TIMESTAMP FOR TAB <= CURRENT TIMESTAMP AND
  ROW CHANGE TIMESTAMP FOR TAB >= CURRENT_TIMESTAMP - 30 days;
```

**Example**

Suppose that you want to return all of the records that have changed since 9:00 AM January 1, 2004. The following query returns all of those rows.

```
SELECT * FROM TAB
WHERE ROW CHANGE TIMESTAMP FOR TAB >= '2004-01-01-09.00.00';
```
Joining data from more than one table

Sometimes the information that you want to see is not in a single table. To form a row of the result table, you might want to retrieve some column values from one table and some column values from another table.

About this task

You can use a SELECT statement to retrieve and join column values from two or more tables into a single row.

A join operation typically matches a row of one table with a row of another on the basis of a join condition. Db2 supports the following types of joins: inner join, left outer join, right outer join, and full outer join. You can specify joins in the FROM clause of a query.

Examples

Nested table expressions and user-defined table functions in joins

An operand of a join can be more complex than the name of a single table. You can specify one of the following items as a join operand:

- **nested table expression**
  A fullselect that is enclosed in parentheses and followed by a correlation name. The correlation name lets you refer to the result of that expression.

  Using a nested table expression in a join can be helpful when you want to create a temporary table to use in a join. You can specify the nested table expression as either the right or left operand of a join, depending on which unmatched rows you want included.

- **user-defined table function**
  A user-defined function that returns a table.

  Using a nested table expression in a join can be helpful when you want to perform some operation on the values in a table before you join them to another table.

Example: Using correlated references

In the following SELECT statement, the correlation name that is used for the nested table expression is CHEAP_PARTS. You can use this correlation name to refer to the columns that are returned by the expression. In this case, those correlated references are CHEAP_PARTS.PROD# and CHEAP_PARTS.PRODUCT.

```
SELECT CHEAP_PARTS.PROD#, CHEAP_PARTS.PRODUCT
FROM (SELECT PROD#, PRODUCT
      FROM PRODUCTS
      WHERE PRICE < 10) AS CHEAP_PARTS;
```

The result table looks similar to the following output:

<table>
<thead>
<tr>
<th>PROD#</th>
<th>PRODUCT</th>
</tr>
</thead>
<tbody>
<tr>
<td>505</td>
<td>SCREWDRIVER</td>
</tr>
<tr>
<td>30</td>
<td>RELAY</td>
</tr>
</tbody>
</table>

The correlated references are valid because they do not occur in the table expression where CHEAP_PARTS is defined. The correlated references are from a table specification at a higher level in the hierarchy of subqueries.
Example: Using a nested table expression as the right operand of a join
The following query contains a fullselect (in bold) as the right operand of a left outer join with the PROJECTS table. The correlation name is TEMP. In this case the unmatched rows from the PROJECTS table are included, but the unmatched rows from the nested table expression are not.

```
SELECT PROJECT, COALESCE(PROJECTS.PROD#, PRODNUM) AS PRODNUM,
       PRODUCT, PART, UNITS
FROM PROJECTS LEFT JOIN
     (SELECT PART,
      COALESCE(PARTS.PROD#, PRODUCTS.PROD#) AS PRODNUM,
      PRODUCTS.PRODUCT
     FROM PARTS FULL OUTER JOIN PRODUCTS
     ON PARTS.PROD# = PRODUCTS.PROD#) AS TEMP
ON PROJECTS.PROD# = PRODNUM;
```

Example: Using a nested table expression as the left operand of a join
The following query contains a fullselect as the left operand of a left outer join with the PRODUCTS table. The correlation name is PARTX. In this case the unmatched rows from the nested table expression are included, but the unmatched rows from the PRODUCTS table are not.

```
SELECT PART, SUPPLIER, PRODNUM, PRODUCT
FROM (SELECT PART, PROD# AS PRODNUM, SUPPLIER
     FROM PARTS
     WHERE PROD# < '200') AS PARTX
LEFT OUTER JOIN PRODUCTS
ON PRODNUM = PROD#;
```

The result table looks similar to the following output:

<table>
<thead>
<tr>
<th>PART</th>
<th>SUPPLIER</th>
<th>PRODNUM</th>
<th>PRODUCT</th>
</tr>
</thead>
<tbody>
<tr>
<td>WIRE</td>
<td>ACWF</td>
<td>10</td>
<td>GENERATOR</td>
</tr>
<tr>
<td>MAGNETS</td>
<td>BATEMAN</td>
<td>10</td>
<td>GENERATOR</td>
</tr>
<tr>
<td>OIL</td>
<td>WESTERN_CHEM</td>
<td>160</td>
<td>----------</td>
</tr>
</tbody>
</table>

Because PROD# is a character field, Db2 does a character comparison to determine the set of rows in the result. Therefore, because the characters '30' are greater than '200', the row in which PROD# is equal to '30' does not appear in the result.

Example: Using a table function as an operand of a join
Suppose that CVTPRICE is a table function that converts the prices in the PRODUCTS table to the currency that you specify and returns the PRODUCTS table with the prices in those units. You can obtain a table of parts, suppliers, and product prices with the prices in your choice of currency by executing a query similar to the following query:

```
SELECT PART, SUPPLIER, PARTS.PROD#, Z.PRODUCT, Z.PRICE
FROM PARTS, TABLE(CVTPRICE(:CURRENCY)) AS Z
WHERE PARTS.PROD# = Z.PROD#;
```

Correlated references in table specifications in joins
Use correlation names to refer to the results of a nested table expression. After you specify the correlation name for an expression, any subsequent reference to this correlation name is called a correlated reference.

You can include correlated references in nested table expressions or as arguments to table functions. The basic rule that applies for both of these cases is that the correlated reference must be from a table specification at a higher level in the hierarchy of subqueries. You can also use a correlated reference and the table specification to which it refers in the same FROM clause if the table specification appears to the left of the correlated reference and the correlated reference is in one of the following clauses:

- A nested table expression that is preceded by the keyword TABLE
- The argument of a table function

For more information about correlated references, see “Correlation names in references” on page 396.
A table function or a table expression that contains correlated references to other tables in the same FROM clause cannot participate in a full outer join or a right outer join. The following examples illustrate valid uses of correlated references in table specifications.

In this example, the correlated reference T.C2 is valid because the table specification, to which it refers, T, is to its left.

```sql
SELECT T.C1, Z.C5
FROM T, TABLE(TF3(T.C2)) AS Z
WHERE T.C3 = Z.C4;
```

If you specify the join in the opposite order, with T following TABLE(TF3(T.C2), T.C2 is invalid.

In this example, the correlated reference D.DEPTNO is valid because the nested table expression within which it appears is preceded by TABLE, and the table specification D appears to the left of the nested table expression in the FROM clause.

```sql
SELECT D.DEPTNO, D.DEPTNAME,
       EMPINFO.AVGSAL, EMPINFO.EMPCOUNT
FROM DEPT D,
     TABLE(SELECT AVG(E.SALARY) AS AVGSAL,
           COUNT(*) AS EMPCOUNT
           FROM EMP E
           WHERE E.WORKDEPT=D.DEPTNO) AS EMPINFO;
```

If you remove the keyword TABLE, D.DEPTNO is invalid.

**Related reference**
from-clause (Db2 SQL)

**Joining more than two tables**
Joins are not limited to two tables. You can join more than two tables in a single SQL statement.

**Procedure**
Specify join conditions that include columns from all of the relevant tables.

**Example**

**Example: Joining three tables**
Suppose that you want a result table that shows employees who have projects that they are responsible for, their projects, and their department names. You need to join three tables to get all the information. You can use the following SELECT statement:

```sql
SELECT EMPNO, LASTNAME, DEPTNAME, PROJNO
FROM DSN8C10.EMP, DSN8C10.PROJ, DSN8C10.DEPT
WHERE EMPNO = RESPEMP
AND WORKDEPT = DSN8C10.DEPT.DEPTNO;
```

The result table looks similar to the following output:

<table>
<thead>
<tr>
<th>EMPNO</th>
<th>LASTNAME</th>
<th>DEPTNAME</th>
<th>PROJNO</th>
</tr>
</thead>
<tbody>
<tr>
<td>000010</td>
<td>HAAS</td>
<td>SPIFFY COMPUTER SERVICE DIV</td>
<td>AD3100</td>
</tr>
<tr>
<td>000010</td>
<td>HAAS</td>
<td>SPIFFY COMPUTER SERVICE DIV</td>
<td>MA2100</td>
</tr>
<tr>
<td>000020</td>
<td>THOMPSON</td>
<td>PLANNING</td>
<td>PL2100</td>
</tr>
<tr>
<td>000030</td>
<td>KWAN</td>
<td>INFORMATION CENTER</td>
<td>IF1000</td>
</tr>
<tr>
<td>000030</td>
<td>KWAN</td>
<td>INFORMATION CENTER</td>
<td>IF2000</td>
</tr>
<tr>
<td>000050</td>
<td>GEYER</td>
<td>SUPPORT SERVICES</td>
<td>OP1000</td>
</tr>
<tr>
<td>000050</td>
<td>GEYER</td>
<td>SUPPORT SERVICES</td>
<td>OP2000</td>
</tr>
<tr>
<td>000060</td>
<td>STERN</td>
<td>MANUFACTURING SYSTEMS</td>
<td>MA2110</td>
</tr>
<tr>
<td>000070</td>
<td>PULASKI</td>
<td>ADMINISTRATION SYSTEMS</td>
<td>AD3110</td>
</tr>
<tr>
<td>000090</td>
<td>HENDERSON</td>
<td>OPERATIONS</td>
<td>OP1010</td>
</tr>
<tr>
<td>000100</td>
<td>SPENGER</td>
<td>SOFTWARE SUPPORT</td>
<td>OP2010</td>
</tr>
<tr>
<td>000150</td>
<td>ADAMSON</td>
<td>MANUFACTURING SYSTEMS</td>
<td>MA2112</td>
</tr>
<tr>
<td>000160</td>
<td>PIANKA</td>
<td>MANUFACTURING SYSTEMS</td>
<td>MA2113</td>
</tr>
<tr>
<td>000220</td>
<td>LUTZ</td>
<td>MANUFACTURING SYSTEMS</td>
<td>MA2111</td>
</tr>
<tr>
<td>000230</td>
<td>JEFFERSON</td>
<td>ADMINISTRATION SYSTEMS</td>
<td>AD3111</td>
</tr>
<tr>
<td>000250</td>
<td>SMITH</td>
<td>ADMINISTRATION SYSTEMS</td>
<td>AD3112</td>
</tr>
</tbody>
</table>
Db2 determines the intermediate and final results of the previous query by performing the following logical steps:

1. Join the employee and project tables on the employee number, dropping the rows with no matching employee number in the project table.
2. Join the intermediate result table with the department table on matching department numbers.
3. Process the select list in the final result table, leaving only four columns.

**Example: Joining more than two tables by using more than one join type**

When joining more than two tables, you do not have to use the same join type for every join.

To join tables by using more than one join type, specify the join types in the FROM clause.

Suppose that you want a result table that shows the following items:

- Employees whose last name begins with 'S' or a letter that comes after 'S' in the alphabet
- The department names for these employees
- Any projects that these employees are responsible for

You can use the following SELECT statement:

```sql
SELECT EMPNO, LASTNAME, DEPTNAME, PROJNO
FROM DSN8C10.EMP INNER JOIN DSN8C10.DEPT
ON WORKDEPT = DSN8C10.DEPT.DEPTNO
LEFT OUTER JOIN DSN8C10.PROJ
ON EMPNO = RESPEMP
WHERE LASTNAME > 'S';
```

The result table looks like similar to the following output:

<table>
<thead>
<tr>
<th>EMPNO</th>
<th>LASTNAME</th>
<th>DEPTNAME</th>
<th>PROJNO</th>
</tr>
</thead>
<tbody>
<tr>
<td>000020</td>
<td>THOMPSON</td>
<td>PLANNING</td>
<td>PL2100</td>
</tr>
<tr>
<td>000060</td>
<td>STERN</td>
<td>MANUFACTURING SYSTEMS</td>
<td>MA2110</td>
</tr>
<tr>
<td>000100</td>
<td>SPENCER</td>
<td>SOFTWARE SUPPORT</td>
<td>OP2010</td>
</tr>
<tr>
<td>000170</td>
<td>YOSHIMURA</td>
<td>MANUFACTURING SYSTEMS</td>
<td>------</td>
</tr>
<tr>
<td>000190</td>
<td>WALKER</td>
<td>MANUFACTURING SYSTEMS</td>
<td>------</td>
</tr>
<tr>
<td>000250</td>
<td>SMITH</td>
<td>ADMINISTRATION SYSTEMS</td>
<td>AD3112</td>
</tr>
<tr>
<td>000300</td>
<td>SMITH</td>
<td>OPERATIONS</td>
<td>------</td>
</tr>
<tr>
<td>000310</td>
<td>SETRIGHT</td>
<td>OPERATIONS</td>
<td>------</td>
</tr>
<tr>
<td>000330</td>
<td>WONG</td>
<td>SOFTWARE SUPPORT</td>
<td>------</td>
</tr>
</tbody>
</table>

Db2 determines the intermediate and final results of the previous query by performing the following logical steps:

1. Join the employee and department tables on matching department numbers, dropping the rows where the last name begins with a letter before 'S' in the alphabet.
2. Join the intermediate result table with the project table on the employee number, keeping the rows for which no matching employee number exists in the project table.
3. Process the select list in the final result table, leaving only four columns.

**Related reference**

*from-clause (Db2 SQL)*
**Inner joins**

An *inner join* is a method of combining two tables that discards rows of either table that do not match any row of the other table. The matching is based on the join condition.

To request an inner join, execute a SELECT statement in which you specify the tables that you want to join in the FROM clause, and specify a WHERE clause or an ON clause to indicate the join condition. The join condition can be any simple or compound search condition that does not contain a subquery reference.

In the simplest type of inner join, the join condition is `column1=column2`.

**Example**

You can join the PARTS and PRODUCTS tables in sample data from joins on the PROD# column to get a table of parts with their suppliers and the products that use the parts.

To do this, you can use either one of the following SELECT statements:

```sql
SELECT PART, SUPPLIER, PARTS.PROD#, PRODUCT
FROM PARTS, PRODUCTS
WHERE PARTS.PROD# = PRODUCTS.PROD#;

SELECT PART, SUPPLIER, PARTS.PROD#, PRODUCT
FROM PARTS INNER JOIN PRODUCTS
ON PARTS.PROD# = PRODUCTS.PROD#;
```

The result table looks like the following output:

<table>
<thead>
<tr>
<th>PART</th>
<th>SUPPLIER</th>
<th>PROD#</th>
<th>PRODUCT</th>
</tr>
</thead>
<tbody>
<tr>
<td>WIRE</td>
<td>ACWF</td>
<td>10</td>
<td>GENERATOR</td>
</tr>
<tr>
<td>MAGNETS</td>
<td>BATEMAN</td>
<td>10</td>
<td>GENERATOR</td>
</tr>
<tr>
<td>PLASTIC</td>
<td>PLASTIK_CORP</td>
<td>30</td>
<td>RELAY</td>
</tr>
<tr>
<td>BLADES</td>
<td>ACE_STEEL</td>
<td>205</td>
<td>SAW</td>
</tr>
</tbody>
</table>

Three things about this example:

- A part in the parts table (OIL) has product (#160), which is not in the products table. A product (SCREWDRIVER, #505) has no parts listed in the parts table. Neither OIL nor SCREWDRIVER appears in the result of the join.

In contrast, an *outer join* includes rows in which the values in the joined columns do not match.

- You can explicitly specify that this join is an inner join (not an outer join). Use INNER JOIN in the FROM clause instead of the comma, and use ON to specify the join condition (rather than WHERE) when you explicitly join tables in the FROM clause.

- If you do not specify a WHERE clause in the first form of the query, the result table contains all possible combinations of rows for the tables that are identified in the FROM clause. You can obtain the same result by specifying a join condition that is always true in the second form of the query, as in the following statement:

```sql
SELECT PART, SUPPLIER, PARTS.PROD#, PRODUCT
FROM PARTS INNER JOIN PRODUCTS
ON 1=1;
```

Regardless of whether you omit the WHERE clause or specify a join condition that is always true, the number of rows in the result table is the product of the number of rows in each table.

You can specify more complicated join conditions to obtain different sets of results. For example, to eliminate the suppliers that begin with the letter A from the table of parts, suppliers, product numbers, and products, write a query like the following query:

```sql
SELECT PART, SUPPLIER, PARTS.PROD#, PRODUCT
FROM PARTS INNER JOIN PRODUCTS
ON PARTS.PROD# = PRODUCTS.PROD#
AND SUPPLIER NOT LIKE 'A%';
```
The result of the query is all rows that do not have a supplier that begins with A. The result table looks like the following output:

<table>
<thead>
<tr>
<th>PART</th>
<th>SUPPLIER</th>
<th>PROD#</th>
<th>PRODUCT</th>
</tr>
</thead>
<tbody>
<tr>
<td>MAGNETS</td>
<td>BATEMAN</td>
<td>10</td>
<td>GENERATOR</td>
</tr>
<tr>
<td>PLASTIC</td>
<td>PLASTIK_CORP</td>
<td>30</td>
<td>RELAY</td>
</tr>
</tbody>
</table>

**Example of joining a table to itself by using an inner join**

Joining a table to itself is useful to show relationships between rows. The following example returns a list of major projects from the PROJ table and the projects that are part of those major projects.

In this example, A indicates the first instance of table DSN8C10.PROJ, and B indicates the second instance of this table. The join condition is such that the value in column PROJNO in table DSN8C10.PROJ A must be equal to a value in column MAJPROJ in table DSN8C10.PROJ B.

The following SQL statement joins table DSN8C10.PROJ to itself and returns the number and name of each major project followed by the number and name of the project that is part of it:

```sql
SELECT A.PROJNO, A.PROJNAME, B.PROJNO, B.PROJNAME
FROM DSN8C10.PROJ A, DSN8C10.PROJ B
WHERE A.PROJNO = B.MAJPROJ;
```

The result table looks similar to the following output:

<table>
<thead>
<tr>
<th>PROJNO</th>
<th>PROJNAME</th>
<th>PROJNO</th>
<th>PROJNAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>AD3100</td>
<td>ADMIN SERVICES</td>
<td>AD3110</td>
<td>GENERAL AD SYSTEMS</td>
</tr>
<tr>
<td>AD3110</td>
<td>GENERAL AD SYSTEMS</td>
<td>AD3111</td>
<td>PAYROLL PROGRAMMING</td>
</tr>
<tr>
<td>AD3110</td>
<td>GENERAL AD SYSTEMS</td>
<td>AD3112</td>
<td>PERSONNEL PROGRAMMG</td>
</tr>
<tr>
<td>OP2010</td>
<td>SYSTEMS SUPPORT</td>
<td>OP2013</td>
<td>DB/DC SUPPORT</td>
</tr>
</tbody>
</table>

In this example, the comma in the FROM clause implicitly specifies an inner join, and it acts the same as if the INNER JOIN keywords had been used. When you use the comma for an inner join, you must specify the join condition on the WHERE clause. When you use the INNER JOIN keywords, you must specify the join condition on the ON clause.

**Related reference**

- **Sample data for joins**
  You can use the sample PARTS table and the PRODUCTS table to practice various types of joins.
- **from-clause (Db2 SQL)**

**Outer joins**

An outer join is a method of combining two or more tables so that the result includes unmatched rows of one of the tables, or of both tables. The matching is based on the join condition.

Db2 supports three types of outer joins:

- **full outer join**
  Includes unmatched rows from both tables. If any column of the result table does not have a value, that column has the null value in the result table.

- **left outer join**
  Includes rows from the table that is specified before LEFT OUTER JOIN that have no matching values in the table that is specified after LEFT OUTER JOIN.

- **right outer join**
  Includes rows from the table that is specified after RIGHT OUTER JOIN that have no matching values in the table that is specified before RIGHT OUTER JOIN.

The following table illustrates how the PARTS and PRODUCTS tables in “Sample data for joins” on page 387 can be combined using the three outer join functions.
The result table contains data that is joined from all of the tables, for rows that satisfy the search conditions.

The result columns of a join have names if the outermost SELECT list refers to base columns. However, if you use a function (such as COALESCE or VALUE) to build a column of the result, that column does not have a name unless you use the AS clause in the SELECT list.

**Full outer join**

An **full outer join** is a method of combining tables so that the result includes unmatched rows of both tables.

If you are joining two tables and want the result set to include unmatched rows from both tables, use a FULL OUTER JOIN clause. The matching is based on the join condition. If any column of the result table does not have a value, that column has the null value in the result table.

The join condition for a full outer join must be a simple search condition that compares two columns or an invocation of a cast function that has a column name as its argument.

**Example:** The following query performs a full outer join of the PARTS and PRODUCTS tables in “Sample data for joins” on page 387:

```sql
SELECT PART, SUPPLIER, PARTS.PROD#, PRODUCT
FROM PARTS FULL OUTER JOIN PRODUCTS
ON PARTS.PROD# = PRODUCTS.PROD#;
```

The result table from the query looks similar to the following output:

<table>
<thead>
<tr>
<th>PART</th>
<th>SUPPLIER</th>
<th>PROD#</th>
<th>PRODUCT</th>
</tr>
</thead>
<tbody>
<tr>
<td>WIRE</td>
<td>ACWF</td>
<td>10</td>
<td>GENERATOR</td>
</tr>
<tr>
<td>MAGNETS</td>
<td>BATEMAN</td>
<td>10</td>
<td>GENERATOR</td>
</tr>
<tr>
<td>BLADES</td>
<td>205</td>
<td>10</td>
<td>RELAY</td>
</tr>
<tr>
<td>PLASTIC</td>
<td>PLASTIK_CORP</td>
<td>30</td>
<td>SAW</td>
</tr>
<tr>
<td>OIL</td>
<td>ACE_STEEL</td>
<td>205</td>
<td>SCREWDRIVER</td>
</tr>
<tr>
<td></td>
<td>WESTERN_CHEM</td>
<td>160</td>
<td></td>
</tr>
</tbody>
</table>

**Example of using COALESCE or VALUE:** COALESCE is the keyword that is specified by the SQL standard as a synonym for the VALUE function. This function, by either name, can be particularly useful in full outer join operations because it returns the first non-null value from the pair of join columns.
The product number in the result of the example for “Full outer join” on page 384 is null for SCREWDRIVER, even though the PRODUCTS table contains a product number for SCREWDRIVER. If you select PRODUCTS.PROD# instead, PROD# is null for OIL. If you select both PRODUCTS.PROD# and PARTS.PROD#, the result contains two columns, both of which contain some null values. You can merge data from both columns into a single column, eliminating the null values, by using the COALESCE function.

With the same PARTS and PRODUCTS tables, the following example merges the non-null data from the PROD# columns:

```
SELECT PART, SUPPLIER, 
COALESCE(PARTS.PROD#, PRODUCTS.PROD#) AS PRODNUM, PRODUCT 
FROM PARTS FULL OUTER JOIN PRODUCTS
ON PARTS.PROD# = PRODUCTS.PROD#;
```

The result table looks similar to the following output:

<table>
<thead>
<tr>
<th>PART</th>
<th>SUPPLIER</th>
<th>PRODNUM</th>
<th>PRODUCT</th>
</tr>
</thead>
<tbody>
<tr>
<td>WIRE</td>
<td>ACWF</td>
<td>10</td>
<td>GENERATOR</td>
</tr>
<tr>
<td>MAGNETS</td>
<td>BATEMAN</td>
<td>10</td>
<td>GENERATOR</td>
</tr>
<tr>
<td>PLASTIC</td>
<td>PLASTIK_CORP</td>
<td>30</td>
<td>RELAY</td>
</tr>
<tr>
<td>BLADES</td>
<td>ACE_STEEL</td>
<td>205</td>
<td>SAW</td>
</tr>
<tr>
<td>OIL</td>
<td>WESTERN_CHEM</td>
<td>160</td>
<td>---------</td>
</tr>
</tbody>
</table>

The AS clause (AS PRODNUM) provides a name for the result of the COALESCE function.

**Left outer join**

A left outer join is a method of combining tables. The result includes unmatched rows from only the table that is specified before the LEFT OUTER JOIN clause.

If you are joining two tables and want the result set to include unmatched rows from only one table, use a LEFT OUTER JOIN clause or a RIGHT OUTER JOIN clause. The matching is based on the join condition.

The clause LEFT OUTER JOIN includes rows from the table that is specified before LEFT OUTER JOIN that have no matching values in the table that is specified after LEFT OUTER JOIN.

As in an inner join, the join condition can be any simple or compound search condition that does not contain a subquery reference.

**Example:** The following example uses the tables in “Sample data for joins” on page 387. To include rows from the PARTS table that have no matching values in the PRODUCTS table, and to include prices that exceed 10.00, run the following query:

```
SELECT PART, SUPPLIER, PARTS.PROD#, PRODUCT, PRICE 
FROM PARTS LEFT OUTER JOIN PRODUCTS 
ON PARTS.PROD#=PRODUCTS.PROD# 
AND PRODUCTS.PRICE>10.00;
```

The result table looks similar to the following output:

<table>
<thead>
<tr>
<th>PART</th>
<th>SUPPLIER</th>
<th>PROD#</th>
<th>PRODUCT</th>
<th>PRICE</th>
</tr>
</thead>
<tbody>
<tr>
<td>WIRE</td>
<td>ACWF</td>
<td>10</td>
<td>GENERATOR</td>
<td>45.75</td>
</tr>
<tr>
<td>MAGNETS</td>
<td>BATEMAN</td>
<td>10</td>
<td>GENERATOR</td>
<td>45.75</td>
</tr>
<tr>
<td>PLASTIC</td>
<td>PLASTIK_CORP</td>
<td>30</td>
<td>---------</td>
<td>------</td>
</tr>
<tr>
<td>BLADES</td>
<td>ACE_STEEL</td>
<td>205</td>
<td>SAW</td>
<td>18.90</td>
</tr>
<tr>
<td>OIL</td>
<td>WESTERN_CHEM</td>
<td>160</td>
<td>---------</td>
<td>------</td>
</tr>
</tbody>
</table>

A row from the PRODUCTS table is in the result table only if its product number matches the product number of a row in the PARTS table and the price is greater than 10.00 for that row. Rows in which the PRICE value does not exceed 10.00 are included in the result of the join, but the PRICE value is set to null.

In this result table, the row for PROD# 30 has null values on the right two columns because the price of PROD# 30 is less than 10.00. PROD# 160 has null values on the right two columns because PROD# 160 does not match another product number.
**Right outer join**

A right outer join is a method of combining tables. The result includes unmatched rows from only the table that is specified after the RIGHT OUTER JOIN clause.

If you are joining two tables and want the result set to include unmatched rows from only one table, use a LEFT OUTER JOIN clause or a RIGHT OUTER JOIN clause. The matching is based on the join condition.

The clause RIGHT OUTER JOIN includes rows from the table that is specified after RIGHT OUTER JOIN that have no matching values in the table that is specified before RIGHT OUTER JOIN.

As in an inner join, the join condition can be any simple or compound search condition that does not contain a subquery reference.

**Example:** The following example uses the tables in “Sample data for joins” on page 387. To include rows from the PRODUCTS table that have no corresponding rows in the PARTS table, execute this query:

```sql
SELECT PART, SUPPLIER, PRODUCTS.PROD#, PRODUCT, PRICE
FROM PARTS RIGHT OUTER JOIN PRODUCTS
ON PARTS.PROD# = PRODUCTS.PROD# AND PRODUCTS.PRICE > 10.00;
```

The result table looks similar to the following output:

<table>
<thead>
<tr>
<th>PART</th>
<th>SUPPLIER</th>
<th>PROD#</th>
<th>PRODUCT</th>
<th>PRICE</th>
</tr>
</thead>
<tbody>
<tr>
<td>WIRE</td>
<td>ACWF</td>
<td>10</td>
<td>GENERATOR</td>
<td>45.75</td>
</tr>
<tr>
<td>MAGNETS</td>
<td>BATEMAN</td>
<td>10</td>
<td>GENERATOR</td>
<td>45.75</td>
</tr>
<tr>
<td>BLADES</td>
<td>ACE_STEEL</td>
<td>205</td>
<td>SAW</td>
<td>18.90</td>
</tr>
<tr>
<td></td>
<td></td>
<td>30</td>
<td>RELAY</td>
<td>7.55</td>
</tr>
<tr>
<td></td>
<td></td>
<td>505</td>
<td>SCREWDRIVER</td>
<td>3.70</td>
</tr>
</tbody>
</table>

A row from the PARTS table is in the result table only if its product number matches the product number of a row in the PRODUCTS table and the price is greater than 10.00 for that row.

Because the PRODUCTS table can have rows with nonmatching product numbers in the result table, and the PRICE column is in the PRODUCTS table, rows in which PRICE is less than or equal to 10.00 are included in the result. The PARTS columns contain null values for these rows in the result table.

**SQL rules for statements that contain join operations**

Typically, Db2 performs a join operation first, before it evaluates the other clauses of the SELECT statement.

SQL rules dictate that the result of a SELECT statement look as if the clauses had been evaluated in this order:

- FROM
- WHERE
- GROUP BY
- HAVING
- SELECT

A join operation is part of a FROM clause; therefore, for the purpose of predicting which rows will be returned from a SELECT statement that contains a join operation, assume that the join operation is performed first.

**Example:** Suppose that you want to obtain a list of part names, supplier names, product numbers, and product names from the PARTS and PRODUCTS tables. You want to include rows from either table where the PROD# value does not match a PROD# value in the other table, which means that you need to do a full outer join. You also want to exclude rows for product number 10. Consider the following SELECT statement:

```sql
SELECT PART, SUPPLIER,
VALUE(PARTS.PROD#,PRODUCTS.PROD#) AS PRODNUM, PRODUCT
FROM PARTS FULL OUTER JOIN PRODUCTS
```
ON PARTS.PROD# = PRODUCTS.PROD#
WHERE PARTS.PROD# <> '10' AND PRODUCTS.PROD# <> '10';

The following result is **not** what you wanted:

<table>
<thead>
<tr>
<th>PART</th>
<th>SUPPLIER</th>
<th>PRODNUM</th>
<th>PRODUCT</th>
</tr>
</thead>
<tbody>
<tr>
<td>PLASTIC</td>
<td>PLASTIK_CORP</td>
<td>30</td>
<td>RELAY</td>
</tr>
<tr>
<td>BLADES</td>
<td>ACE_STEEL</td>
<td>205</td>
<td>SAW</td>
</tr>
</tbody>
</table>

Db2 performs the join operation first. The result of the join operation includes rows from one table that do not have corresponding rows from the other table. However, the WHERE clause then excludes the rows from both tables that have null values for the PROD# column.

The following statement is a correct SELECT statement to produce the list:

```sql
SELECT PART, SUPPLIER,
VALUE(X.PROD#, Y.PROD#) AS PRODNUM, PRODUCT
FROM
(SELECT PART, SUPPLIER, PROD# FROM PARTS WHERE PROD# <> '10') X
FULL OUTER JOIN
(SELECT PROD#, PRODUCT FROM PRODUCTS WHERE PROD# <> '10') Y
ON X.PROD# = Y.PROD#;
```

For this statement, Db2 applies the WHERE clause to each table separately. Db2 then performs the full outer join operation, which includes rows in one table that do not have a corresponding row in the other table. The final result includes rows with the null value for the PROD# column and looks similar to the following output:

<table>
<thead>
<tr>
<th>PART</th>
<th>SUPPLIER</th>
<th>PRODNUM</th>
<th>PRODUCT</th>
</tr>
</thead>
<tbody>
<tr>
<td>OIL</td>
<td>WESTERN_CHEM</td>
<td>160</td>
<td>-----------</td>
</tr>
<tr>
<td>BLADES</td>
<td>ACE_STEEL</td>
<td>205</td>
<td>SAW</td>
</tr>
<tr>
<td>PLASTIC</td>
<td>PLASTIK_CORP</td>
<td>30</td>
<td>RELAY</td>
</tr>
<tr>
<td></td>
<td></td>
<td>505</td>
<td>SCREWDRIVER</td>
</tr>
</tbody>
</table>

**Sample data for joins**

You can use the sample PARTS table and the PRODUCTS table to practice various types of joins.

The examples in these topics use the following two tables to show various types of joins:

<table>
<thead>
<tr>
<th>The PARTS table</th>
<th>The PRODUCTS table</th>
</tr>
</thead>
<tbody>
<tr>
<td>PART</td>
<td>PROD#</td>
</tr>
<tr>
<td>------</td>
<td>-------</td>
</tr>
<tr>
<td>WIRE</td>
<td>10</td>
</tr>
<tr>
<td>OIL</td>
<td>160</td>
</tr>
<tr>
<td>MAGNETS</td>
<td>10</td>
</tr>
<tr>
<td>PLASTIC</td>
<td>30</td>
</tr>
<tr>
<td>BLADES</td>
<td>205</td>
</tr>
</tbody>
</table>

**Optimizing retrieval for a small set of rows**

When you need only a few of the thousands of rows that satisfy a query, you can tell Db2 to optimize its retrieval process to return only a specified number of rows.

**About this task**

**Question:** How can I tell Db2 that I want only a few of the thousands of rows that satisfy a query?

**Answer:** Use the optimize clause or the fetch clause of the SELECT statement.

Db2 usually optimizes queries to retrieve all rows that qualify. But sometimes you want to retrieve a few rows. For example, to retrieve the first row that is greater than or equal to a known value, code your SELECT statement like the following:

```sql
SELECT column list FROM table
WHERE key >= value
ORDER BY key ASC
```
Even with the ORDER BY clause, Db2 might fetch all the data first and sort it after the fetch, which could impact performance. Instead, you can write the query in one of the following ways:

```
SELECT * FROM table
WHERE key >= value
ORDER BY key ASC
OPTIMIZE FOR 1 ROW
```

```
SELECT * FROM table
WHERE key >= value
ORDER BY key ASC
FETCH FIRST n ROWS ONLY
```

Use OPTIMIZE FOR 1 ROW clause to influence the access path. OPTIMIZE FOR 1 ROW tells Db2 to select an access path that returns the first qualifying row quickly.

Use FETCH FIRST n ROWS ONLY clause to limit the number of rows in the result table to $n$ rows. FETCH FIRST $n$ ROWS ONLY has the following benefits:

- When you use FETCH statements to retrieve data from a result table, the fetch clause causes Db2 to retrieve only the number of rows that you need. This can have performance benefits, especially in distributed applications. If you try to execute a FETCH statement to retrieve the $n+1$st row, Db2 returns a +100 SQLCODE.

- When you use fetch clause in a SELECT INTO statement, you never retrieve more than one row. Using fetch clause in a SELECT INTO statement can prevent SQL errors that are caused by inadvertently selecting more than one value into a host variable.

When you specify the fetch clause but not the optimize clause, the optimize clause is implicit. When you specify FETCH FIRST $n$ ROWS ONLY and OPTIMIZE FOR $m$ ROWS, and $m$ is less than $n$, Db2 optimizes the query for $m$ rows. If $m$ is greater than $n$, Db2 optimizes the query for $n$ rows.

**Related tasks**
Fetching a limited number of rows (Db2 Performance)

**Related reference**
optimize-clause (Db2 SQL)
fetch-clause (Db2 SQL)

### Creating recursive SQL by using common table expressions

Queries that use recursion are useful in applications like bill-of-materials applications, network planning applications, and reservation systems.

**About this task**

You can use common table expressions to create recursive SQL. If a fullselect of a common table expression contains a reference to itself in a FROM clause, the common table expression is a *recursive common table expression*.

Recursive common table expressions must follow these rules:

- The first fullselect of the first union (the initialization fullselect) must not include a reference to the common table expression.
- Each fullselect that is part of the recursion cycle must:
  - Start with SELECT or SELECT ALL. SELECT DISTINCT is not allowed.
  - Include only one reference to the common table expression that is part of the recursion cycle in its FROM clause.
  - Not include aggregate functions, a GROUP BY clause, or a HAVING clause.
- The column names must be specified after the table name of the common table expression.
- The data type, length, and CCSID of each column from the common table expression must match the data type, length, and CCSID of each corresponding column in the iterative fullselect.
• If you use the UNION keyword, specify UNION ALL instead of UNION.
• You cannot specify INTERSECT or EXCEPT.
• Outer joins must not be part of any recursion cycle.
• A subquery must not be part of any recursion cycle.

**Important:** You should be careful to avoid an infinite loop when you use a recursive common table expression. Db2 issues a warning if one of the following items is not found in the iterative fullselect of a recursive common table expression:

• An integer column that increments by a constant
• A predicate in the WHERE clause in the form of `counter_column < constant` or `counter_column < :host variable`

See “Examples of recursive common table expressions” on page 145 for examples of bill-of-materials applications that use recursive common table expressions.

### Updating data as it is retrieved from the database

As you retrieve rows, you can update them at the same time.

**About this task**

**Question:** How can I update rows of data as I retrieve them?

**Answer:** On the SELECT statement, use the FOR UPDATE clause without a column list, or the FOR UPDATE OF clause with a column list. For a more efficient program, specify a column list with only those columns that you intend to update. Then use the positioned UPDATE statement. The clause WHERE CURRENT OF identifies the cursor that points to the row you want to update.

### Avoiding decimal arithmetic errors

When you request that Db2 perform a decimal operation, errors might occur if Db2 does not use the appropriate precision and scale.

**About this task**

For static SQL statements, the simplest way to avoid a division error is to override DEC31 rules by specifying the precompiler option DEC(15). In some cases you can avoid a division error by specifying D31.s, where s is a number between 1 and 9 and represents the minimum scale to be used for division operations. This specification reduces the probability of errors for statements that are embedded in the program.

If the dynamic SQL statements have bind, define, or invoke behavior and the value of the installation option for USE FOR DYNAMICRULES on panel DSNTIP4 is NO, you can use the precompiler option DEC(15), DEC15, or D15.s to override DEC31 rules, where s is a number between 1 and 9.

For a dynamic statement, or for a single static statement, use the scalar function DECIMAL to specify values of the precision and scale for a result that causes no errors.

Before you execute a dynamic statement, set the value of special register CURRENT PRECISION to DEC15 or D15.s, where s is a number between 1 and 9.

Even if you use DEC31 rules, multiplication operations can sometimes cause overflow because the precision of the product is greater than 31. To avoid overflow from multiplication of large numbers, use the MULTIPLY_ALT built-in function instead of the multiplication operator.
**Precision for operations with decimal numbers**

Db2 accepts two sets of rules for determining the precision and scale of the result of an operation with decimal numbers.

- **DEC15 rules** allow a maximum precision of 15 digits in the result of an operation. DEC15 rules are in effect when both operands have a precision of 15 or less, or unless the DEC31 rules apply.

- **DEC31 rules** allow a maximum precision of 31 digits in the result. DEC31 rules are in effect if any of the following conditions is true:
  - Either operand of the operation has a precision greater than 15 digits.
  - The operation is in a dynamic SQL statement, and any of the following conditions is true:
    - The current value of special register CURRENT PRECISION is DEC31 or D31.s, where s is a number between 1 and 9 and represents the minimum scale to be used for division operations.
    - The installation option for DECIMAL ARITHMETIC on panel DSNTIP4 is DEC31, 31, or D31.s, where s is a number between 1 and 9; the installation option for USE FOR DYNAMICRULES on panel DSNTIP4 is YES; and the value of CURRENT PRECISION has not been set by the application.
    - The SQL statement has bind, define, or invoke behavior; the statement is in an application that is precompiled with option DEC(31); the installation option for USE FOR DYNAMICRULES on panel DSNTIP4 is NO; and the value of CURRENT PRECISION has not been set by the application. See “Dynamic rules options for dynamic SQL statements” on page 879 for an explanation of bind, define, and invoke behavior.
  - The operation is in an embedded (static) SQL statement that you precompiled with the DEC(31), DEC31, or D31.s option, or with the default for that option when the installation option DECIMAL ARITHMETIC is DEC31 or 31. s is a number between 1 and 9 and represents the minimum scale to be used for division operations. See “Processing SQL statements” on page 839 for information about precompiling and for a list of all precompiler options.

**Recommendation:** To reduce the chance of overflow, or when dealing with a precision greater than 15 digits, choose DEC31 or D31.s, where s is a number between 1 and 9 and represents the minimum scale to be used for division operations.

**Controlling how Db2 rounds decimal floating point numbers**

You can specify a default rounding mode that Db2 is to use for all DECIMAL values.

**Procedure**

Set the CURRENT DECIMAL ROUNDING MODE special register.

**Related reference**

CURRENT DECIMAL ROUNDING MODE (Db2 SQL)
SET CURRENT DECIMAL ROUNDING MODE (Db2 SQL)

**Implications of using SELECT * **

Generally, you should use SELECT * only when you want to select all columns, except for hidden columns. Otherwise, specify the specific columns that you want to view.

**Question:** What are the implications of using SELECT * ?

**Answer:** Generally, you should select only the columns you need because Db2 is sensitive to the number of columns selected. Use SELECT * only when you are sure you want to select all columns, except hidden columns. (Hidden columns are not returned when you specify SELECT *.) One alternative to selecting all columns is to use views defined with only the necessary columns, and use SELECT * to access the views. Avoid SELECT * if all the selected columns participate in a sort operation (SELECT DISTINCT and SELECT...UNION, for example).
Subqueries

When you need to narrow your search condition based on information in an interim table, you can use a subquery. For example, you might want to find all employee numbers in one table that also exist for a given project in a second table.

Conceptual overview of subqueries

Suppose that you want a list of the employee numbers, names, and commissions of all employees who work on a particular project, whose project number is MA2111. The first part of the SELECT statement is easy to write:

```
SELECT EMPNO, LASTNAME, COMM
FROM DSN8C10.EMP
WHERE EMPNO
```

However, you cannot proceed because the DSN8C10.EMP table does not include project number data. You do not know which employees are working on project MA2111 without issuing another SELECT statement against the DSN8C10.EMPPROJACT table.

You can use a subquery to solve this problem. A subquery is a subselect or a fullselect in a WHERE clause. The SELECT statement that surrounds the subquery is called the outer SELECT.

```
SELECT EMPNO, LASTNAME, COMM
FROM DSN8C10.EMP
WHERE EMPNO IN
  (SELECT EMPNO
   FROM DSN8C10.EMPPROJACT
   WHERE PROJNO = 'MA2111');
```

To better understand the results of this SQL statement, imagine that Db2 goes through the following process:

1. Db2 evaluates the subquery to obtain a list of EMPNO values:

```
(SELECT EMPNO
 FROM DSN8C10.EMPPROJACT
 WHERE PROJNO = 'MA2111');
```

The result is in an interim result table, similar to the one in the following output:

```
from EMPNO
 =====
 200
 290
 220
```

2. The interim result table then serves as a list in the search condition of the outer SELECT. Effectively, Db2 executes this statement:

```
SELECT EMPNO, LASTNAME, COMM
FROM DSN8C10.EMP
WHERE EMPNO IN
  ('000200', '000220');
```

As a consequence, the result table looks similar to the following output:

```
EMPNO   LASTNAME  COMM
======  ========  ====
000200  BROWN     2217
000220  LUTZ      2387
```
Correlated and uncorrelated subqueries

Subqueries supply information that is needed to qualify a row (in a WHERE clause) or a group of rows (in a HAVING clause). The subquery produces a result table that is used to qualify the row or group of selected rows.

A subquery executes only once, if the subquery is the same for every row or group. This kind of subquery is uncorrelated, which means that it executes only once. For example, in the following statement, the content of the subquery is the same for every row of the table DSN8C10.EMP:

```
SELECT EMPNO, LASTNAME, COMM
FROM DSN8C10.EMP
WHERE EMPNO IN
  (SELECT EMPNO
   FROM DSN8C10.EMPPROJECT
   WHERE PROJNO = 'MA2111');
```

Subqueries that vary in content from row to row or group to group are correlated subqueries. For information about correlated subqueries, see “Correlated subqueries” on page 395.

Subqueries and predicates

A predicate is an element of a search condition that specifies a condition that is true, false, or unknown about a given row or group. A subquery, which is a SELECT statement within the WHERE or HAVING clause of another SQL statement, is always part of a predicate. The predicate is of the form:

```
operand operator (subquery)
```

A WHERE or HAVING clause can include predicates that contain subqueries. A predicate that contains a subquery, like any other search predicate, can be enclosed in parentheses, can be preceded by the keyword NOT, and can be linked to other predicates through the keywords AND and OR. For example, the WHERE clause of a query can look something like the following clause:

```
WHERE X IN (subquery1) AND (Y > SOME (subquery2) OR Z IS NULL)
```

Subqueries can also appear in the predicates of other subqueries. Such subqueries are nested subqueries at some level of nesting. For example, a subquery within a subquery within an outer SELECT has a nesting level of 2. Db2 allows nesting down to a level of 15, but few queries require a nesting level greater than 1.

The relationship of a subquery to its outer SELECT is the same as the relationship of a nested subquery to a subquery, and the same rules apply, except where otherwise noted.

The subquery result table

A subquery must produce a result table that has the same number of columns as the number of columns on the left side of the comparison operator. For example, both of the following SELECT statements are acceptable:

```
SELECT EMPNO, LASTNAME
FROM DSN8C10.EMP
WHERE SALARY =
  (SELECT AVG(SALARY)
   FROM DSN8C10.EMP);
```

```
SELECT EMPNO, LASTNAME
FROM DSN8C10.EMP
WHERE (SALARY, BONUS) IN
  (SELECT AVG(SALARY), AVG(BONUS)
   FROM DSN8C10.EMP);
```

Except for a subquery of a basic predicate, the result table can contain more than one row. For more information, see “Places where you can include a subquery” on page 393.

Related concepts

Subquery access (Db2 Performance)
**Predicates (Db2 SQL)**

**Related tasks**
- Writing efficient subqueries (Db2 Performance)

**Related reference**
- where-clause (Db2 SQL)
- having-clause (Db2 SQL)

**Places where you can include a subquery**

You can specify a subquery in either a WHERE clause or a HAVING clause.

You can specify a subquery in either a WHERE or HAVING clause by using one of the following items:

**Example: Basic predicate in a subquery**

You can use a subquery immediately after any of the comparison operators. If you do, the subquery can return at most one value. Db2 compares that value with the value to the left of the comparison operator.

The following SQL statement returns the employee numbers, names, and salaries for employees whose education level is higher than the average company-wide education level.

```sql
SELECT EMPNO, LASTNAME, SALARY
FROM DSN8C10.EMP
WHERE EDLEVEL >
     (SELECT AVG(EDLEVEL)
      FROM DSN8C10.EMP);
```

**Example: Quantified predicate in a subquery: ALL, ANY, or SOME**

You can use a subquery after a comparison operator, followed by the keyword ALL, ANY, or SOME. The number of columns and rows that the subquery can return for a quantified predicate depends on the type of quantified predicate:

- For = SOME, = ANY, or <> ALL, the subquery can return one or many rows and one or many columns. The number of columns in the result table must match the number of columns on the left side of the operator.
- For all other quantified predicates, the subquery can return one or many rows, but no more than one column.

See the information about quantified predicates, including what to do if a subquery that returns one or more null values gives you unexpected results.

**Example: ALL predicate**

Use ALL to indicate that the operands on the left side of the comparison must compare in the same way with all of the values that the subquery returns. For example, suppose that you use the greater-than comparison operator with ALL:

```sql
WHERE column > ALL (subquery)
```

To satisfy this WHERE clause, the column value must be greater than all of the values that the subquery returns. A subquery that returns an empty result table satisfies the predicate.

Now suppose that you use the <> operator with ALL in a WHERE clause like this:

```sql
WHERE (column1, column1, ... columnn) <> ALL (subquery)
```

To satisfy this WHERE clause, each column value must be unequal to all of the values in the corresponding column of the result table that the subquery returns. A subquery that returns an empty result table satisfies the predicate.
Example: ANY or SOME predicate

Use ANY or SOME to indicate that the values on the left side of the operator must compare in the indicated way to at least one of the values that the subquery returns. For example, suppose that you use the greater-than comparison operator with ANY:

```
WHERE expression > ANY (subquery)
```

To satisfy this WHERE clause, the value in the expression must be greater than at least one of the values (that is, greater than the lowest value) that the subquery returns. A subquery that returns an empty result table does not satisfy the predicate.

Now suppose that you use the = operator with SOME in a WHERE clause like this:

```
WHERE (column1, column1, ... columnn) = SOME (subquery)
```

To satisfy this WHERE clause, each column value must be equal to at least one of the values in the corresponding column of the result table that the subquery returns. A subquery that returns an empty result table does not satisfy the predicate.

Example: IN predicate in a subquery

You can use IN to say that the value or values on the left side of the IN operator must be among the values that are returned by the subquery. Using IN is equivalent to using = ANY or = SOME.

The following query returns the names of department managers:

```
SELECT EMPNO, LASTNAME
FROM DSN8C10.EMP
WHERE EMPNO IN
  (SELECT DISTINCT MGRNO
   FROM DSN8C10.DEPT);
```

EXISTS predicate in a subquery

When you use the keyword EXISTS, Db2 checks whether the subquery returns one or more rows. Returning one or more rows satisfies the condition; returning no rows does not satisfy the condition.

The search condition in the following query is satisfied if any project that is represented in the project table has an estimated start date that is later than 1 January 2005:

```
SELECT EMPNO, LASTNAME
FROM DSN8C10.EMP
WHERE EXISTS
  (SELECT *
   FROM DSN8C10.PROJ
   WHERE PRSTDATE > '2005-01-01');
```

The result of the subquery is always the same for every row that is examined for the outer SELECT. Therefore, either every row appears in the result of the outer SELECT or none appears. A correlated subquery is more powerful than the uncorrelated subquery that is used in this example because the result of a correlated subquery is evaluated for each row of the outer SELECT.

As shown in the example, you do not need to specify column names in the subquery of an EXISTS clause. Instead, you can code SELECT *. You can also use the EXISTS keyword with the NOT keyword in order to select rows when the data or condition that you specify does not exist; that is, you can code the following clause:

```
WHERE NOT EXISTS (SELECT ...);
```

Related tasks
Writing efficient subqueries (Db2 Performance)

Related reference
Quantified predicate (Db2 SQL)
Correlated subqueries

A correlated subquery is a subquery that Db2 reevaluates when it examines a new row (in a WHERE clause) or a group of rows (in a HAVING clause) as it executes the outer SELECT statement.

In an uncorrelated subquery, Db2 executes the subquery once, substitutes the result of the subquery in the right side of the search condition, and evaluates the outer SELECT based on the value of the search condition.

User-defined functions in correlated subqueries

Use care when you invoke a user-defined function in a correlated subquery, and that user-defined function uses a scratchpad. Db2 does not refresh the scratchpad between invocations of the subquery. This can cause undesirable results because the scratchpad keeps values across the invocations of the subquery.

An example of a correlated subquery

Suppose that you want a list of all the employees whose education levels are higher than the average education levels in their respective departments. To get this information, Db2 must search the DSN8C10.EMP table. For each employee in the table, Db2 needs to compare the employee's education level to the average education level for that employee's department.

For this example, you need to use a correlated subquery, which differs from an uncorrelated subquery. An uncorrelated subquery compares the employee's education level to the average of the entire company, which requires looking at the entire table. A correlated subquery evaluates only the department that corresponds to the particular employee.

In the subquery, you tell Db2 to compute the average education level for the department number in the current row. The following query performs this action:

```
SELECT EMPNO, LASTNAME, WORKDEPT, EDLEVEL
FROM DSN8C10.EMP X
WHERE EDLEVEL > (SELECT AVG(EDLEVEL)
    FROM DSN8C10.EMP
    WHERE WORKDEPT = X.WORKDEPT);
```

A correlated subquery looks like an uncorrelated one, except for the presence of one or more correlated references. In the example, the single correlated reference is the occurrence of X.WORKDEPT in the WHERE clause of the subselect. In this clause, the qualifier X is the correlation name that is defined in the FROM clause of the outer SELECT statement. X designates rows of the first instance of DSN8C10.EMP. At any time during the execution of the query, X designates the row of DSN8C10.EMP to which the WHERE clause is being applied.

Consider what happens when the subquery executes for a given row of DSN8C10.EMP. Before it executes, X.WORKDEPT receives the value of the WORKDEPT column for that row. Suppose, for example, that the row is for Christine Haas. Her work department is A00, which is the value of WORKDEPT for that row. Therefore, the following is the subquery that is executed for that row:

```
(SELECT AVG(EDLEVEL)
    FROM DSN8C10.EMP
    WHERE WORKDEPT = 'A00');
```

The subquery produces the average education level of Christine's department. The outer SELECT then compares this average to Christine's own education level. For some other row for which WORKDEPT has a different value, that value appears in the subquery in place of A00. For example, in the row for Michael...
L Thompson, this value is B01, and the subquery for his row delivers the average education level for department B01.

The result table that is produced by the query is similar to the following output:

<table>
<thead>
<tr>
<th>EMPNO</th>
<th>LASTNAME</th>
<th>WORKDEPT</th>
<th>EDLEVEL</th>
</tr>
</thead>
<tbody>
<tr>
<td>000010</td>
<td>HASS</td>
<td>A00</td>
<td>18</td>
</tr>
<tr>
<td>000030</td>
<td>KWAN</td>
<td>C01</td>
<td>20</td>
</tr>
<tr>
<td>000070</td>
<td>PULASKI</td>
<td>D21</td>
<td>16</td>
</tr>
<tr>
<td>000090</td>
<td>HENDERSON</td>
<td>E11</td>
<td>16</td>
</tr>
</tbody>
</table>

Related concepts
Correlated and non-correlated subqueries (Db2 Performance)

Related reference
having-clause (Db2 SQL)
where-clause (Db2 SQL)

Correlation names in references
A correlation name is a name that you specify for a table, view, nested table expression or table function. This name is valid only within the context in which it is defined. Use correlation names to avoid ambiguity, to establish correlated references, or to use shorter names for tables or views.

A correlated reference can appear in a subquery, in a nested table expression, or as an argument of a user-defined table function. For information about correlated references in nested table expressions and table functions, see “Joining data from more than one table” on page 378. In a subquery, the reference should be of the form X.C, where X is a correlation name and C is the name of a column in the table that X represents.

Any number of correlated references can appear in a subquery, with no restrictions on variety. For example, you can use one correlated reference in the outer SELECT, and another in a nested subquery.

When you use a correlated reference in a subquery, the correlation name can be defined in the outer SELECT or in any of the subqueries that contain the reference. Suppose, for example, that a query contains subqueries A, B, and C, and that A contains B and B contains C. The subquery C can use a correlation reference that is defined in B, A, or the outer SELECT.

You can define a correlation name for each table name in a FROM clause. Specify the correlation name after its table name. Leave one or more blanks between a table name and its correlation name. You can include the word AS between the table name and the correlation name to increase the readability of the SQL statement.

The following example demonstrates the use of a correlated reference in the search condition of a subquery:

```sql
SELECT EMPNO, LASTNAME, WORKDEPT, EDLEVEL
FROM DSN8C10.EMP AS X
WHERE EDLEVEL >
(SELECT AVG(EDLEVEL)
FROM DSN8C10.EMP
WHERE WORKDEPT = X.WORKDEPT);
```

The following example demonstrates the use of a correlated reference in the select list of a subquery:

```sql
UPDATE BP1TBL T1
SET (KEY1, CHAR1, VCHAR1) =
(SELECT VALUE(T2.KEY1,T1.KEY1), VALUE(T2.CHAR1,T1.CHAR1),
VALUE(T2.VCHAR1,T1.VCHAR1)
FROM BP2TBL T2
WHERE (T2.KEY1 = T1.KEY1))
WHERE KEY1 IN
(SELECT KEY1
FROM BP2TBL T3
WHERE KEY2 > 0);
```

Using correlated subqueries in an UPDATE statement:
Use correlation names in an UPDATE statement to refer to the rows that you are updating. The subquery for which you specified a correlation name is called a *correlated subquery*.

For example, when all activities of a project must complete before September 2006, your department considers that project to be a priority project. Assume that you have added the PRIORITY column to DSN8C10.PROJ. You can use the following SQL statement to evaluate the projects in the DSN8C10.PROJ table, and write a 1 (a flag to indicate PRIORITY) in the PRIORITY column for each priority project:

```
UPDATE DSN8C10.PROJ X
SET PRIORITY = 1
WHERE DATE('2006-09-01') >
(SELECT MAX(ACENDATE)
  FROM DSN8C10.PROJACT
  WHERE PROJNO = X.PROJNO);
```

As Db2 examines each row in the DSN8C10.PROJ table, it determines the maximum activity end date (the ACENDATE column) for all activities of the project (from the DSN8C10.PROJACT table). If the end date of each activity that is associated with the project is before September 2006, the current row in the DSN8C10.PROJ table qualifies, and Db2 updates it.

**Using correlated subqueries in a DELETE statement:**

Use correlation names in a DELETE statement to refer to the rows that you are deleting. The subquery for which you specified a correlation name is called a *correlated subquery*. Db2 evaluates the correlated subquery once for each row in the table that is named in the DELETE statement to decide whether to delete the row.

**Using tables with no referential constraints:**

Suppose that a department considers a project to be complete when the combined amount of time currently spent on it is less than or equal to half of a person’s time. The department then deletes the rows for that project from the DSN8C10.PROJ table. In the examples in this topic, PROJ and PROJACT are independent tables; that is, they are separate tables with no referential constraints defined on them.

```
DELETE FROM DSN8C10.PROJ X
WHERE .5 >
(SELECT SUM(ACSTAFF)
  FROM DSN8C10.PROJACT
  WHERE PROJNO = X.PROJNO);
```

To process this statement, Db2 determines for each project (represented by a row in the DSN8C10.PROJ table) whether the combined staffing for that project is less than 0.5. If it is, Db2 deletes that row from the DSN8C10.PROJ table.

To continue this example, suppose that Db2 deletes a row in the DSN8C10.PROJ table. You must also delete rows that are related to the deleted project in the DSN8C10.PROJACT table. To do this, use a statement similar to this statement:

```
DELETE FROM DSN8C10.PROJACT X
WHERE NOT EXISTS
  (SELECT *
   FROM DSN8C10.PROJ
   WHERE PROJNO = X.PROJNO);
```

Db2 determines, for each row in the DSN8C10.PROJACT table, whether a row with the same project number exists in the DSN8C10.PROJ table. If not, Db2 deletes the row from DSN8C10.PROJACT.

**Using a single table:**

A subquery of a searched DELETE statement (a DELETE statement that does not use a cursor) can reference the same table from which rows are deleted. In the following statement, which deletes the employee with the highest salary from each department, the employee table appears in the outer DELETE and in the subselect:

```
DELETE FROM YEMP X
WHERE SALARY = (SELECT MAX(SALARY) FROM YEMP Y
  WHERE X.WORKDEPT = Y.WORKDEPT);
```
This example uses a copy of the employee table for the subquery.

The following statement, without a correlated subquery, yields equivalent results:

```
DELETE FROM YEMP
WHERE (SALARY, WORKDEPT) IN (SELECT MAX(SALARY), WORKDEPT
FROM YEMP
GROUP BY WORKDEPT);
```

**Using tables with referential constraints:**

Db2 restricts delete operations for dependent tables that are involved in referential constraints. If a DELETE statement has a subquery that references a table that is involved in the deletion, make the last delete rule in the path to that table RESTRICT or NO ACTION. This action ensures that the result of the subquery is not materialized before the deletion occurs. However, if the result of the subquery is materialized before the deletion, the delete rule can also be CASCADE or SET NULL.

**Example:** Without referential constraints, the following statement deletes departments from the department table whose managers are not listed correctly in the employee table:

```
DELETE FROM DSN8C10.DEPT THIS
WHERE NOT DEPTNO =
(SELECT WORKDEPT
FROM DSN8C10.EMP
WHERE EMPNO = THIS.MGRNO);
```

With the referential constraints that are defined for the sample tables, this statement causes an error because the result table for the subquery is not materialized before the deletion occurs. Because DSN8C10.EMP is a dependent table of DSN8C10.DEPT, the deletion involves the table that is referred to in the subquery, and the last delete rule in the path to EMP is SET NULL, not RESTRICT or NO ACTION. If the statement could execute, its results would depend on the order in which Db2 accesses the rows. Therefore, Db2 prohibits the deletion.

**Restrictions when using distinct types with UNION, EXCEPT, and INTERSECT**

Db2 enforces strong typing of distinct types with UNION, EXCEPT, and INTERSECT. When you use these keywords to combine column values from several tables, the combined columns must be of the same types. If a column is a distinct type, the corresponding column must be the same distinct type.

**Example:** Suppose that you create a view that combines the values of the US_SALES, EUROPEAN_SALES, and JAPAN_SALES tables. The TOTAL columns in the three tables are of different distinct types. Before you combine the table values, you must convert the types of two of the TOTAL columns to the type of the third TOTAL column. Assume that the US_DOLLAR type has been chosen as the common distinct type. Because Db2 does not generate cast functions to convert from one distinct type to another, two user-defined functions must exist:

- A function called EURO_TO_US that converts values of type EURO to type US_DOLLAR
- A function called YEN_TO_US that converts values of type JAPANESE_YEN to type US_DOLLAR

Then you can execute a query like this to display a table of combined sales:

```
SELECT PRODUCT_ITEM, MONTH, YEAR, TOTAL
FROM US_SALES
UNION
SELECT PRODUCT_ITEM, MONTH, YEAR, EURO_TO_US(TOTAL)
FROM EUROPEAN_SALES
UNION
SELECT PRODUCT_ITEM, MONTH, YEAR, YEN_TO_US(TOTAL)
FROM JAPAN_SALES;
```

Because the result type of both the YEN_TO_US function and the EURO_TO_US function is US_DOLLAR, you have satisfied the requirement that the distinct types of the combined columns are the same.
Comparison of distinct types

You can compare an object with a distinct type only to an object with exactly the same distinct type. You cannot compare data of a distinct type directly to data of its source type. However, you can compare a distinct type to its source type by using a cast function.

The basic rule for comparisons is that the data types of the operands must be compatible. The compatibility rule defines, for example, that all numeric types (SMALLINT, INTEGER, FLOAT, and DECIMAL) are compatible. That is, you can compare an INTEGER value with a value of type FLOAT. However, you cannot compare an object of a distinct type to an object of a different type. You can compare an object with a distinct type only to an object with exactly the same distinct type.

For example, suppose you want to know which products sold more than $100 000.00 in the US in the month of July in 2003 (7/03). Because you cannot compare data of type US_DOLLAR with instances of data of the source type of US_DOLLAR (DECIMAL) directly, you must use a cast function to cast data from DECIMAL to US_DOLLAR or from US_DOLLAR to DECIMAL. Whenever you create a distinct type, Db2 creates two cast functions, one to cast from the source type to the distinct type and the other to cast from the distinct type to the source type. For distinct type US_DOLLAR, Db2 creates a cast function called DECIMAL and a cast function called US_DOLLAR. When you compare an object of type US_DOLLAR to an object of type DECIMAL, you can use one of those cast functions to make the data types identical for the comparison. Suppose table US_SALES is defined like this:

```
CREATE TABLE US_SALES
  (PRODUCT_ITEM  INTEGER,
   MONTH         INTEGER CHECK (MONTH BETWEEN 1 AND 12),
   YEAR          INTEGER CHECK (YEAR > 1990),
   TOTAL         US_DOLLAR);
```

Then you can cast DECIMAL data to US_DOLLAR like this:

```
SELECT PRODUCT_ITEM
FROM   US_SALES
WHERE  TOTAL > US_DOLLAR(100000.00)
AND    MONTH = 7
AND    YEAR  = 2003;
```

The casting satisfies the requirement that the compared data types are identical.

You cannot use host variables in statements that you prepare for dynamic execution. As explained in “Dynamically executing an SQL statement by using PREPARE and EXECUTE” on page 520, you can substitute parameter markers for host variables when you prepare a statement, and then use host variables when you execute the statement.

If you use a parameter marker in a predicate of a query, and the column to which you compare the value represented by the parameter marker is of a distinct type, you must cast the parameter marker to the distinct type, or cast the column to its source type.

For example, suppose that distinct type CNUM is defined like this:

```
CREATE DISTINCT TYPE CNUM AS INTEGER;
```

Table CUSTOMER is defined like this:

```
CREATE TABLE CUSTOMER
  (CUST_NUM     CNUM NOT NULL,
   FIRST_NAME   CHAR(30) NOT NULL,
   LAST_NAME    CHAR(30) NOT NULL,
   PHONE_NUM    CHAR(20) WITH DEFAULT,
   PRIMARY KEY  (CUST_NUM));
```

In an application program, you prepare a SELECT statement that compares the CUST_NUM column to a parameter marker. Because CUST_NUM is of a distinct type, you must cast the distinct type to its source type:

```
SELECT FIRST_NAME, LAST_NAME, PHONE_NUM FROM CUSTOMER
WHERE CAST(CUST_NUM AS INTEGER) = ?
```
Alternatively, you can cast the parameter marker to the distinct type:

```sql
SELECT FIRST_NAME, LAST_NAME, PHONE_NUM FROM CUSTOMER
WHERE CUST_NUM = CAST (?) AS CNUM)
```

### Nested SQL statements

An SQL statement can explicitly invoke user-defined functions or stored procedures or can implicitly activate triggers that invoke user-defined functions or stored procedures. This situation is known as nesting of SQL statements. Db2 supports as many as to 64 levels of nesting.

The following example shows SQL statement nesting.

```
Trigger TR1 is defined on table T3:
CREATE TRIGGER TR1
AFTER UPDATE ON T3
FOR EACH STATEMENT MODE DB2SQL
BEGIN ATOMIC
  CALL SP3(PARM1);
END
```

Program P1 (nesting level 1) contains:
```
SELECT UDF1(C1) FROM T1;
```
UDF1 (nesting level 2) contains:
```
CALL SP2(C2);
```
SP2 (nesting level 3) contains:
```
UPDATE T3 SET C3=1;
```
SP3 (nesting level 4) contains:
```
SELECT UDF4(C4) FROM T4;
```
⋮
SP16 (nesting level 16) cannot invoke stored procedures
or user-defined functions

Be aware of the following Db2 restrictions on nested SQL statements:

- **Restrictions for SELECT statements:**
  
  When you execute a SELECT statement on a table, you cannot execute INSERT, UPDATE, MERGE, or DELETE statements on the same table at a lower level of nesting.

  For example, suppose that you execute this SQL statement at level 1 of nesting:

  ```sql
  SELECT UDF1(C1) FROM T1;
  ```

  You cannot execute this SQL statement at a lower level of nesting:

  ```sql
  INSERT INTO T1 VALUES(...);
  ```

- **Restrictions for SELECT FROM FINAL TABLE statements that specify INSERT, UPDATE, or DELETE statements to change data:**
  
  When you execute this type of statement, an error occurs if both of the following conditions exist:
  
  - The SELECT statement that modifies data (by specifying INSERT, UPDATE, or DELETE) activates an AFTER TRIGGER.
  - The AFTER TRIGGER results in additional nested SQL operations that modify the table that is the target of the original SELECT statement that modifies data.

- **Restrictions for INSERT, UPDATE, MERGE, and DELETE statements:**
  
  When you execute an INSERT, UPDATE, MERGE, or DELETE statement on a table, you cannot access that table from a user-defined function or stored procedure that is at a lower level of nesting.

  For example, suppose that you execute this SQL statement at level 1 of nesting:

  ```sql
  DELETE FROM T1 WHERE UDF3(T1.C1) = 3;
  ```
You cannot execute this SELECT statement at a lower level of nesting:

```
SELECT * FROM T1;
```

If the AFTER trigger is not activated by an INSERT, UPDATE, or DELETE data change statement that is specified in a data-change-table-reference SELECT FROM FINAL TABLE, the preceding list of restrictions do not apply to SQL statements that are executed at a lower level of nesting as a result of an after trigger. For example, suppose an UPDATE statement at nesting level 1 activates an after update trigger, which calls a stored procedure. The stored procedure executes two SQL statements that reference the triggering table: one SELECT statement and one INSERT statement. In this situation, both the SELECT and the INSERT statements can be executed even though they are at nesting level 3.

Although trigger activations count in the levels of SQL statement nesting, the previous restrictions on SQL statements do not apply to SQL statements that are executed in the trigger body.

**Example:** Suppose that trigger TR1 is defined on table T1:

```
CREATE TRIGGER TR1
AFTER INSERT ON T1
FOR EACH STATEMENT MODE DB2SQL
BEGIN ATOMIC
  UPDATE T1 SET C1=1;
END
```

Now suppose that you execute this SQL statement at level 1 of nesting:

```
INSERT INTO T1 VALUES(…);
```

Although the UPDATE statement in the trigger body is at level 2 of nesting and modifies the same table that the triggering statement updates, Db2 can execute the INSERT statement successfully.

**Retrieving a set of rows by using a cursor**

In an application program, you can retrieve a set of rows from a table or a result table that is returned by a stored procedure. You can retrieve one or more rows at a time.

**About this task**

Use either of the following types of cursors to retrieve rows from a result table:

- A row-positioned cursor retrieves at most a single row at a time from the result table into host variables. At any point in time, the cursor is positioned on at most a single row. For information about how to use a row-positioned cursor, see “Accessing data by using a row-positioned cursor” on page 406.
- A rowset-positioned cursor retrieves zero, one, or more rows at a time, as a rowset, from the result table into host-variable arrays. At any point in time, the cursor can be positioned on a rowset. You can reference all of the rows in the rowset, or only one row in the rowset, when you use a positioned DELETE or positioned UPDATE statement. For information about how to use a rowset-positioned cursor, see “Accessing data by using a rowset-positioned cursor” on page 410.

**Cursors**

A cursor is a mechanism that points to one or more rows in a set of rows. The rows are retrieved from a table or in a result set that is returned by a stored procedure. Your application program can use a cursor to retrieve rows from a table.

**About this task**

Cursors bound with cursor stability that are used in block fetch operations are particularly vulnerable to reading data that has already changed. In a block fetch, database access prefetches rows ahead of the row retrieval controlled by the application. During that time the cursor might close, and the locks might be released, before the application receives the data. Thus, it is possible for the application to fetch a row of
values that no longer exists, or to miss a recently inserted row. In many cases, that is acceptable; a case for which it is not acceptable is said to require data currency.

If your application requires data currency for a cursor, you need to prevent block fetching for the data to which it points. To prevent block fetching for a distributed cursor, declare the cursor with the FOR UPDATE clause.

**Types of cursors**

You can declare row-positioned or rowset-positioned cursors in a number of ways. These cursors can be scrollable or not scrollable, held or not held, or returnable or not returnable.

In addition, you can declare a returnable cursor in a stored procedure by including the WITH RETURN clause; the cursor can return result sets to a caller of the stored procedure.

**Scrollable and non-scrollable cursors:**

When you declare a cursor, you tell Db2 whether you want the cursor to be scrollable or non-scrollable by including or omitting the SCROLL clause. This clause determines whether the cursor moves sequentially forward through the result table or can move randomly through the result table.

**Using a non-scrollable cursor:**

The simplest type of cursor is a non-scrollable cursor. A non-scrollable cursor can be either row-positioned or rowset-positioned. A row-positioned non-scrollable cursor moves forward through its result table one row at a time. Similarly, a rowset-positioned non-scrollable cursor moves forward through its result table one rowset at a time.

A non-scrollable cursor always moves sequentially forward in the result table. When the application opens the cursor, the cursor is positioned before the first row (or first rowset) in the result table. When the application executes the first FETCH, the cursor is positioned on the first row (or first rowset). When the application executes subsequent FETCH statements, the cursor moves one row ahead (or one rowset ahead) for each FETCH. After each FETCH statement, the cursor is positioned on the row (or rowset) that was fetched.

After the application executes a positioned UPDATE or positioned DELETE statement, the cursor stays at the current row (or rowset) of the result table. You cannot retrieve rows (or rowsets) backward or move to a specific position in a result table with a non-scrollable cursor.

**Using a scrollable cursor:**

To make a cursor scrollable, you declare it as scrollable. A scrollable cursor can be either row-positioned or rowset-positioned. To use a scrollable cursor, you execute FETCH statements that indicate where you want to position the cursor.

If you want to order the rows of the cursor’s result set, and you also want the cursor to be updatable, you need to declare the cursor as scrollable, even if you use it only to retrieve rows (or rowsets) sequentially. You can use the ORDER BY clause in the declaration of an updatable cursor only if you declare the cursor as scrollable.

**Declaring a scrollable cursor:**

To indicate that a cursor is scrollable, you declare it with the SCROLL keyword. The following examples show a characteristic of scrollable cursors: the sensitivity.

The following figure shows a declaration for an insensitive scrollable cursor.

```
EXEC SQL DECLARE C1 INSENSITIVE SCROLL CURSOR FOR
SELECT DEPTNO, DEPTNAME, MGRNO
FROM DSN8C10.DEPT
ORDER BY DEPTNO
END-EXEC.
```
Declaring a scrollable cursor with the INSENSITIVE keyword has the following effects:

- The size, the order of the rows, and the values for each row of the result table do not change after the application opens the cursor.
- Rows that are inserted into the underlying table are not added to the result table.
- The result table is read-only. Therefore, you cannot declare the cursor with the FOR UPDATE clause, and you cannot use the cursor for positioned update or delete operations.

The following figure shows a declaration for a sensitive static scrollable cursor.

```sql
EXEC SQL DECLARE C2 SENSITIVE STATIC SCROLL CURSOR FOR
SELECT DEPTNO, DEPTNAME, MGRNO
FROM DSN8C10.DEPT
ORDER BY DEPTNO
END-EXEC.
```

Declaring a cursor as SENSITIVE STATIC has the following effects:

- The size of the result table does not grow after the application opens the cursor.
- Rows that are inserted into the underlying table are not added to the result table.
- The order of the rows does not change after the application opens the cursor.
- If the cursor declaration contains an ORDER BY clause, and the columns that are in the ORDER BY clause are updated after the cursor is opened, the order of the rows in the result table does not change.
- When the application executes positioned UPDATE and DELETE statements with the cursor, those changes are visible in the result table.
- When the current value of a row no longer satisfies the SELECT statement that was used in the cursor declaration, that row is no longer visible in the result table.
- When a row of the result table is deleted from the underlying table, that row is no longer visible in the result table.
- Changes that are made to the underlying table by other cursors or other application processes can be visible in the result table, depending on whether the FETCH statements that you use with the cursor are FETCH INSENSITIVE or FETCH SENSITIVE statements.

The following figure shows a declaration for a sensitive dynamic scrollable cursor.

```sql
EXEC SQL DECLARE C2 SENSITIVE DYNAMIC SCROLL CURSOR FOR
SELECT DEPTNO, DEPTNAME, MGRNO
FROM DSN8C10.DEPT
ORDER BY DEPTNO
END-EXEC.
```

Declaring a cursor as SENSITIVE DYNAMIC has the following effects:

- The size and contents of the result table can change with every fetch.
- The base table can change while the cursor is scrolling on it. If another application process changes the data, the cursor sees the newly changed data when it is committed. If the application process of the cursor changes the data, the cursor sees the newly changed data immediately.
- The order of the rows can change after the application opens the cursor.
- If the SELECT statement of the cursor declaration contains an ORDER BY clause, and columns that are in the ORDER BY clause are updated after the cursor is opened, the order of the rows in the result table changes.
- When the application executes positioned UPDATE and DELETE statements with the cursor, those changes are visible. In addition, when the application executes insert, update, or delete operations (within the application but outside the cursor), those changes are visible.
- All committed inserts, updates, and deletes by other application processes are visible.
Because the FETCH statement executes against the base table, the cursor needs no temporary result table. When you define a cursor as SENSITIVE DYNAMIC, you cannot specify the INSENSITIVE keyword in a FETCH statement for that cursor.

Visibility of changes to a result table:

Whether a cursor can view its own changes or the changes that are made to the data by other processes or cursors depends on how the cursor is declared, and the updatability of the cursor. Visibility also depends on the type of fetch operation that is executed with the cursor. The following table summarizes the visibility of changes to a result table for each type of cursor.

<table>
<thead>
<tr>
<th>Declared cursor type</th>
<th>Cursor is updatable or read-only?</th>
<th>Changes by the cursor are visible in the result table?</th>
<th>Changes by other cursors or processes are visible to the result table?</th>
</tr>
</thead>
<tbody>
<tr>
<td>NO SCROLL (result table is materialized)</td>
<td>Read-only [1] on page 404</td>
<td>Not applicable</td>
<td>No</td>
</tr>
<tr>
<td>NO SCROLL (result table is not materialized)</td>
<td>Updatable [2] on page 404</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>INSENSITIVE SCROLL</td>
<td>Read-only [4] on page 404</td>
<td>Not applicable</td>
<td>No</td>
</tr>
</tbody>
</table>

Notes:

1. The content of the SELECT statement of the cursor makes the cursor implicitly read-only.
2. The cursor is updatable only if FOR READ ONLY or FOR FETCH ONLY is not specified as part of the SELECT statement of the cursor, and there is nothing in the content of the SELECT statement makes the cursor implicitly read-only.
3. If INSENSITIVE is specified on FETCH, only changes made by the same cursor are visible, assuming that the rows being fetched have not already been read by a SENSITIVE FETCH on the same cursor.
4. An INSENSITIVE cursor is read-only if an updatability clause is not specified.
5. The sensitivity clause in a FETCH statement affects the visibility of others’ changes as follows:
   - For FETCH INSENSITIVE: Only positioned updates and deletes that are made by the same cursor are visible.
   - For FETCH SENSITIVE: All updates and deletes are visible.
6. Positioned updates and deletes are disallowed if the values of the selected columns do not match the current values of the columns in the base table, even if the row satisfies the predicate of the SELECT statement of the cursor.
7. All updates and deletes that are made by this cursor, and committed changes that are made by other processes are visible on subsequent FETCH statements. Inserts that are made by this process are also be visible as the result table is scrolled. Inserts by other processes into the base tables underlying the result table are visible after they are committed.

Related concepts
FETCH statement interaction between row and rowset positioning
When you declare a cursor with the WITH ROWSET POSITIONING clause, you can intermix rowpositioned FETCH statements with rowset-positioned FETCH statements.

Comparison of scorable cursors
Whether a scorable cursor can view the changes that are made to the data by other processes or cursors depends on how the cursor is declared. It also depends on the type of fetch operation that is executed.

**Held and non-held cursors**
A held cursor does not close after a commit operation. A cursor that is not held closes after a commit operation. You specify whether you want a cursor to be held or not held by including or omitting the WITH HOLD clause when you declare the cursor.

After a commit operation, the position of a held cursor depends on its type:

- A non-scorable cursor that is held is positioned after the last retrieved row and before the next logical row. The next row can be returned from the result table with a FETCH NEXT statement.
- A static scorable cursor that is held is positioned on the last retrieved row. The last retrieved row can be returned from the result table with a FETCH CURRENT statement.
- A dynamic scorable cursor that is held is positioned after the last retrieved row and before the next logical row. Use a FETCH statement to reposition the cursor to retrieve the desired row or rowset. Db2 returns SQLCODE +231 for a FETCH statement that specifies the CURRENT keyword for a single-row fetch.

A held cursor can close when:

- You issue a CLOSE cursor, ROLLBACK, or CONNECT statement
- You issue a CAF CLOSE function call or an RRSAF TERMINATE THREAD function call
- The application program terminates.

If the program abnormally terminates, the cursor position is lost. To prepare for restart, your program must reposition the cursor.

The following restrictions apply to cursors that are declared WITH HOLD:

- Do not use DECLARE CURSOR WITH HOLD with the new user signon from a Db2 attachment facility, because all open cursors are closed.
- Do not declare a WITH HOLD cursor in a thread that might become inactive. If you do, its locks are held indefinitely.

**IMS**
You **cannot** use DECLARE CURSOR...WITH HOLD in message processing programs (MPP) and message-driven batch message processing (BMP). Each message is a new user for Db2; whether or not you declare them using WITH HOLD, no cursors continue for new users. You can use WITH HOLD in non-message-driven BMP and DL/I batch programs.

**CICS**
In CICS applications, you can use DECLARE CURSOR...WITH HOLD to indicate that a cursor should not close at a commit or sync point. However, SYNCPOINT ROLLBACK closes all cursors, and end-of-task (EOT) closes all cursors before Db2 reuses or terminates the thread. Because pseudo-conversational transactions usually have multiple EXEC CICS RETURN statements and thus span multiple EOTs, the scope of a held cursor is limited. Across EOTs, you must reopen and reposition a cursor declared WITH HOLD, as if you had not specified WITH HOLD.

You should always close cursors that you no longer need. If you let Db2 close a CICS attachment cursor, the cursor might not close until the CICS attachment facility reuses or terminates the thread.

If the CICS application is using a protected entry thread, this thread will continue to hold resources, even when the task that has used these resources ends. These resources will not be released until the protected thread terminates.
The following cursor declaration causes the cursor to maintain its position in the DSN8C10.EMP table after a commit point:

```sql
EXEC SQL
DECLARE EMPLUPDT CURSOR WITH HOLD FOR
SELECT EMPNO, LASTNAME, PHONENO, JOB, SALARY, WORKDEPT
FROM DSN8C10.EMP
WHERE WORKDEPT < 'D11'
ORDER BY EMPNO
END-EXEC.
```

**Accessing data by using a row-positioned cursor**

A row-positioned cursor is a cursor that points to a single row and retrieves at most a single row at a time from the result table. You can specify a fetch request to specify which rows to retrieve, relative to the current cursor position.

**Procedure**

To access data by using a row-positioned cursor:

1. Execute a DECLARE CURSOR statement to define the result table on which the cursor operates. See “Declaring a row cursor” on page 406.
2. Execute an OPEN CURSOR to make the cursor available to the application. See “Opening a row cursor” on page 408.
3. Specify what the program is to do when all rows have been retrieved. See “Specifying the action that the row cursor is to take when it reaches the end of the data” on page 408.
4. Execute multiple SQL statements to retrieve data from the table or modify selected rows of the table. See “Executing SQL statements by using a row cursor” on page 409.
5. Execute a CLOSE CURSOR statement to make the cursor unavailable to the application. See “Closing a row cursor” on page 410.

**Results**

Your program can have several cursors, each of which performs the previous steps.

**Declaring a row cursor**

Before you can use a row-positioned cursor to retrieve rows, you must declare the cursor. When you declare a cursor, you identify a set of rows that are to be accessed with the cursor.

**Procedure**

To declare a row cursor, issue a DECLARE CURSOR statement.

The DECLARE CURSOR statement names a cursor and specifies a SELECT statement. The SELECT statement defines the criteria for the rows that are to make up the result table.

The following example shows a simple form of the DECLARE CURSOR statement:

```sql
EXEC SQL
DECLARE C1 CURSOR FOR
SELECT EMPNO, FIRSTNME, MIDINIT, LASTNAME, SALARY
FROM DSN8C10.EMP
END-EXEC.
```

You can use this cursor to list select information about employees.

More complicated cursors might include WHERE clauses or joins of several tables. For example, suppose that you want to use a cursor to list employees who work on a certain project. Declare a cursor like this to identify those employees:

```sql
EXEC SQL
DECLARE C2 CURSOR FOR
```
**Declaring cursors for tables that use multilevel security**

You can declare a cursor that retrieves rows from a table that uses multilevel security with row-level granularity. However, the result table for the cursor contains only those rows that have a security label value that is equivalent to or dominated by the security label value of your ID.

**Updating a column**

You can update columns in the rows that you retrieve. Updating a row after you use a cursor to retrieve it is called a positioned update. If you intend to perform any positioned updates on the identified table, include the FOR UPDATE clause. The FOR UPDATE clause has two forms:

- The first form is FOR UPDATE OF column-list. Use this form when you know in advance which columns you need to update.
- The second form is FOR UPDATE, with no column list. Use this form when you might use the cursor to update any of the columns of the table.

For example, you can use this cursor to update only the SALARY column of the employee table:

```sql
EXEC SQL
DECLARE C1 CURSOR FOR
SELECT EMPNO, FIRSTNME, MIDINIT, LASTNAME, SALARY
FROM DSN8C10.EMP X
WHERE EXISTS
(SELECT *
FROM DSN8C10.PROJ Y
WHERE X.EMPNO=Y.RESPEMP
AND Y.PROJNO=:GOODPROJ)
FOR UPDATE OF SALARY;
```

If you might use the cursor to update any column of the employee table, define the cursor like this:

```sql
EXEC SQL
DECLARE C1 CURSOR FOR
SELECT EMPNO, FIRSTNME, MIDINIT, LASTNAME, SALARY
FROM DSN8C10.EMP X
WHERE EXISTS
(SELECT *
FROM DSN8C10.PROJ Y
WHERE X.EMPNO=Y.RESPEMP
AND Y.PROJNO=:GOODPROJ)
FOR UPDATE;
```

Db2 must do more processing when you use the FOR UPDATE clause without a column list than when you use the FOR UPDATE clause with a column list. Therefore, if you intend to update only a few columns of a table, your program can run more efficiently if you include a column list.

The precompiler options NOFOR and STDSQL affect the use of the FOR UPDATE clause in static SQL statements. If you do not specify the FOR UPDATE clause in a DECLARE CURSOR statement, and you do not specify the STDSQL(YES) option or the NOFOR precompiler options, you receive an error if you execute a positioned UPDATE statement.

You can update a column of the identified table even though it is not part of the result table. In this case, you do not need to name the column in the SELECT statement. When the cursor retrieves a row (using FETCH) that contains a column value you want to update, you can use UPDATE ... WHERE CURRENT OF to identify the row that is to be updated.

**Read-only result table**

Some result tables cannot be updated—for example, the result of joining two or more tables.

**Related concepts**

Multilevel security (Managing Security)
Related reference
Descriptions of SQL processing options
You can specify any SQL processing options regardless of whether you use the Db2 precompiler or the Db2 coprocessor. However, the Db2 coprocessor might ignore certain options because host language compiler options exist that provide the same information.

DECLARE CURSOR (Db2 SQL)
select-statement (Db2 SQL)

Opening a row cursor
After you declare a row cursor, you need to tell Db2 that you are ready to process the first row of the result table. This action is called opening the cursor.

About this task
To open a row cursor, execute the OPEN statement in your program. Db2 then uses the SELECT statement within DECLARE CURSOR to identify a set of rows. If you use host variables in the search condition of that SELECT statement, Db2 uses the current value of the variables to select the rows. The result table that satisfies the search condition might contain zero, one, or many rows. An example of an OPEN statement is:

```
EXEC SQL
OPEN C1
END-EXEC.
```

If you use the CURRENT DATE, CURRENT TIME, or CURRENT TIMESTAMP special registers in a cursor, Db2 determines the values in those special registers only when it opens the cursor. Db2 uses the values that it obtained at OPEN time for all subsequent FETCH statements.

Two factors that influence the amount of time that Db2 requires to process the OPEN statement are:

- Whether Db2 must perform any sorts before it can retrieve rows
- Whether Db2 uses parallelism to process the SELECT statement of the cursor

Specifying the action that the row cursor is to take when it reaches the end of the data
Your program must be coded to recognize and handle an end-of-data condition whenever you use a row cursor to fetch a row.

About this task
To determine whether the program has retrieved the last row of data, test the SQLCODE field for a value of 100 or the SQLSTATE field for a value of '02000'. These codes occur when a FETCH statement has retrieved the last row in the result table and your program issues a subsequent FETCH. For example:

```
IF SQLCODE = 100 GO TO DATA-NOT-FOUND.
```

An alternative to this technique is to code the WHENEVER NOT FOUND statement. The WHENEVER NOT FOUND statement causes your program to branch to another part that then issues a CLOSE statement. For example, to branch to label DATA-NOT-FOUND when the FETCH statement does not return a row, use this statement:

```
EXEC SQL
WHENEVER NOT FOUND GO TO DATA-NOT-FOUND
END-EXEC.
```

For more information about the WHENEVER NOT FOUND statement, see “Checking the execution of SQL statements” on page 527.
**Executing SQL statements by using a row cursor**

You can use row cursors to execute FETCH statements, positioned UPDATE statements, and positioned DELETE statements.

**About this task**

Execute a FETCH statement for one of the following purposes:

- To copy data from a row of the result table into one or more host variables
- To position the cursor before you perform a positioned update or positioned delete operation

The following example shows a FETCH statement that retrieves selected columns from the employee table:

```sql
EXEC SQL
  FETCH C1 INTO
END-EXEC.
```

The SELECT statement within DECLARE CURSOR statement identifies the result table from which you fetch rows, but Db2 does not retrieve any data until your application program executes a FETCH statement.

When your program executes the FETCH statement, Db2 positions the cursor on a row in the result table. That row is called the current row. Db2 then copies the current row contents into the program host variables that you specify on the INTO clause of FETCH. This sequence repeats each time you issue FETCH, until you process all rows in the result table.

The row that Db2 points to when you execute a FETCH statement depends on whether the cursor is declared as a scrollable or non-scrollable.

When you query a remote subsystem with FETCH, consider using block fetch for better performance. Block fetch processes rows ahead of the current row. You cannot use a block fetch when you perform a positioned update or delete operation.

After your program has executed a FETCH statement to retrieve the current row, you can use a positioned UPDATE statement to modify the data in that row. An example of a positioned UPDATE statement is:

```sql
EXEC SQL
  UPDATE DSN8C10.EMP
    SET SALARY = 50000
  WHERE CURRENT OF C1
END-EXEC.
```

A positioned UPDATE statement updates the row on which the cursor is positioned.

A positioned UPDATE statement is subject to these restrictions:

- You cannot update a row if your update violates any unique, check, or referential constraints.
- You cannot use an UPDATE statement to modify the rows of a created temporary table. However, you can use an UPDATE statement to modify the rows of a declared temporary table.
- If the right side of the SET clause in the UPDATE statement contains a fullselect, that fullselect cannot include a correlated name for a table that is being updated.
- You cannot use an SQL data change statement in the FROM clause of a SELECT statement that defines a cursor that is used in a positioned UPDATE statement.
- A positioned UPDATE statement will fail if the value of the security label column of the row where the cursor is positioned is not equivalent to the security label value of your user id. If your user id has write down privilege, a positioned UPDATE statement will fail if the value of the security label column of the row where the cursor is positioned does not dominate the security label value of your user id.

After your program has executed a FETCH statement to retrieve the current row, you can use a positioned DELETE statement to delete that row. A example of a positioned DELETE statement looks like this:
A positioned DELETE statement deletes the row on which the cursor is positioned.

A positioned DELETE statement is subject to these restrictions:

- You cannot use a DELETE statement with a cursor to delete rows from a created temporary table. However, you can use a DELETE statement with a cursor to delete rows from a declared temporary table.
- After you have deleted a row, you cannot update or delete another row using that cursor until you execute a FETCH statement to position the cursor on another row.
- You cannot delete a row if doing so violates any referential constraints.
- You cannot use an SQL data change statement in the FROM clause of a SELECT statement that defines a cursor that is used in a positioned DELETE statement.
- A positioned DELETE statement will fail if the value of the security label column of the row where the cursor is positioned is not equivalent to the security label value of your user id. If your user id has write down privilege, a positioned DELETE statement will fail if the value of the security label column of the row where the cursor is positioned does not dominate the security label value of your user id.

**Closing a row cursor**

Close a row cursor when it finishes processing rows if you want to free the resources or if you want to use the cursor again. Otherwise, you can let Db2 automatically close the cursor when the current transaction terminates or when your program terminates.

**About this task**

To free the resources that are held by the cursor, close the cursor explicitly by issuing the CLOSE statement.

If you want to use the rowset cursor again, reopen it.

**Procedure**

Issue a CLOSE statement.

An example of a CLOSE statement looks like this:

```
EXEC SQL
CLOSE C1
END-EXEC.
```

**Accessing data by using a rowset-positioned cursor**

A rowset-positioned cursor is a cursor that can return one or more rows for a single fetch operation. The cursor is positioned on the set of rows that are to be fetched.

**Procedure**

To access data by using a rowset-positioned cursor:

1. Execute a DECLARE CURSOR statement to define the result table on which the cursor operates. See “Declaring a rowset cursor” on page 411.
2. Execute an OPEN CURSOR to make the cursor available to the application. See “Opening a rowset cursor” on page 411.
3. Specify what the program is to do when all rows have been retrieved. See “Specifying the action that the rowset cursor is to take when it reaches the end of the data” on page 411.
4. Execute multiple SQL statements to retrieve data from the table or modify selected rows of the table. See “Executing SQL statements by using a rowset cursor” on page 412.

5. Execute a CLOSE CURSOR statement to make the cursor unavailable to the application. See “Closing a rowset cursor” on page 415.

**Results**

Your program can have several cursors, each of which performs the previous steps.

*Declaring a rowset cursor*

Before you can use a rowset-positioned cursor to retrieve rows, you must declare a cursor that is enabled to fetch rowsets. When you declare a cursor, you identify a set of rows that are to be accessed with the cursor.

**About this task**

For restrictions that apply to rowset-positioned cursors and row-positioned cursors, see “Declaring a row cursor” on page 406.

**Procedure**

Use the WITH ROWSET POSITIONING clause in the DECLARE CURSOR statement.

The following example shows how to declare a rowset cursor:

```sql
EXEC SQL
   DECLARE C1 CURSOR WITH ROWSET POSITIONING FOR
       SELECT EMPNO, LASTNAME, SALARY
       FROM DSN8C10.EMP
END-EXEC.
```

*Opening a rowset cursor*

After you declare a rowset cursor, you need to tell Db2 that you are ready to process the first rowset of the result table. This action is called opening the cursor.

**About this task**

To open a rowset cursor, execute the OPEN statement in your program. Db2 then uses the SELECT statement within DECLARE CURSOR to identify the rows in the result table. For more information about the OPEN CURSOR process, see “Opening a row cursor” on page 408.

*Specifying the action that the rowset cursor is to take when it reaches the end of the data*

Your program must be coded to recognize and handle an end-of-data condition whenever you use a rowset cursor to fetch rows.

**About this task**

To determine whether the program has retrieved the last row of data in the result table, test the SQLCODE field for a value of +100 or the SQLSTATE field for a value of '02000'. With a rowset cursor, these codes occur when a FETCH statement retrieves the last row in the result table. However, when the last row has been retrieved, the program must still process the rows in the last rowset through that last row. For an example of end-of-data processing for a rowset cursor, see “Examples of fetching rows by using cursors” on page 428.

To determine the number of retrieved rows, use either of the following values:

- The contents of the SQLERRD(3) field in the SQLCA
The contents of the ROW_COUNT item of GET DIAGNOSTICS

For information about GET DIAGNOSTICS, see “Checking the execution of SQL statements by using the GET DIAGNOSTICS statement ” on page 534.

If you declare the cursor as dynamic scrollable, and SQLCODE has the value +100, you can continue with a FETCH statement until no more rows are retrieved. Additional fetches might retrieve more rows because a dynamic scrollable cursor is sensitive to updates by other application processes. For information about dynamic cursors, see “Types of cursors” on page 402.

**Executing SQL statements by using a rowset cursor**

You can use rowset cursors to execute multiple-row FETCH statements, positioned UPDATE statements, and positioned DELETE statements.

**About this task**

You can execute these static SQL statements when you use a rowset cursor:

- A multiple-row FETCH statement that copies a rowset of column values into either of the following data areas:
  - Host-variable arrays that are declared in your program
  - Dynamically-allocated arrays whose storage addresses are put into an SQL descriptor area (SQLDA), along with the attributes of the columns that are to be retrieved

- After either form of the multiple-row FETCH statement, you can issue:
  - A positioned UPDATE statement on the current rowset
  - A positioned DELETE statement on the current rowset

You must use the WITH ROWSET POSITIONING clause of the DECLARE CURSOR statement if you plan to use a rowset-positioned FETCH statement.

The following example shows a FETCH statement that retrieves 20 rows into host-variable arrays that are declared in your program:

```sql
EXEC SQL
  FETCH NEXT ROWSET FROM C1
  FOR 20 ROWS
END-EXEC.
```

When your program executes a FETCH statement with the ROWSET keyword, the cursor is positioned on a rowset in the result table. That rowset is called the *current rowset*. The dimension of each of the host-variable arrays must be greater than or equal to the number of rows to be retrieved.

Suppose that you want to dynamically allocate the storage needed for the arrays of column values that are to be retrieved from the employee table. You must:

1. Declare an SQLDA structure and the variables that reference the SQLDA.
2. Dynamically allocate the SQLDA and the arrays needed for the column values.
3. Set the fields in the SQLDA for the column values to be retrieved.
4. Open the cursor.
5. Fetch the rows.

You must first declare the SQLDA structure. The following SQL INCLUDE statement requests a standard SQLDA declaration:

```sql
EXEC SQL INCLUDE SQLDA;
```
Your program must also declare variables that reference the SQLDA structure, the SQLVAR structure within the SQLDA, and the DECLEN structure for the precision and scale if you are retrieving a DECIMAL column. For C programs, the code looks like this:

```c
struct sqlda *sqldaptr;
struct sqlvar *varptr;
struct DECLEN {
    unsigned char precision;
    unsigned char scale;
};
```

Before you can set the fields in the SQLDA for the column values to be retrieved, you must dynamically allocate storage for the SQLDA structure. For C programs, the code looks like this:

```c
sqldaptr = (struct sqlda *) malloc (3 * 44 + 16);
```

The size of the SQLDA is SQLN * 44 + 16, where the value of the SQLN field is the number of output columns.

You must set the fields in the SQLDA structure for your FETCH statement. Suppose you want to retrieve the columns EMPNO, LASTNAME, and SALARY. The C code to set the SQLDA fields for these columns looks like this:

```c
strcpy(sqldaptr->sqldaid,"SQLDA");
sqldaptr->sqldabc = 148;  /* number bytes of storage allocated for the SQLDA */
sqldaptr->sqln = 3;       /* number of SQLVAR occurrences */
sqldaptr->sqlid = 3;
varptr = (struct sqlvar *) (&(sqldaptr->sqlvar[0]));     /* Point to first SQLVAR */
varptr->sqllen = 6;
varptr->sqldata = (char *) hva1;
varptr->sqlind = (short *) inda1;
memcpy(varptr->sqlname.data, "\00\00\00\00\00\00\01\14",varptr->sqlname.length);
varptr = (struct sqlvar *) (&(sqldaptr->sqlvar[0]) + 1); /* Point to next SQLVAR */
varptr->sqllen = 15;
varptr->sqldata = (char *) hva2;
varptr->sqlind = (short *) inda2;
varptr->sqlname.length = 8;
memcpy(varptr->sqlname.data, "\00\00\00\00\00\00\01\14",varptr->sqlname.length);
varptr = (struct sqlvar *) (&(sqldaptr->sqlvar[0]) + 2); /* Point to next SQLVAR */
varptr->sqllen = 485; /* data type DECIMAL(9,2) */
((struct DECLEN *) &(varptr->sqllen))->precision = 9;
((struct DECLEN *) &(varptr->sqllen))->scale = 2;
varptr->sqldata = (char *) hva3;
varptr->sqlind = (short *) inda3;
varptr->sqlname.length = 8;
memcpy(varptr->sqlname.data, "\00\00\00\00\00\00\01\14",varptr->sqlname.length);
```

The SQLDA structure has these fields:

- SQLDABC indicates the number of bytes of storage that are allocated for the SQLDA. The storage includes a 16-byte header and 44 bytes for each SQLVAR field. The value is SQLN x 44 + 16, or 148 for this example.
- SQLN is the number of SQLVAR occurrences (or the number of output columns).
- SQLD is the number of variables in the SQLDA that are used by Db2 when processing the FETCH statement.
- Each SQLVAR occurrence describes a host-variable array or buffer into which the values for a column in the result table are to be returned. Within each SQLVAR:
  - SQLTYPE indicates the data type of the column.
  - SQLLEN indicates the length of the column. If the data type is DECIMAL, this field has two parts: the PRECISION and the SCALE.
  - SQLDATA points to the first element of the array for the column values. For this example, assume that your program allocates the dynamic variable arrays hva1, hva2, and hva3, and their indicator arrays inda1, inda2, and inda3.
- SQLIND points to the first element of the array of indicator values for the column. If SQLTYPE is an odd number, this attribute is required. (If SQLTYPE is an odd number, null values are allowed for the column.)

- SQLNAME has two parts: the LENGTH and the DATA. The LENGTH is 8. The first two bytes of the DATA field is X'0000'. Bytes 5 and 6 of the DATA field are a flag indicating whether the variable is an array or a FOR n ROWS value. Bytes 7 and 8 are a two-byte binary integer representation of the dimension of the array.

You can open the cursor only after all of the fields have been set in the output SQLDA:

```
EXEC SQL OPEN C1;
```

After the OPEN statement, the program fetches the next rowset:

```
EXEC SQL
  FETCH NEXT ROWSET FROM C1
  FOR 20 ROWS
  USING DESCRIPTOR :*sqladptr;
```

The USING clause of the FETCH statement names the SQLDA that describes the columns that are to be retrieved.

After your program executes a FETCH statement to establish the current rowset, you can use a positioned UPDATE statement with either of the following clauses:

- Use WHERE CURRENT OF to modify all of the rows in the current rowset
- Use FOR ROW n OF ROWSET to modify row n in the current rowset

An example of a positioned UPDATE statement that uses the WHERE CURRENT OF clause is:

```
EXEC SQL
  UPDATE DSN8C10.EMP
  SET SALARY = 50000
  WHERE CURRENT OF C1
END-EXEC.
```

When the UPDATE statement is executed, the cursor must be positioned on a row or rowset of the result table. If the cursor is positioned on a row, that row is updated. If the cursor is positioned on a rowset, all of the rows in the rowset are updated.

An example of a positioned UPDATE statement that uses the FOR ROW n OF ROWSET clause is:

```
EXEC SQL
  UPDATE DSN8C10.EMP
  SET SALARY = 50000
  FOR CURSOR C1 FOR ROW 5 OF ROWSET
END-EXEC.
```

When the UPDATE statement is executed, the cursor must be positioned on a rowset of the result table. The specified row (in the example, row 5) of the current rowset is updated.

After your program executes a FETCH statement to establish the current rowset, you can use a positioned DELETE statement with either of the following clauses:

- Use WHERE CURRENT OF to delete all of the rows in the current rowset
- Use FOR ROW n OF ROWSET to delete row n in the current rowset

An example of a positioned DELETE statement that uses the WHERE CURRENT OF clause is:

```
EXEC SQL
  DELETE FROM DSN8C10.EMP
  WHERE CURRENT OF C1
END-EXEC.
```

When the DELETE statement is executed, the cursor must be positioned on a row or rowset of the result table. If the cursor is positioned on a row, that row is deleted, and the cursor is positioned before the next
row of its result table. If the cursor is positioned on a rowset, all of the rows in the rowset are deleted, and the cursor is positioned before the next rowset of its result table.

An example of a positioned DELETE statement that uses the FOR ROW n OF ROWSET clause is:

```sql
EXEC SQL
DELETE FROM DSN8C10.EMP
  FOR CURSOR C1 FOR ROW 5 OF ROWSET
END-EXEC.
```

When the DELETE statement is executed, the cursor must be positioned on a rowset of the result table. The specified row of the current rowset is deleted, and the cursor remains positioned on that rowset. The deleted row (in the example, row 5 of the rowset) cannot be retrieved or updated.

**Related tasks**

- Including dynamic SQL in your program
  Dynamic SQL is prepared and executed while the program is running.

- Executing SQL statements by using a row cursor
  You can use row cursors to execute FETCH statements, positioned UPDATE statements, and positioned DELETE statements.

**Related reference**

- SQL descriptor area (SQLDA) (Db2 SQL)

  *Specifying the number of rows in a rowset*
  If you do not explicitly specify the number of rows in a rowset, Db2 implicitly determines the number of rows based on the last fetch request.

**About this task**

To explicitly set the size of a rowset, use the FOR n ROWS clause in the FETCH statement. If a FETCH statement specifies the ROWSET keyword, and not the FOR n ROWS clause, the size of the rowset is implicitly set to the size of the rowset that was most recently specified in a prior FETCH statement. If a prior FETCH statement did not specify the FOR n ROWS clause or the ROWSET keyword, the size of the current rowset is implicitly set to 1. For examples of rowset positioning, see Table 78 on page 427.

**Closing a rowset cursor**

Close a rowset cursor when it finishes processing rows if you want to free the resources or if you want to use the cursor again. Otherwise, you can let Db2 automatically close the cursor when the current transaction terminates or when your program terminates.

**About this task**

To free the resources held by the cursor, close the cursor explicitly by issuing the CLOSE statement. If you want to use the rowset cursor again, reopen it.

**Procedure**

Issue a CLOSE statement.
Retrieving rows by using a scrollable cursor

A **scrollable cursor** is cursor that can be moved in both a forward and a backward direction. Scrollable cursors can be either row-positioned or rowset-positioned.

**Procedure**

When you open any cursor, the cursor is positioned before the first row of the result table. You move a scrollable cursor around in the result table by specifying a *fetch orientation* keyword in a FETCH statement.

A fetch orientation keyword indicates the absolute or relative position of the cursor when the FETCH statement is executed. The following table lists the fetch orientation keywords that you can specify and their meanings. These keywords apply to both row-positioned scrollable cursors and rowset-positioned scrollable cursors.

<table>
<thead>
<tr>
<th>Keyword in FETCH statement</th>
<th>Cursor position when FETCH is executed</th>
</tr>
</thead>
<tbody>
<tr>
<td>BEFORE</td>
<td>Before the first row</td>
</tr>
<tr>
<td>FIRST or ABSOLUTE +1</td>
<td>On the first row</td>
</tr>
<tr>
<td>LAST or ABSOLUTE -1</td>
<td>On the last row</td>
</tr>
<tr>
<td>AFTER</td>
<td>After the last row</td>
</tr>
<tr>
<td>ABSOLUTE</td>
<td>On an absolute row number, from before the first row forward or from after the last row backward</td>
</tr>
<tr>
<td>RELATIVE</td>
<td>On the row that is forward or backward a relative number of rows from the current row</td>
</tr>
<tr>
<td>CURRENT</td>
<td>On the current row</td>
</tr>
<tr>
<td>PRIOR or RELATIVE -1</td>
<td>On the previous row</td>
</tr>
<tr>
<td>NEXT</td>
<td>On the next row (default)</td>
</tr>
</tbody>
</table>

**Table notes**

- The cursor position applies to both row position and rowset position, for example, before the first row or before the first rowset.
- For more information about ABSOLUTE and RELATIVE, see FETCH (Db2 SQL)

**Example**

To use the cursor that is declared in “Types of cursors” on page 402 to fetch the fifth row of the result table, use a FETCH statement like this:

```sql
EXEC SQL FETCH ABSOLUTE +5 C1 INTO :HVDEPTNO, :DEPTNAME, :MGRNO;
```

To fetch the fifth row from the end of the result table, use this FETCH statement:

```sql
EXEC SQL FETCH ABSOLUTE -5 C1 INTO :HVDEPTNO, :DEPTNAME, :MGRNO;
```

**Related concepts**

Types of cursors

You can declare row-positioned or rowset-positioned cursors in a number of ways. These cursors can be scrollable or not scrollable, held or not held, or returnable or not returnable.

**Related reference**

FETCH (Db2 SQL)
Comparison of scrollable cursors

Whether a scrollable cursor can view the changes that are made to the data by other processes or cursors depends on how the cursor is declared. It also depends on the type of fetch operation that is executed.

When you declare a cursor as SENSITIVE STATIC, changes that other processes or cursors make to the underlying table can be visible to the result table of the cursor. Whether those changes are visible depends on whether you specify SENSITIVE or INSENSITIVE when you execute FETCH statements with the cursor. When you specify FETCH INSENSITIVE, changes that other processes or other cursors make to the underlying table are not visible in the result table. When you specify FETCH SENSITIVE, changes that other processes or cursors make to the underlying table are visible in the result table.

When you declare a cursor as SENSITIVE DYNAMIC, changes that other processes or cursors make to the underlying table are visible to the result table after the changes are committed.

The following table summarizes the sensitivity values and their effects on the result table of a scrollable cursor.

<table>
<thead>
<tr>
<th>DECLARE sensitivity</th>
<th>FETCH INSENSITIVE</th>
<th>FETCH SENSITIVE</th>
</tr>
</thead>
<tbody>
<tr>
<td>INSENSITIVE</td>
<td>No changes to the underlying table are visible in the result table. Positioned UPDATE and DELETE statements using the cursor are not allowed.</td>
<td>Not valid.</td>
</tr>
<tr>
<td>SENSITIVE STATIC</td>
<td>Only positioned updates and deletes that are made by the cursor are visible in the result table.</td>
<td>All updates and deletes are visible in the result table. Inserts made by other processes are not visible in the result table.</td>
</tr>
<tr>
<td>SENSITIVE DYNAMIC</td>
<td>Not valid.</td>
<td>All committed changes are visible in the result table, including updates, deletes, inserts, and changes in the order of the rows.</td>
</tr>
</tbody>
</table>

Scrolling through a table in any direction

Use a scrollable cursor to move through the table in both a forward and a backward direction.

About this task

**Question:** How can I fetch rows from a table in any direction?

**Answer:** Declare your cursor as scrollable. When you select rows from the table, you can use the various forms of the FETCH statement to move to an absolute row number, move ahead or back a certain number of rows, to the first or last row, before the first row or after the last row, forward, or backward. You can use any combination of these FETCH statements to change direction repeatedly.

You can use code like the following example to move forward in the department table by 10 records, backward five records, and forward again by three records:

```
/*******************/
/* Declare host variables */
/*******************/
EXEC SQL BEGIN DECLARE SECTION;
char[37] hv_deptname;
EXEC SQL END DECLARE SECTION;
/*******************/
/* Declare scrollable cursor to retrieve department names */
/*******************/
EXEC SQL DECLARE C1 SCROLL CURSOR FOR
    SELECT DEPTNAME FROM DSN8C10.DEPT;
:
/*******************/
/* Open the cursor and position it before the start of */
```
Determining the number of rows in the result table for a static scrollable cursor

You can determine how many rows are in the result table of an INSENSITIVE or SENSITIVE STATIC scrollable cursor.

Procedure

To determine the number of rows in the result table for a static scrollable cursor, follow these steps:

1. Execute a FETCH statement, such as FETCH AFTER, that positions the cursor after the last row.
2. Perform one of the following actions:
   - Retrieve the values of fields SQLERRD(1) and SQLERRD(2) in the SQLCA (fields sqlerrd[0] and sqlerrd[1] for C and C++). SQLERRD(1) and SQLERRD(2) together form a double-word value that contains the number of rows in the result table.
   - Issue a GET DIAGNOSTICS statement to retrieve the value of the DB2_NUMBER_ROWS item.

Example

The following C language code demonstrates how to obtain the number of rows in a result table of a sensitive static cursor.

```c
EXEC SQL INCLUDE SQLCA;
long int rowcount;
EXEC SQL
DECLARE SENSTAT SENSITIVE STATIC SCROLL CURSOR FOR
SELECT * FROM EMP;
EXEC SQL OPEN SENSTAT;
if (SQLCODE==0) {
  EXEC SQL FETCH AFTER SENSTAT; /* Position the cursor after the end */
  /* of the result table */
  if (SQLCODE==0) {
```
Removing a delete hole or update hole

If you try to fetch data from a delete hole or an update hole, Db2 issues an SQL warning. If you try to update or to delete a delete hole or delete an update hole, Db2 issues an SQL error.

About this task

You can remove a delete hole only by opening the scrollable cursor, setting a savepoint, executing a positioned DELETE statement with the scrollable cursor, and rolling back to the savepoint.

You can convert an update hole back to a result table row by updating the row in the base table, as shown in the following figure. You can update the base table with a searched UPDATE statement in the same application process, or a searched or positioned UPDATE statement in another application process. After you update the base table, if the row qualifies for the result table, the update hole disappears.

A hole becomes visible to a cursor when a cursor operation returns a non-zero SQLCODE. The point at which a hole becomes visible depends on the following factors:

- Whether the scrollable cursor creates the hole
- Whether the FETCH statement is FETCH SENSITIVE or FETCH INSENSITIVE

If the scrollable cursor creates the hole, the hole is visible when you execute a FETCH statement for the row that contains the hole. The FETCH statement can be FETCH INSENSITIVE or FETCH SENSITIVE.

If an update or delete operation outside the scrollable cursor creates the hole, the hole is visible at the following times:

- If you execute a FETCH SENSITIVE statement for the row that contains the hole, the hole is visible when you execute the FETCH statement.
• If you execute a FETCH INSENSITIVE statement, the hole is not visible when you execute the FETCH statement. Db2 returns the row as it was before the update or delete operation occurred. However, if you follow the FETCH INSENSITIVE statement with a positioned UPDATE or DELETE statement, the hole becomes visible.

_Holes in the result table of a scrollable cursor_

A hole in the result table means that the result table does not shrink to fill the space of deleted rows. It also does not shrink to fill the space of rows that have been updated and no longer satisfy the search condition. You cannot access a delete or update hole. However, you can remove holes in specific situations.

In some situations, you might not be able to fetch a row from the result table of a scrollable cursor, depending on how the cursor is declared:

• Scrollable cursors that are declared as INSENSITIVE or SENSITIVE STATIC follow a static model, which means that Db2 determines the size of the result table and the order of the rows when you open the cursor.

Deleting or updating rows after a static cursor is open can result in holes in the result table. See “Removing a delete hole or update hole” on page 419.

• Scrollable cursors that are declared as SENSITIVE DYNAMIC follow a dynamic model, which means that the size and contents of the result table, and the order of the rows, can change after you open the cursor.

A dynamic cursor scrolls directly on the base table. If the current row of the cursor is deleted or if it is updated so that it no longer satisfies the search condition, and the next cursor operation is FETCH CURRENT, then Db2 issues an SQL warning.

The following examples demonstrate how delete and update holes can occur when you use a SENSITIVE STATIC scrollable cursor.

**Creating a delete hole with a static scrollable cursor:**

Suppose that table A consists of one integer column, COL1, which has the values shown in the following figure.

![Figure 20. Values for COL1 of table A](image)

Now suppose that you declare the following SENSITIVE STATIC scrollable cursor, which you use to delete rows from A:

```sql
EXEC SQL DECLARE C3 SENSITIVE STATIC SCROLL CURSOR FOR
SELECT COL1
FROM A
FOR UPDATE OF COL1;
```

Now you execute the following SQL statements:

```sql
EXEC SQL OPEN C3;
EXEC SQL FETCH ABSOLUTE +3 C3 INTO :HVCOL1;
EXEC SQL DELETE FROM A WHERE CURRENT OF C3;
```

The positioned delete statement creates a delete hole, as shown in the following figure.
After you execute the positioned delete statement, the third row is deleted from the result table, but the result table does not shrink to fill the space that the deleted row creates.

**Creating an update hole with a static scrollable cursor**

Suppose that you declare the following SENSITIVE STATIC scrollable cursor, which you use to update rows in A:

```sql
EXEC SQL DECLARE C4 SENSITIVE STATIC SCROLL CURSOR FOR
SELECT COL1
FROM A
WHERE COL1<6;
```

Now you execute the following SQL statements:

```sql
EXEC SQL OPEN C4;
UPDATE A SET COL1=COL1+1;
```

The searched UPDATE statement creates an update hole, as shown in the following figure.

After you execute the searched UPDATE statement, the last row no longer qualifies for the result table, but the result table does not shrink to fill the space that the disqualified row creates.

**Accessing XML or LOB data quickly by using FETCH WITH CONTINUE**

Use the FETCH WITH CONTINUE statement to improve the performance of some queries that reference XML and LOB columns with unknown or very large maximum lengths.

**About this task**

FETCH WITH CONTINUE breaks XML and LOB values into manageable pieces and processes the pieces one at a time to avoid the following buffer allocation problems:

- Allocating overly large or unnecessary space for buffers. If some LOB values are shorter than the maximum length for values in a column, you can waste buffer space if you allocate enough space for the maximum length. The buffer allocation problem can be even worse for XML data because an XML column does not have a defined maximum length. If you use FETCH WITH CONTINUE, you can allocate more appropriate buffer space for the actual length of the XML and LOB values.

- Truncating very large XML and LOB data. If a very large XML or LOB value does not fit in the host variable buffer space that is provided by the application program, Db2 truncates the value. If the application
program retries this fetch with a larger buffer, two problems exist. First, when using a non-scrollable cursor, you cannot re-fetch the current row without closing, reopening, and repositioning the cursor to the row that was truncated. Second, if you do not use FETCH WITH CONTINUE, Db2 does not return the actual length of the entire value to the application program. Thus, Db2 does not know how large a buffer to reallocate. If you use FETCH WITH CONTINUE, Db2 preserves the truncated portion of the data for subsequent retrieval and returns the actual length of the entire data value so that the application can reallocate a buffer of the appropriate size.

Db2 provides two methods for using FETCH WITH CONTINUE with LOB and XML data:

- “Dynamically allocating buffers when fetching XML and LOB data” on page 422
- “Moving data through fixed-size buffers when fetching XML and LOB data” on page 423

**Dynamically allocating buffers when fetching XML and LOB data**

If you specify FETCH WITH CONTINUE, Db2 returns information about which data does not fit in the buffer. Your application can then use the information about the truncated data to allocate an appropriate target buffer and execute a fetch operation with the CURRENT CONTINUE clause to retrieve the remaining data.

**Procedure**

To use dynamic buffer allocation for LOB and XML data:

1. Use an initial FETCH WITH CONTINUE to fetch data into a pre-allocated buffer of a moderate size.
2. If the value is too large to fit in the buffer, use the length information that is returned by Db2 to allocate the appropriate amount of storage.
3. Use a single FETCH CURRENT CONTINUE statement to retrieve the remainder of the data.

**Example**

Suppose that table T1 was created with the following statement:

```sql
CREATE TABLE T1 (C1 INT, C2 CLOB(100M), C3 CLOB(32K), C4 XML);
```

A row exists in T1 where C1 contains a valid integer, C2 contains 10MB of data, C3 contains 32KB of data, and C4 contains 4MB of data.

Now, suppose that you declare CURSOR1, prepare and describe statement DYNSQLSTMT1 with descriptor sqlda, and open CURSOR1 with the following statements:

```sql
EXEC SQL DECLARE CURSOR1 CURSOR FOR DYNSQLSTMT1;
EXEC SQL PREPARE DYNSQLSTMT1 FROM 'SELECT * FROM T1';
EXEC SQL DESCRIBE DYNSQLSTMT1 INTO DESCRIPTOR :SQLDA;
EXEC SQL OPEN CURSOR1;
```

Next, suppose that you allocate moderately sized buffers (32 KB for each CLOB or XML column) and set data pointers and lengths in SQLDA. Then, you use the following FETCH WITH CONTINUE statement:

```sql
EXEC SQL FETCH WITH CONTINUE CURSOR1 INTO DESCRIPTOR :SQLDA;
```

Because C2 and C4 contain data that do not fit in the buffer, some of the data is truncated. Your application can use the information that Db2 returns to allocate large enough buffers for the remaining data and reset the data pointers and length fields in SQLDA. At that point, you can resume the fetch and complete the process with the following FETCH CURRENT CONTINUE statement and CLOSE CURSOR statement:

```sql
EXEC SQL FETCH CURRENT CONTINUE CURSOR1 INTO DESCRIPTOR :SQLDA;
EXEC SQL CLOSE CURSOR1;
```

The application needs to concatenate the two returned pieces of the data value. One technique is to move the first piece of data to the dynamically-allocated larger buffer before the FETCH CONTINUE. Set the
SQLDATA pointer in the SQLDA structure to point immediately after the last byte of this truncated value. Db2 then writes the remaining data to this location and thus completes the concatenation.

Moving data through fixed-size buffers when fetching XML and LOB data
If you use the WITH CONTINUE clause, Db2 returns information about which data does not fit in the buffer. Your application can then use repeated FETCH CURRENT CONTINUE operations to effectively "stream" large XML and LOB data through a fixed-size buffer, one piece at a time.

Procedure
To use fixed buffer allocation for LOB and XML data, perform the following steps:
1. Use an initial FETCH WITH CONTINUE to fetch data into a pre-allocated buffer of a moderate size.
2. If the value is too large to fit in the buffer, use as many FETCH CONTINUE statements as necessary to process all of the data through a fixed buffer.

After each FETCH operation, check whether a column was truncated by first examining the SQLWARN1 field in the returned SQLCA. If that field contains a 'W' value, at least one column in the returned row has been truncated. To then determine if a particular LOB or XML column was truncated, your application must compare the value that is returned in the length field with the declared length of the host variable. If a column is truncated, continue to use FETCH CONTINUE statements until all of the data has been retrieved.

After you fetch each piece of the data, move it out of the buffer to make way for the next fetch. Your application can write the pieces to an output file or reconstruct the entire data value in a buffer above the 2-GB bar.

Example
Suppose that table T1 was created with the following statement:

```
CREATE TABLE T1 (C1 INT, C2 CLOB(100M), C3 CLOB(32K), C4 XML);
```

A row exists in T1 where C2 contains 10 MB of data.

Now, suppose that you declare a 32 KB section CLOBHV:

```
EXEC SQL BEGIN DECLARE SECTION
  DECLARE CLOBHV SQL TYPE IS CLOB(32767);
EXEC SQL END DECLARE SECTION.
```

Next, suppose that you use the following statements to declare and open CURSOR1 and to FETCH WITH CONTINUE:

```
EXEC SQL DECLARE CURSOR1 CURSOR FOR SELECT C2 FROM T1;
EXEC SQL OPEN CURSOR1;
EXEC SQL FETCH WITH CONTINUE CURSOR1 INTO :CLOBHV;
```

As each piece of the data value is fetched, move it from the buffer to the output file.

Because the 10 MB value in C2 does not fit into the 32 KB buffer, some of the data is truncated. Your application can loop through the following FETCH CURRENT CONTINUE:

```
EXEC SQL FETCH CURRENT CONTINUE CURSOR1 INTO :CLOBHV;
```

After each FETCH operation, you can determine if the data was truncated by first checking if the SQLWARN1 field in the returned SQLCA contains a 'W' value. If so, then check if the length value, which is returned in CLOBHV_LENGTH, is greater than the declared length of 32767. (CLOBHV_LENGTH is declared as part of the precompiler expansion of the CLOBHV declaration.) If the value is greater, that value has been truncated and more data can be retrieved with the next FETCH CONTINUE operation.

When all of the data has moved to the output file, you can close the cursor:
Determining the attributes of a cursor by using the SQLCA

An SQL communications area (SQLCA) is an area that is set apart for communication with Db2 and consists of a collection of variables. Using the SQLCA is one way to get information about any open cursors. Alternatively, you can use the GET DIAGNOSTICS statement.

About this task

After you open a cursor, you can determine the following attributes of the cursor by checking the following SQLWARN and SQLERRD fields of the SQLCA:

**SQLWARN1**
Indicates whether the cursor is scrollable or non-scrollable.

**SQLWARN4**
Indicates whether the cursor is insensitive (I), sensitive static (S), or sensitive dynamic (D).

**SQLWARN5**
Indicates whether the cursor is read-only, readable and deletable, or readable, deletable, and updatable.

**SQLERRD(1) and SQLERRD(2)**
These two fields together contain a double-word integer that represents the number of rows in the result table of a cursor when the cursor is positioned after the last row. The cursor is positioned after the last row when the SQLCODE is 100. These fields are not set for dynamic scrollable cursors.

**SQLERRD(3)**
The number of rows in the result table when the SELECT statement of the cursor contains a data change statement.

If the OPEN statement executes with no errors or warnings, Db2 does not set SQLWARN0 when it sets SQLWARN1, SQLWARN4, or SQLWARN5.

Related reference
Description of SQLCA fields (Db2 SQL)

Determining the attributes of a cursor by using the GET DIAGNOSTICS statement

Using the GET DIAGNOSTICS statement is one way to get information about any open cursors. Alternatively, you can use the SQLCA.

About this task

After you open a cursor, you can determine the following attributes of the cursor by checking these GET DIAGNOSTICS items:

**DB2_SQL_ATTR_CURSOR_HOLD**
Indicates whether the cursor can be held open across commits (Y or N)

**DB2_SQL_ATTR_CURSOR_ROWSET**
Indicates whether the cursor can use rowset positioning (Y or N)

**DB2_SQL_ATTR_CURSOR_SCROLLABLE**
Indicates whether the cursor is scrollable (Y or N)

**DB2_SQL_ATTR_CURSOR_SENSITIVITY**
Indicates whether the cursor is insensitive or sensitive to changes that are made by other processes (I or S)
DB2_SQL_ATTR_CURSOR_TYPE

Indicates whether the cursor is forward (F) declared static (S for INSENSITIVE or SENSITIVE STATIC) or dynamic (D for SENSITIVE DYNAMIC)

For more information about the GET DIAGNOSTICS statement, see “Checking the execution of SQL statements by using the GET DIAGNOSTICS statement” on page 534.

Scrolling through previously retrieved data

To scroll backward through data, use a scrollable cursor, or use a ROWID column or identity column to retrieve data in reverse order.

Procedure

When a program retrieves data from the database, it can scroll backward through the data by using one of the following techniques:

- Use a scrollable cursor to fetch backward through data by following these steps:
  a) Declare the cursor with the SCROLL keyword.
  b) Open the cursor.
  c) Execute a FETCH statement to position the cursor at the end of the result table.
  d) In a loop, execute FETCH statements that move the cursor backward and then retrieve the data.
  e) When you have retrieved all the data, close the cursor.

For example, you can use code like the following example to retrieve department names in reverse order from table DSN8C10.DEPT:

```sql
/*****************************/
/* Declare host variables */
/*****************************/
EXEC SQL BEGIN DECLARE SECTION;
char[37] hv_deptname;
EXEC SQL END DECLARE SECTION;
/*****************************/
/* Declare scrollable cursor to retrieve department names */
/*****************************/
EXEC SQL DECLARE C1 SCROLL CURSOR FOR
  SELECT DEPTNAME FROM DSN8C10.DEPT;
⋮
/*****************************/
/* Open the cursor and position it after the end of the result table. */
/*****************************/
EXEC SQL OPEN C1;
EXEC SQL FETCH AFTER FROM C1;
/*****************************/
/* Fetch rows backward until all rows are fetched. */
/*****************************/
while(SQLCODE==0) {
  EXEC SQL FETCH PRIOR FROM C1 INTO :hv_deptname;
  ...
}
EXEC SQL CLOSE C1;
```

- If the table contains a ROWID or an identity column, retrieve the values from that column into an array. Then use the ROWID or identity column values to retrieve the rows in reverse order.

  You can use the ROWID column or identity column to rapidly retrieve the rows in reverse order. When you perform the original SELECT, you can store the ROWID or identity column value for each row you retrieve. Then, to retrieve the values in reverse order, you can execute SELECT statements with a WHERE clause that compares the ROWID or identity column value to each stored value.

  For example, suppose you add ROWID column DEPTROWID to table DSN8C10.DEPT. You can use code like the following example to select all department names, then retrieve the names in reverse order:

```sql
/*****************************/
/* Declare host variables */
/*****************************/
EXEC SQL BEGIN DECLARE SECTION;
```

SQL TYPE IS ROWID  hv_dept_rowid;
char[37] hv_deptname;
EXEC SQL END DECLARE SECTION;
/**************************
/* Declare other variables */
/***************************/
struct rowid_struct {
    short int length;
    char data[40];  /* ROWID variable structure */
}
struct rowid_struct rowid_array[200];    /* ROWIDs. Assume no more */
    /* than 200 rows will be */
    /* retrieved. */
short int i,j,n;
/**************************
/* Declare cursor to retrieve department names */
/***************************/
EXEC SQL DECLARE C1 CURSOR FOR
SELECT DEPTNAME, DEPTROWID FROM DSN8C10.DEPT;
EXEC SQL OPEN C1;
i=0;
while(SQLCODE==0) {
    EXEC SQL FETCH C1 INTO :hv_deptname, :hv_dept_rowid;
    rowid_array[i].length=hv_dept_rowid.length;
    for(j=0;j<hv_dept_rowid.length;j++)
        rowid_array[i].data[j]=hv_dept_rowid.data[j];
i++;
}
EXEC SQL CLOSE C1;
n=i-1;                /* Get the number of array elements */
/**************************
/* Use the ROWID values to retrieve the department names */
/***************************/
for(i=n;i>=0;i--) {
    hv_dept_rowid.length=rowid_array[i].length;
    for(j=0;j<hv_dept_rowid.length;j++)
        hv_dept_rowid.data[j]=rowid_array[i].data[j];
    EXEC SQL SELECT DEPTNAME INTO :hv_deptname FROM DSN8C10.DEPT
WHERE DEPTROWID=:hv_dept_rowid;
}

**Related concepts**

**Row ID values (Db2 SQL)**

**Identity columns**

An identity column contains a unique numeric value for each row in the table. Db2 can automatically generate sequential numeric values for this column as rows are inserted into the table. Thus, identity columns are ideal for primary key values, such as employee numbers or product numbers.

**Related tasks**

Retrieving rows by using a scrollable cursor

A scrollable cursor is cursor that can be moved in both a forward and a backward direction. Scrollable cursors can be either row-positioned or rowset-positioned.

**Updating previously retrieved data**

To scroll backward through data and update it, use a scrollable cursor that is declared with the FOR UPDATE clause.

**About this task**

If a cursor uses FETCH statements to retrieve columns that will be updated later, specify FOR UPDATE OF when you select the columns. Then specify WHERE CURRENT OF in the subsequent UPDATE or DELETE statements. These clauses prevent Db2 from selecting access through an index on the columns that are being updated, which might otherwise cause Db2 to read the same row more than once.
Procedure

To update previously retrieved data, use these steps:
1. Declare the cursor with the SENSITIVE STATIC SCROLL keywords.
2. Open the cursor.
3. Issue a FETCH statement to position the cursor at the end of the result table.
4. Issue FETCH statements to move the cursor backward to the row that you want to update.
5. Specify the WHERE CURRENT OF clause in the UPDATE or DELETE statement that updates the current row.
6. Repeat steps “4” on page 427 and “5” on page 427 until all required rows are updated.
7. When you have retrieved and updated all the data, close the cursor.

Related reference
update-clause (Db2 SQL)
DECLARE CURSOR (Db2 SQL)
FETCH (Db2 SQL)
UPDATE (Db2 SQL)
DELETE (Db2 SQL)

FETCH statement interaction between row and rowset positioning

When you declare a cursor with the WITH ROWSET POSITIONING clause, you can intermix row-positioned FETCH statements with rowset-positioned FETCH statements.

The following table shows the interaction between row and rowset positioning for a scrollable cursor. Assume that you declare the scrollable cursor on a table with 15 rows.

<table>
<thead>
<tr>
<th>Keywords in FETCH statement</th>
<th>Cursor position when FETCH is executed</th>
</tr>
</thead>
<tbody>
<tr>
<td>FIRST</td>
<td>On row 1</td>
</tr>
<tr>
<td>FIRST ROWSET</td>
<td>On a rowset of size 1, consisting of row 1</td>
</tr>
<tr>
<td>FIRST ROWSET FOR 5 ROWS</td>
<td>On a rowset of size 5, consisting of rows 1, 2, 3, 4, and 5</td>
</tr>
<tr>
<td>CURRENT ROWSET</td>
<td>On a rowset of size 5, consisting of rows 1, 2, 3, 4, and 5</td>
</tr>
<tr>
<td>CURRENT</td>
<td>On row 1</td>
</tr>
<tr>
<td>NEXT (default)</td>
<td>On row 2</td>
</tr>
<tr>
<td>NEXT ROWSET</td>
<td>On a rowset of size 1, consisting of row 3</td>
</tr>
<tr>
<td>NEXT ROWSET FOR 3 ROWS</td>
<td>On a rowset of size 3, consisting of rows 4, 5, and 6</td>
</tr>
<tr>
<td>LAST</td>
<td>On row 15</td>
</tr>
<tr>
<td>LAST ROWSET FOR 2 ROWS</td>
<td>On a rowset of size 2, consisting of rows 14 and 15</td>
</tr>
<tr>
<td>PRIOR ROWSET</td>
<td>On a rowset of size 2, consisting of rows 12 and 13</td>
</tr>
<tr>
<td>ABSOLUTE 2</td>
<td>On row 2</td>
</tr>
<tr>
<td>ROWSET STARTING AT ABSOLUTE 2 FOR 3 ROWS</td>
<td>On a rowset of size 3, consisting of rows 2, 3, and 4</td>
</tr>
<tr>
<td>RELATIVE 2</td>
<td>On row 4</td>
</tr>
<tr>
<td>ROWSET STARTING AT ABSOLUTE 2 FOR 4 ROWS</td>
<td>On a rowset of size 4, consisting of rows 2, 3, 4, and 5</td>
</tr>
</tbody>
</table>
### Table 78. Interaction between row and rowset positioning for a scrollable cursor (continued)

<table>
<thead>
<tr>
<th>Keywords in FETCH statement</th>
<th>Cursor position when FETCH is executed</th>
</tr>
</thead>
<tbody>
<tr>
<td>RELATIVE -1</td>
<td>On row 1</td>
</tr>
<tr>
<td>ROWSET STARTING AT ABSOLUTE 3 FOR 2 ROWS</td>
<td>On a rowset of size 2, consisting of rows 3 and 4</td>
</tr>
<tr>
<td>ROWSET STARTING AT RELATIVE 4</td>
<td>On a rowset of size 2, consisting of rows 7 and 8</td>
</tr>
<tr>
<td>PRIOR</td>
<td>On row 6</td>
</tr>
<tr>
<td>ROWSET STARTING AT ABSOLUTE 13 FOR 5 ROWS</td>
<td>On a rowset of size 3, consisting of rows 13, 14, and 15</td>
</tr>
<tr>
<td>FIRST ROWSET</td>
<td>On a rowset of size 5, consisting of rows 1, 2, 3, 4, and 5</td>
</tr>
</tbody>
</table>

**Related reference**

FETCH (Db2 SQL)

**Examples of fetching rows by using cursors**

You can use SQL statements that you include in a COBOL program to define and use non-scrollable cursor for row-positioned updates, scrollable cursors to retrieve rows backward, non-scrollable cursors for rowset-positioned updates, and scrollable cursors for rowset-positioned operations.

The following example shows how to update a row by using a cursor.

```sql
EXEC SQL
DECLARE THISEMP CURSOR FOR
   SELECT EMPNO, LASTNAME, WORKDEPT, JOB
   FROM DSN8C10.EMP
   WHERE WORKDEPT = 'D11'
   FOR UPDATE OF JOB
END-EXEC.

EXEC SQL
OPEN THISEMP
END-EXEC.

EXEC SQL
WHENEVER NOT FOUND
   GO TO CLOSE-THISEMP
END-EXEC.

EXEC SQL
FETCH FROM THISEMP
   INTO :EMP-NUM, :NAME2, :DEPT, :JOB-NAME
END-EXEC.

EXEC SQL
UPDATE DSN8C10.EMP
   SET JOB = :NEW-JOB
   WHERE CURRENT OF THISEMP
END-EXEC.

EXEC SQL
UPDATE DSN8C10.EMP
   SET JOB = :NEW-JOB
   WHERE CURRENT OF THISEMP
END-EXEC.

EXEC SQL
WHENEVER NOT FOUND
   GO TO CLOSE-THISEMP
END-EXEC.
```

* Declare a cursor that will be used to update the JOB column of the EMP table. *
* Open the cursor *
* Indicate what action to take when all rows in the result table have been fetched. *
* Fetch a row to position the cursor. *
* Update the row where the cursor is positioned. *
* Branch back to fetch and process the next row. *
The following example shows how to retrieve data backward with a cursor.

```
* Declare a cursor to retrieve the data backward from the EMP table. The cursor has access to changes by other processes. *
EXEC SQL
DECLARE THISEMP SENSITIVE STATIC SCROLL CURSOR FOR
SELECT EMPNO, LASTNAME, WORKDEPT, JOB
FROM DSN8C10.EMP
END-EXEC.

* Open the cursor *
EXEC SQL
OPEN THISEMP
END-EXEC.

* Indicate what action to take when all rows in the result table have been fetched. *
EXEC SQL
WHENEVER NOT FOUND GO TO CLOSE-THISEMP
END-EXEC.

* Position the cursor after the last row of the result table. This FETCH statement cannot include the SENSITIVE or INSENSITIVE keyword and cannot contain an INTO clause. *
EXEC SQL
FETCH AFTER FROM THISEMP
END-EXEC.

* Fetch the previous row in the table. *
EXEC SQL
FETCH SENSITIVE PRIOR FROM THISEMP
INTO :EMP-NUM, :NAME2, :DEPT, :JOB-NAME
END-EXEC.

* Check that the fetched row is not a hole. *
* (SQLCODE +222). If not, print the contents. *
IF SQLCODE IS GREATER THAN OR EQUAL TO 0 AND
SQLCODE IS NOT EQUAL TO +100 AND
SQLCODE IS NOT EQUAL TO +222 THEN
PERFORM PRINT-RESULTS.

* Branch back to fetch the previous row. *
EXEC SQL
FETCH SENSITIVE PRIOR FROM THISEMP
INTO :EMP-NUM, :NAME2, :DEPT, :JOB-NAME
END-EXEC.

* Close the cursor *
EXEC SQL
CLOSE THISEMP
END-EXEC.
```

The following example shows how to update an entire rowset with a cursor.

```
* Declare a rowset cursor to update the JOB column of the EMP table. *
EXEC SQL
DECLARE THISEMP SENSITIVE STATIC SCROLL CURSOR FOR
SELECT EMPNO, LASTNAME, WORKDEPT, JOB
FROM DSN8C10.EMP
END-EXEC.

* Open the cursor *
EXEC SQL
OPEN THISEMP
END-EXEC.

* Indicate what action to take when all rows in the result table have been fetched. *
EXEC SQL
WHENEVER NOT FOUND GO TO CLOSE-THISEMP
END-EXEC.

* Position the cursor after the last row of the result table. This FETCH statement cannot include the SENSITIVE or INSENSITIVE keyword and cannot contain an INTO clause. *
EXEC SQL
FETCH AFTER FROM THISEMP
END-EXEC.

* Fetch the previous row in the table. *
EXEC SQL
FETCH SENSITIVE PRIOR FROM THISEMP
INTO :EMP-NUM, :NAME2, :DEPT, :JOB-NAME
END-EXEC.

* Check that the fetched row is not a hole. *
* (SQLCODE +222). If not, print the contents. *
IF SQLCODE IS GREATER THAN OR EQUAL TO 0 AND
SQLCODE IS NOT EQUAL TO +100 AND
SQLCODE IS NOT EQUAL TO +222 THEN
PERFORM PRINT-RESULTS.

* Branch back to fetch the previous row. *
EXEC SQL
FETCH SENSITIVE PRIOR FROM THISEMP
INTO :EMP-NUM, :NAME2, :DEPT, :JOB-NAME
END-EXEC.

* Close the cursor *
EXEC SQL
CLOSE THISEMP
END-EXEC.
```
EXEC SQL
DECLARE EMPSET CURSOR
WITH ROWSET POSITIONING FOR
SELECT EMPNO, LASTNAME, WORKDEPT, JOB
FROM DSN8C10.EMP
WHERE WORKDEPT = 'D11'
FOR UPDATE OF JOB
END-EXEC.

* Open the cursor. *
EXEC SQL
OPEN EMPSET
END-EXEC.

* Indicate what action to take when end-of-data *
* occurs in the rowset being fetched. *
EXEC SQL
WHENEVER NOT FOUND
GO TO CLOSE-EMPSET
END-EXEC.

* Fetch next rowset to position the cursor. *
EXEC SQL
FETCH NEXT ROWSET FROM EMPSET
FOR :SIZE-ROWSET ROWS
INTO :HVA-EMPNO, :HVA-LASTNAME,
:HVA-WORKDEPT, :HVA-JOB
END-EXEC.

* Update rowset where the cursor is positioned. *
EXEC SQL
UPDATE DSN8C10.EMP
SET JOB = :NEW-JOB
WHERE CURRENT OF EMPSET
END-EXEC.

* Branch back to fetch the next rowset. *
EXEC SQL
FETCH NEXT ROWSET FROM EMPSET
FOR :SIZE-ROWSET ROWS
INTO :HVA-EMPNO, :HVA-LASTNAME,
:HVA-WORKDEPT, :HVA-JOB
END-EXEC.

* Update the remaining rows in the current *
* rowset and close the cursor. *
CLOSE-EMPSET.
PERFORM UPDATE-ROWSET.
EXEC SQL
CLOSE EMPSET
END-EXEC.

The following example shows how to update specific rows with a rowset cursor.

EXEC SQL
DECLARE EMPSET SENSITIVE STATIC SCROLL CURSOR
WITH ROWSET POSITIONING FOR
SELECT EMPNO, WORKDEPT, JOB
FROM DSN8C10.EMP
FOR UPDATE OF JOB
END-EXEC.

* Open the cursor. *
EXEC SQL
OPEN EMPSET
END-EXEC.

* Fetch next rowset to position the cursor. *
EXEC SQL
FETCH NEXT ROWSET FROM EMPSET
FOR :SIZE-ROWSET ROWS
INTO :HVA-EMPNO, :HVA-LASTNAME,
:HVA-WORKDEPT, :HVA-JOB
END-EXEC.
EXEC SQL
FETCH SENSITIVE NEXT ROWSET FROM EMPSET
  FOR :SIZE-ROWSET ROWS
  INTO :HVA-EMPNO,
  :HVA-WORKDEPT :INDA-WORKDEPT,
  :HVA-JOB :INDA-JOB
END-EXEC.
**********************************************************************
* Process fetch results if no error and no hole.                      *
**********************************************************************
IF SQLCODE >= 0
EXEC SQL GET DIAGNOSTICS
  :HV-ROWCNT = ROW_COUNT
END-EXEC
PERFORM VARYING N FROM 1 BY 1 UNTIL N > HV-ROWCNT
  IF INDA-WORKDEPT(N) NOT = -3
    EVALUATE HVA-WORKDEPT(N)
      WHEN ('D11')
        PERFORM UPDATE-ROW
      WHEN ('E11')
        PERFORM DELETE-ROW
    END-EVALUATE
  END-IF
END-PERFORM
IF SQLCODE = 100
GO TO CLOSE-EMPSET
ELSE
EXEC SQL GET DIAGNOSTICS
  :HV-NUMCOND = NUMBER
END-EXEC
PERFORM VARYING N FROM 1 BY 1 UNTIL N > HV-NUMCOND
  EXEC SQL GET DIAGNOSTICS CONDITION :N
  :HV-SQLCODE = DB2_RETURNED_SQLCODE,
  :HV-ROWNUM = DB2_ROW_NUMBER
END-EXEC
DISPLAY "SQLCODE = " HV-SQLCODE
DISPLAY "ROW NUMBER = " HV-ROWNUM
END-PERFORM
GO TO CLOSE-EMPSET
END-IF.
...
**********************************************************************
* Branch back to fetch and process                                  *
* the next rowset.                                                   *
**********************************************************************
UPDATE-ROW.
EXEC SQL
  UPDATE DSN8C10.EMP
  SET JOB = :NEW-JOB
  FOR CURSOR EMPSET FOR ROW :N OF ROWSET
END-EXEC.
END-UPDATE-ROW.
**********************************************************************
* Delete row N in current rowset.                                    *
**********************************************************************
DELETE-ROW.
EXEC SQL
  DELETE FROM DSN8C10.EMP
  WHERE CURRENT OF EMPSET FOR ROW :N OF ROWSET
END-EXEC.
END-DELETE-ROW.
...
**********************************************************************
* Close the cursor.                                                  *
**********************************************************************
CLOSE-EMPSET.
EXEC SQL
  CLOSE EMPSET
END-EXEC.
Specifying direct row access by using row IDs

For some applications, you can use the value of a ROWID column to navigate directly to a row.

Before you begin

Ensure that the query qualifies for direct row access. To qualify, the search condition must be a Boolean term, stage 1 predicate that fits one of the following criteria:

- A simple Boolean term predicate of the following form:

  \[
  \text{RID (table designator)} = \text{noncolumn expression}
  \]

  Where the noncolumn expression contains a result of a RID function.

- A compound Boolean term that combines several simple predicates by using the AND operator, where one of the simple predicates fits the first criteria.

About this task

Introductory concepts

ROWID data type (Introduction to Db2 for z/OS)

A ROWID column uniquely identifies each row in a table. With ROWID columns you can write queries that navigate directly to a row in the table because the column implicitly contains the location of the row. You can define a ROWID column as either GENERATED BY DEFAULT or GENERATED ALWAYS:

- If you define the column as GENERATED BY DEFAULT, you can insert a value. Db2 provides a default value if you do not supply one. However, to be able to insert an explicit value (by using the INSERT statement with the VALUES clause), you must create a unique index on that column.
- If you define the column as GENERATED ALWAYS (which is the default), Db2 always generates a unique value for the column. You cannot insert data into that column. In this case, Db2 does not require an index to guarantee unique values.

When you select a ROWID column, the value implicitly contains the location of the retrieved row. If you use the value from the ROWID column in the search condition of a subsequent query, Db2 can choose to navigate directly to that row.

If you define a column in a table to have the ROWID data type, Db2 provides a unique value for each row in the table only if you define the column as GENERATED ALWAYS. The purpose of the value in the ROWID column is to uniquely identify rows in the table.

You can use a ROWID column to write queries that navigate directly to a row, which can be useful in situations where high performance is a requirement. This direct navigation, without using an index or scanning the table space, is called direct row access. In addition, a ROWID column is a requirement for tables that contain LOB columns. This topic discusses the use of a ROWID column in direct row access.

For example, suppose that an EMPLOYEE table is defined in the following way:

```sql
CREATE TABLE EMPLOYEE
(EMP_ROWID ROWID NOT NULL GENERATED ALWAYS,
 EMPNO SMALLINT,
 NAME CHAR(30),
 SALARY DECIMAL(7,2),
 WORKDEPT SMALLINT);
```

The following code uses the SELECT from INSERT statement to retrieve the value of the ROWID column from a new row that is inserted into the EMPLOYEE table. This value is then used to reference that row for the update of the SALARY column.

```sql
EXEC SQL BEGIN DECLARE SECTION;
SQL TYPE IS ROWID hv_emp_rowid;
short    hv_dept, hv_empno;
char     hv_name[30];
decimal(7,2)     hv_salary;
```
EXEC SQL END DECLARE SECTION;
...
EXEC SQL
    SELECT EMP_ROWID INTO :hv_emp_rowid
    FROM FINAL TABLE (INSERT INTO EMPLOYEE
    VALUES (DEFAULT, :hv_empno, :hv_name, :hv_salary, :hv_dept));
EXEC SQL
    UPDATE EMPLOYEE
    SET SALARY = SALARY + 1200
    WHERE EMP_ROWID = :hv_emp_rowid;
EXEC SQL COMMIT;

For Db2 to be able to use direct row access for the update operation, the SELECT from INSERT statement and the UPDATE statement must execute within the same unit of work. Alternatively, you can use a SELECT from MERGE statement. The MERGE statement performs INSERT and UPDATE operations as one coordinated statement.

**Requirement:** To use direct row access, you must use a retrieved ROWID value before you commit. When your application commits, it releases its claim on the table space. After the commit, if a REORG is run on your table space, the physical location of the rows might change.

**Restriction:** In general, you cannot use a ROWID column as a key that is to be used as a single column value across multiple tables. The ROWID value for a particular row in a table might change over time due to a REORG of the table space. In particular, you cannot use a ROWID column as part of a parent key or foreign key.

The value that you retrieve from a ROWID column is a varying-length character value that is not monotonically ascending or descending (the value is not always increasing or not always decreasing). Therefore, a ROWID column does not provide suitable values for many types of entity keys, such as order numbers or employee numbers.

**Procedure**

Call the RID built-in function in the search condition of a SELECT, DELETE, or UPDATE statement. The RID function returns the RID of a row, which you can use to uniquely identify a row.

**Restriction:** Because Db2 might reuse RID numbers when the REORG utility is run, the RID function might return different values when invoked for a row multiple times.

If you specify a RID and Db2 cannot locate the row through direct row access, Db2 does not switch to another access method. Instead, Db2 returns no rows.

**Related concepts**

Rules for inserting data into a ROWID column

A **ROWID column** contains unique values that identify each row in a table. Whether you can insert data into a ROWID column and how that data gets inserted depends on how the column is defined.

**Direct row access (PRIMARY_ACCESSSTYPE='D') (Db2 Performance)**

**Row ID values (Db2 SQL)**

**Related reference**

RID (Db2 SQL)

**Ways to manipulate LOB data**

You can use SQL statements, LOB locators, and LOB file reference variables in your application programs to manipulate LOB data that is stored in Db2.

For example, you can use the following statements to extract information about an employee’s department from the resume:

```sql
EXEC SQL BEGIN DECLARE SECTION;
    char employeenum[6];
    long deptInfoBeginLoc;
    long deptInfoEndLoc;
    SQL TYPE IS CLOB_LOCATOR resume;
EXEC SQL END DECLARE SECTION;
```
These statements use host variables of data type large object locator (LOB locator). LOB locators let you manipulate LOB data without moving the LOB data into host variables. By using LOB locators, you need much smaller amounts of memory for your programs.

You can also use LOB file reference variables when you are working with LOB data. You can use LOB file reference variables to insert LOB data from a file into a Db2 table or to retrieve LOB data from a Db2 table.

**Sample LOB applications:** The following table lists the sample programs that Db2 provides to assist you in writing applications to manipulate LOB data. All programs reside in data set DSN1210.SDSNSAMP.

<table>
<thead>
<tr>
<th>Member that contains source code</th>
<th>Language</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>DSNTEJ7</td>
<td>JCL</td>
<td>Demonstrates how to create a table with LOB columns, an auxiliary table, and an auxiliary index. Also demonstrates how to load LOB data that is 32 KB or less into a LOB table space.</td>
</tr>
<tr>
<td>DSN8DLPL</td>
<td>C</td>
<td>Demonstrates the use of LOB locators and UPDATE statements to move binary data into a column of type BLOB.</td>
</tr>
<tr>
<td>DSN8DLRV</td>
<td>C</td>
<td>Demonstrates how to use a locator to manipulate data of type CLOB.</td>
</tr>
<tr>
<td>DSNTEP2</td>
<td>PL/I</td>
<td>Demonstrates how to allocate an SQLDA for rows that include LOB data and use that SQLDA to describe an input statement and fetch data from LOB columns.</td>
</tr>
</tbody>
</table>

**Related concepts**
LOB file reference variables
In a host application, you can use a file reference variable to insert a LOB or XML value from a file into a Db2 table. You can also use a file reference variable to select a LOB or XML value from a Db2 table into a file.

**Phase 7: Accessing LOB data (Db2 Installation and Migration)**

**Related tasks**
Saving storage when manipulating LOBs by using LOB locators
LOB locators let you manipulate LOB data without retrieving the data from the Db2 table. By using locators, you avoid needing to allocate the large amounts of storage that are needed for host variables to hold LOB data.

**LOB host variable, LOB locator, and LOB file reference variable declarations**

When you write applications to manipulate LOB data, you need to declare host variables to hold the LOB data or LOB locator. Alternatively, you need to declare LOB file reference variables to point to the LOB data.

You can declare LOB host variables and LOB locators in assembler, C, C++, COBOL, Fortran, and PL/I. Additionally, you can declare LOB file reference variables in assembler, C, C++, COBOL, and PL/I. For each host variable, locator, or file reference variable of SQL type BLOB, CLOB, or DBCLOB that you declare, Db2 generates an equivalent declaration that uses host language data types. When you refer to a LOB host variable, LOB locator, or LOB file reference variable in an SQL statement, you must use the variable that you specified in the SQL type declaration. When you refer to the host variable in a host language statement, you must use the variable that Db2 generates.

Db2 supports host variable declarations for LOBs with lengths of up to 2 GB - 1. However, the size of a LOB host variable is limited by the restrictions of the host language and the amount of storage available to the program.

Declare LOB host variables that are referenced by the precompiler in SQL statements by using the SQL TYPE IS BLOB, SQL TYPE IS CLOB, or SQL TYPE IS DBCLOB keywords.

LOB host variables that are referenced only by an SQL statement that uses a DESCRIPTOR should use the same form as declared by the precompiler. In this form, the LOB host-variable-array consists of a 31-bit length, followed by the data, followed by another 31-bit length, followed by the data, and so on. The 31-bit length must be fullword aligned.

**Example:** Suppose that you want to allocate a LOB array of 10 elements, each with a length of 5 bytes. You need to allocate the following bytes for each element, for a total of 120 bytes:

- 4 bytes for the 31-bit integer
- 5 bytes for the data
- 3 bytes to force fullword alignment

The following examples show you how to declare LOB host variables in each supported language. In each table, the left column contains the declaration that you code in your application program. The right column contains the declaration that Db2 generates.

### Declarations of LOB host variables in assembler

The following table shows assembler language declarations for some typical LOB types.

<table>
<thead>
<tr>
<th>You declare this variable</th>
<th>Db2 generates this variable</th>
</tr>
</thead>
</table>
| clob_var SQL TYPE IS CLOB 40000K | clob_var DS 0FL4  
clob_var_length DS FL4  
clob_var_data DS CL65535  
ORG clob_var_data +(4096000-65535) |
| dbclob_var SQL TYPE IS DBCLOB 4000K | dbclob_var DS 0FL4  
dbclob_var_length DS FL4  
dbclob_var_data DS GL65534  
ORG dbclob_var_data+(8192000-65534) |
Table 80. Example of assembler LOB variable declarations (continued)

<table>
<thead>
<tr>
<th>You declare this variable</th>
<th>Db2 generates this variable</th>
</tr>
</thead>
<tbody>
<tr>
<td>blob_var SQL TYPE IS BLOB 1M</td>
<td>blob_var DS 0FL4</td>
</tr>
<tr>
<td></td>
<td>blob_var_length DS FL4</td>
</tr>
<tr>
<td></td>
<td>blob_var_data DS CL65535</td>
</tr>
<tr>
<td></td>
<td>ORG blob_var_data+(1048476-65535)</td>
</tr>
<tr>
<td>clob_loc SQL TYPE IS CLOB_LOCATOR</td>
<td>clob_loc DS FL4</td>
</tr>
<tr>
<td>dbclblob_loc SQL TYPE IS DBCLOB_LOCATOR</td>
<td>dbclblob_loc DS FL4</td>
</tr>
<tr>
<td>blob_loc SQL TYPE IS BLOB_LOCATOR</td>
<td>blob_loc DS FL4</td>
</tr>
<tr>
<td>clob_file SQL TYPE IS CLOB_FILE</td>
<td>clob_file DS FL4</td>
</tr>
<tr>
<td>dbclblob_file SQL TYPE IS DBCLOB_FILE</td>
<td>dbclblob_file DS FL4</td>
</tr>
<tr>
<td>blob_file SQL TYPE IS BLOB_FILE</td>
<td>blob_file DS FL4</td>
</tr>
</tbody>
</table>

Notes:
1. Because assembler language allows character declarations of no more than 65535 bytes, Db2 separates the host language declarations for BLOB and CLOB host variables that are longer than 65535 bytes into two parts.
2. Because assembler language allows graphic declarations of no more than 65534 bytes, Db2 separates the host language declarations for DBCLOB host variables that are longer than 65534 bytes into two parts.

Declarations of LOB host variables in C and C++

The following table shows C and C++ language declarations for some typical LOB types.

Table 81. Examples of C language variable declarations

<table>
<thead>
<tr>
<th>You declare this variable</th>
<th>Db2 generates this variable</th>
</tr>
</thead>
</table>
| SQL TYPE IS BLOB (1M) blob_var; | struct {
|                          | unsigned long length; |
|                          | char data[1048576]; |
|                          | } blob_var; |
| SQL TYPE IS CLOB(400K) clob_var; | struct {
|                          | unsigned long length; |
|                          | char data[409600]; |
|                          | } clob_var; |
| SQL TYPE IS DBCLOB (4000K) dbclblob_var; | struct {
|                          | unsigned long length; |
|                          | sqldbchar data[4096000]; |
|                          | } dbclblob_var; |
Table 81. Examples of C language variable declarations (continued)

You declare this variable | Db2 generates this variable
--- | ---
SQL TYPE IS BLOB_FILE FBLOBhv; | #pragma pack(full)
| struct {
| unsigned long name_length;
| unsigned long data_length;
| unsigned long file_options;
| char name??(255??);
| } FBLOBhv
| #pragma pack(reset)

SQL TYPE IS CLOB_FILE FCLOBhv; | #pragma pack(full)
| struct {
| unsigned long name_length;
| unsigned long data_length;
| unsigned long file_options;
| char name??(255??);
| } FCLOBhv
| #pragma pack(reset)

SQL TYPE IS DBCLOB_FILE FDBCLOBhv; | #pragma pack(full)
| struct {
| unsigned long name_length;
| unsigned long data_length;
| unsigned long file_options;
| char name??(255??);
| } FDBCLOBhv
| #pragma pack(reset)

Declarations of LOB host variables in COBOL

The declarations that are generated for COBOL depend on whether you use the Db2 precompiler or the Db2 coprocessor. The following table shows COBOL declarations that the Db2 precompiler generates for some typical LOB types. The declarations that the Db2 coprocessor generates might be different.

Table 82. Examples of COBOL variable declarations by the Db2 precompiler

You declare this variable | Db2 precompiler generates this variable
--- | ---
01  BLOB-VAR
SQL TYPE IS BLOB(1M). | 01  BLOB-VAR.
| 49 BLOB-VAR-LENGTH PIC S9(9) COMP-5.
| 49 BLOB-VAR-DATA PIC X(1048576).

01  CLOB-VAR
SQL TYPE IS CLOB(40000K). | 01  CLOB-VAR.
| 49 CLOB-VAR-LENGTH PIC S9(9) COMP-5.
| 49 CLOB-VAR-DATA PIC X(40960000).

01  DBCLOB-VAR
SQL TYPE IS DBCLOB(4000K). | 01  DBCLOB-VAR.
| 49 DBCLOB-VAR-LENGTH PIC S9(9) COMP-5
| 49 DBCLOB-VAR-DATA PIC G(40960000) DISPLAY-1.

01  BLOB-LOC
SQL TYPE IS BLOB-LOCATOR. | 01  BLOB-LOC PIC S9(9) COMP-5.

01  CLOB-LOC
SQLTYPE IS CLOB-LOCATOR. | 01  CLOB-LOC PIC S9(9) COMP-5.
Table 82. Examples of COBOL variable declarations by the Db2 precompiler (continued)

<table>
<thead>
<tr>
<th>You declare this variable</th>
<th>Db2 precompiler generates this variable</th>
</tr>
</thead>
<tbody>
<tr>
<td>01 DBCLOB-LOC SQLTYPE IS DBCLOB-LOCATOR.</td>
<td>01 DBCLOB-LOC PIC S9(9) COMP-5.</td>
</tr>
</tbody>
</table>

| 01 BLOB-FILE SQLTYPE IS BLOB-FILE. | 01 BLOB-FILE. 49 BLOB-FILE-NAME-LENGTH PIC S9(9) COMP-5 SYNC. 49 BLOB-FILE-NAME-LENGTH PIC S9(9) COMP-5. 49 BLOB-FILE-FILE-OPTION PIC S9(9) COMP-5. 49 BLOB-FILE-NAME PIC X(255). |

| 01 CLOB-FILE SQLTYPE IS CLOB-FILE. | 01 CLOB-FILE. 49 CLOB-FILE-NAME-LENGTH PIC S9(9) COMP-5 SYNC. 49 CLOB-FILE-NAME-LENGTH PIC S9(9) COMP-5. 49 CLOB-FILE-FILE-OPTION PIC S9(9) COMP-5. 49 CLOB-FILE-NAME PIC X(255). |

| 01 DBCLOB-FILE SQLTYPE IS DBCLOB-FILE. | 01 DBCLOB-FILE. 49 DBCLOB-FILE-NAME-LENGTH PIC S9(9) COMP-5 SYNC. 49 DBCLOB-FILE-NAME-LENGTH PIC S9(9) COMP-5. 49 DBCLOB-FILE-FILE-OPTION PIC S9(9) COMP-5. 49 DBCLOB-FILE-NAME PIC X(255). |

Declarations of LOB host variables in Fortran

The following table shows Fortran declarations for some typical LOB types.

Table 83. Examples of Fortran variable declarations

<table>
<thead>
<tr>
<th>You declare this variable</th>
<th>Db2 generates this variable</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQL TYPE IS BLOB(1M) blob_var</td>
<td>CHARACTER blob_var(1048580) INTEGER*4 blob_var_LENGTH CHARACTER blob_var_DATA EQUIVALENCE( blob_var(1), + blob_var_LENGTH ) EQUIVALENCE( blob_var(5), + blob_var_DATA )</td>
</tr>
</tbody>
</table>

| SQL TYPE IS CLOB(40000K) clob_var | CHARACTER clob_var(4096004) INTEGER*4 clob_var_length CHARACTER clob_var_data EQUIVALENCE( clob_var(1), + clob_var_length ) EQUIVALENCE( clob_var(5), + clob_var_data ) |

| SQL TYPE IS BLOB_LOCATOR blob_loc | INTEGER*4 blob_loc |
| SQL TYPE IS CLOB_LOCATOR clob_loc | INTEGER*4 clob_loc |

Declarations of LOB host variables in PL/I

The declarations that are generated for PL/I depend on whether you use the Db2 precompiler or the Db2 coprocessor. The following table shows PL/I declarations that the Db2 precompiler generates for some typical LOB types. The declarations that the Db2 coprocessor generates might be different.
Table 84. Examples of PL/I variable declarations by the Db2 precompiler

<table>
<thead>
<tr>
<th>You declare this variable</th>
<th>Db2 precompiler generates this variable</th>
</tr>
</thead>
<tbody>
<tr>
<td>DCL BLOB_VAR</td>
<td>DCL 1 BLOB_VAR, 2 BLOB_VAR_LENGTH FIXED BINARY(31), 3 BLOB_VAR_DATA(^1) CHARACTER(32767), 3 BLOB_VAR_DATA2 CHARACTER(1048576-32*32767);</td>
</tr>
<tr>
<td>SQL TYPE IS BLOB (1M);</td>
<td></td>
</tr>
<tr>
<td></td>
<td>DCL CLOB_VAR, 2 CLOB_VAR_LENGTH FIXED BINARY(31), 3 CLOB_VAR_DATA(^1) CHARACTER(32767), 3 CLOB_VAR_DATA2 CHARACTER(4096000-1250*32767);</td>
</tr>
<tr>
<td></td>
<td>DCL DBCLOB_VAR, 2 DBCLOB_VAR_LENGTH FIXED BINARY(31), 3 DBCLOB_VAR_DATA(^2) GRAPHIC(16383), 3 DBCLOB_VAR_DATA2 GRAPHIC(4096000-250*16383);</td>
</tr>
<tr>
<td></td>
<td>DCL blob_loc FIXED BINARY(31);</td>
</tr>
<tr>
<td></td>
<td>DCL clob_loc FIXED BINARY(31);</td>
</tr>
<tr>
<td></td>
<td>DCL dblob_loc FIXED BINARY(31);</td>
</tr>
<tr>
<td></td>
<td>DCL blob_file SQL TYPE IS BLOB_FILE;</td>
</tr>
<tr>
<td></td>
<td>DCL 1 blob_file, 2 blob_file_NAME_LENGTH BIN FIXED(31) ALIGNED, 2 blob_file_DATA_LENGTH BIN FIXED(31), 2 blob_file_FILE_OPTIONS BIN FIXED(31), 2 blob_file_NAME CHAR(255);</td>
</tr>
<tr>
<td></td>
<td>DCL clob_file SQL TYPE IS CLOB_FILE;</td>
</tr>
<tr>
<td></td>
<td>DCL 1 clob_file, 2 clob_file_NAME_LENGTH BIN FIXED(31) ALIGNED, 2 clob_file_DATA_LENGTH BIN FIXED(31), 2 clob_file_FILE_OPTIONS BIN FIXED(31), 2 clob_file_NAME CHAR(255);</td>
</tr>
</tbody>
</table>

Chapter 3. Db2 SQL programming 439
You declare this variable | Db2 precompiler generates this variable
--- | ---
DCL dbclob_file SQL TYPE IS DBCLOB_FILE; | DCL 1
dbcclob_file,
2  dbcclob_file_NAME_LENGTH BIN FIXED(31)
ALIGNED,
2  dbcclob_file_DATA_LENGTH BIN
FIXED(31),
2  dbcclob_file_FILE_OPTIONS BIN
FIXED(31),
2  dbcclob_file_NAME CHAR(255) ;

Notes:

1. For BLOB or CLOB host variables that are greater than 32767 bytes in length, Db2 creates PL/I host language declarations in the following way:
   - If the length of the LOB is greater than 32767 bytes and evenly divisible by 32767, Db2 creates an array of 32767-byte strings. The dimension of the array is length/32767.
   - If the length of the LOB is greater than 32767 bytes but not evenly divisible by 32767, Db2 creates two declarations: The first is an array of 32767 byte strings, where the dimension of the array, n, is length/32767. The second is a character string of length length-n*32767.

2. For DBCLOB host variables that are greater than 16383 double-byte characters in length, Db2 creates PL/I host language declarations in the following way:
   - If the length of the LOB is greater than 16383 characters and evenly divisible by 16383, Db2 creates an array of 16383-character strings. The dimension of the array is length/16383.
   - If the length of the LOB is greater than 16383 characters but not evenly divisible by 16383, Db2 creates two declarations: The first is an array of 16383 byte strings, where the dimension of the array, m, is length/16383. The second is a character string of length length-m*16383.

Related concepts
LOB file reference variables
In a host application, you can use a file reference variable to insert a LOB or XML value from a file into a Db2 table. You can also use a file reference variable to select a LOB or XML value from a Db2 table into a file.

Related tasks
Saving storage when manipulating LOBs by using LOB locators
LOB locators let you manipulate LOB data without retrieving the data from the Db2 table. By using locators, you avoid needing to allocate the large amounts of storage that are needed for host variables to hold LOB data.

LOB and XML materialization
Materialization means that Db2 puts the data that is selected into a buffer for processing. This action can slow performance. Because LOB values can be very large, Db2 avoids materializing LOB data until absolutely necessary.

Beginning in DB2 10, LOB and XML materialization has been reduced or eliminated within Db2 for several local and distributed cases including utilities (LOAD and cross-loader). Some of the cases where materialization has been eliminated or reduced include during DRDA streaming, file reference variable processing, CCSID conversion and distributed XML fetch processing. However, whether the values will be materialized and how much will be materialized also depends on the number and size of each LOB or XML.

Db2 stores LOB values in contiguous storage. Db2 must materialize LOBs when your application program performs the following actions:
   - Calls a user-defined function with a LOB as an argument
• Moves a LOB into or out of a stored procedure
• Assigns a LOB host variable to a LOB locator host variable

The amount of storage that is used for LOB and XML materialization depends on a number of factors including:

• The size of the LOBs
• The number of LOBs that need to be materialized in a statement

Db2 loads LOBs into virtual pools above the bar. If insufficient space is available for LOB materialization, your application receives SQLCODE -904.

Although you cannot completely avoid LOB materialization, you can minimize it by using LOB locators, rather than LOB host variables in your application programs.

**Related tasks**

Saving storage when manipulating LOBs by using LOB locators

LOB locators let you manipulate LOB data without retrieving the data from the Db2 table. By using locators, you avoid needing to allocate the large amounts of storage that are needed for host variables to hold LOB data.

**Saving storage when manipulating LOBs by using LOB locators**

LOB locators let you manipulate LOB data without retrieving the data from the Db2 table. By using locators, you avoid needing to allocate the large amounts of storage that are needed for host variables to hold LOB data.

**About this task**

To retrieve LOB data from a Db2 table, you can define host variables that are large enough to hold all of the LOB data. This requires your application to allocate large amounts of storage, and requires Db2 to move large amounts of data, which can be inefficient or impractical. Instead, you can use LOB locators. LOB locators let you manipulate LOB data without retrieving the data from the Db2 table. Using LOB locators for LOB data retrieval is a good choice in the following situations:

• When you move only a small part of a LOB to a client program
• When the entire LOB does not fit in the application's memory
• When the program needs a temporary LOB value from a LOB expression but does not need to save the result
• When performance is important

A LOB locator is associated with a LOB value or expression, not with a row in a Db2 table or a physical storage location in a table space. Therefore, after you select a LOB value using a locator, the value in the locator normally does not change until the current unit of work ends. However the value of the LOB itself can change.

If you want to remove the association between a LOB locator and its value before a unit of work ends, execute the FREE LOCATOR statement. To keep the association between a LOB locator and its value after the unit of work ends, execute the HOLD LOCATOR statement. After you execute a HOLD LOCATOR statement, the locator keeps the association with the corresponding value until you execute a FREE LOCATOR statement or the program ends.

If you execute HOLD LOCATOR or FREE LOCATOR dynamically, you cannot use EXECUTE IMMEDIATE.

Applications that use a huge number of locators, which commit infrequently, or do not explicitly free the locators, can use large amounts of valuable database services address space (ssnmDBM1) storage and CPU costs. Frequently use COMMIT or FREE LOCATORS to avoid storage shortage on the database services address space (ssnmDBM1) and a shortage of system CPU resource.
To free LOB locators after their associated LOB values are retrieved, run the FREE LOCATOR statement:

```
EXEC SQL FREE LOCATOR :LOCRES, :LOCHIST, :LOCPIC
```

Related reference
FREE LOCATOR (Db2 SQL)
HOLD LOCATOR (Db2 SQL)

**Indicator variables and LOB locators**
Db2 uses indicator variables for LOB locators differently than it uses indicator variables for host variables.

For host variables other than LOB locators, when you select a null value into a host variable, Db2 assigns a negative value to the associated indicator variable. However, for LOB locators, Db2 uses indicator variables differently. A LOB locator is never null. When you select a LOB column using a LOB locator and the LOB column contains a null value, Db2 assigns a null value to the associated indicator variable. The value in the LOB locator does not change. In a client/server environment, this null information is recorded only at the client.

When you use LOB locators to retrieve data from columns that can contain null values, define indicator variables for the LOB locators, and check the indicator variables after you fetch data into the LOB locators. If an indicator variable is null after a fetch operation, you cannot use the value in the LOB locator.

**Valid assignments for LOB locators**
Although you usually use LOB locators to assign data to and retrieve data from LOB columns, you can also use LOB locators to assign data to non-LOB columns.

You can use LOB locators to make the following assignments:

- A CLOB or DBCLOB locator can be assigned to a CHAR, VARCHAR, GRAPHIC, or VARGRAPHIC column. However, you cannot fetch data from CHAR, VARCHAR, GRAPHIC, or VARGRAPHIC columns into a CLOB or DBCLOB locators.
- A BLOB locator can be assigned to a BINARY or VARBINARY column. However, you cannot fetch data from a BINARY or VARBINARY column into a BLOB locator.

**Avoiding character conversion for LOB locators**
In certain situations, Db2 materializes the entire LOB value and converts it to the encoding scheme of a particular SQL statement. This extra processing can degrade performance and should be avoided.

**About this task**
You can use a VALUES INTO or SET statement to obtain the results of functions that operate on LOB locators, such as LENGTH or SUBSTR. VALUES INTO and SET statements are processed in the application encoding scheme for the plan or package that contains the statement. If that encoding scheme is different from the encoding scheme of the LOB data, the entire LOB value is materialized and converted to the encoding scheme of the statement. This materialization and conversion processing can cause performance degradation.

To avoid the character conversion, SELECT from the SYSIBM.SYSDUMMYA, SYSIBM.SYSDUMMYE, or SYSIBM.SYSDUMMYU sample table. These dummy tables perform functions similar to SYSIBM.SYSDUMMY1, and are each associated with an encoding scheme:

- **SYSIBM.SYSDUMMYA**
  ASCII
- **SYSIBM.SYSDUMMYE**
  EBCDIC
- **SYSIBM.SYSDUMMYU**
  Unicode

By using these tables, you can obtain the same result as you would with a VALUES INTO or SET statement.
Example
Suppose that the encoding scheme of the following statement is EBCDIC:

```
SET : unicode_hv = SUBSTR(:Unicode_lob_locator,X,Y);
```

Db2 must materialize the LOB that is specified by :Unicode_lob_locator and convert that entire LOB to EBCDIC before executing the statement. To avoid materialization and conversion, you can execute the following statement, which produces the same result but is processed by the Unicode encoding scheme of the table:

```
SELECT SUBSTR(:Unicode_lob_locator,X,Y) INTO :unicode_hv
FROM SYSIBM.SYSDUMMYU;
```

Deferring evaluation of a LOB expression to improve performance

Db2 does not move any bytes of a LOB value until a program assigns a LOB expression to a target destination. When you use a LOB locator with string functions and operators, Db2 does not evaluate the expression until the time of assignment. This deferred evaluation can improve performance.

About this task

The following example is a C language program that defers evaluation of a LOB expression. The program runs on a client and modifies LOB data at a server. The program searches for a particular resume (EMPNO = '000130') in the EMP_RESUME table. It then uses LOB locators to rearrange a copy of the resume (with EMPNO = 'A00130'). In the copy, the Department Information Section appears at the end of the resume. The program then inserts the copy into EMP_RESUME without modifying the original resume.

Because the program in the following figure uses LOB locators, rather than placing the LOB data into host variables, no LOB data is moved until the INSERT statement executes. In addition, no LOB data moves between the client and the server.

```sql
EXEC SQL INCLUDE SQLCA;
/***********************
* Declare host variables */
/***************************/
EXEC SQL BEGIN DECLARE SECTION;
char userid[9];
char passwd[19];
long HV_START_DEPTINFO;
long HV_START_EDUC;
long HV_RETURN_CODE;
SQL TYPE IS CLOB_LOCATOR HV_NEW_SECTION_LOCATOR;
SQL TYPE IS CLOB_LOCATOR HV_DOC_LOCATOR1;
SQL TYPE IS CLOB_LOCATOR HV_DOC_LOCATOR2;
SQL TYPE IS CLOB_LOCATOR HV_DOC_LOCATOR3;
EXEC SQL END DECLARE SECTION;

/**********************
* Delete any instance of "A00130" from previous */
* executions of this sample */
EXEC SQL DELETE FROM EMP_RESUME WHERE EMPNO = 'A00130';

/**********************
* Use a single row select to get the document */
EXEC SQL SELECT RESUME INTO :HV_DOC_LOCATOR1
FROM EMP_RESUME
WHERE EMPNO = '000130'
AND Resume_FORMAT = 'ascii';

/**********************
* Use the POSSTR function to locate the start of */
* sections "Department Information" and "Education" */
EXEC SQL SET :HV_START_DEPTINFO =
POSSTR(:HV_DOC_LOCATOR1, 'Department Information');
```
EXEC SQL SET :HV_START_EDUC = POSSTR(:HV_DOC_LOCATOR1, 'Education');

EXEC SQL SET :HV_DOC_LOCATOR2 = SUBSTR(:HV_DOC_LOCATOR1, 1, :HV_START_DEPTINFO -1) || SUBSTR (:HV_DOC_LOCATOR1, :HV_START_EDUC);

EXEC SQL SET :HV_NEW_SECTION_LOCATOR = SUBSTR(:HV_DOC_LOCATOR1, :HV_START_DEPTINFO, :HV_START_EDUC - :HV_START_DEPTINFO);

EXEC SQL SET :HV_DOC_LOCATOR3 = :HV_DOC_LOCATOR2 || :HV_NEW_SECTION_LOCATOR;

EXEC SQL INSERT INTO EMP_RESUME VALUES ('A00130', 'ascii', :HV_DOC_LOCATOR3, DEFAULT);

EXEC SQL FREE LOCATOR :HV_DOC_LOCATOR1, :HV_DOC_LOCATOR2, :HV_DOC_LOCATOR3;

Notes:

1. Declare the LOB locators here.

2. This SELECT statement associates LOB locator HV_DOC_LOCATOR1 with the value of column RESUME for employee number 000130.

3. The next five SQL statements use LOB locators to manipulate the resume data without moving the data.

4. Evaluation of the LOB expressions in the previous statements has been deferred until execution of this INSERT statement.

5. Free all LOB locators to release them from their associated values.

LOB file reference variables

In a host application, you can use a file reference variable to insert a LOB or XML value from a file into a Db2 table. You can also use a file reference variable to select a LOB or XML value from a Db2 table into a file.

The file reference variables are BLOB_FILE, CLOB_FILE, or DBCLOB_FILE. For COBOL, the file reference variables are BLOB-FILE, CLOB-FILE, or DBCLOB-FILE.

When you use a file reference variable, you can select or insert an entire LOB or XML value without contiguous application storage to contain the entire LOB or XML value. LOB file reference variables move LOB or XML values from the database server to an application or from an application to the database server without going through the application's memory. Furthermore, LOB file reference variables bypass the host language limitation on the maximum size allowed for dynamic storage to contain a LOB value.

You can declare LOB or XML values as LOB file reference variables or LOB file reference arrays for applications that are written in C, COBOL, PL/I, and assembler. The LOB file reference variables do not
contain LOB data; they represent a file that contains LOB data. Database queries, updates, and inserts can use file reference variables to store or retrieve column values. As with other host variables, a LOB file reference variable can have an associated indicator variable.

Db2-generated LOB file reference variable constructs
For each LOB file reference variable that an application declares for a LOB or XML value, Db2 generates an equivalent construct that uses the host language data types. When an application references a LOB file reference variable, it must use the equivalent construct that Db2 generates; otherwise the Db2 precompiler issues an error.

The construct describes the following properties of the file:

Data type
BLOB, CLOB, or DBCLOB. This property is specified when the variable is declared by using the BLOB_FILE, CLOB_FILE, or DBCLOB_FILE data type.
For COBOL, the data types are BLOB-FILE, CLOB-FILE, or DBCLOB-FILE.

Direction
This property must be specified by the application program at run time as part of the file option property. The direction property can have the following values:

Input
Used as a data source on an EXECUTE, OPEN, UPDATE, INSERT, DELETE, SET, or MERGE statement.

Output
Used as the target of data on a FETCH statement or a SELECT INTO statement.

File name
This property must be specified by the application program at run time. The file name property can have the following values:

• The complete path name of the file. This is recommended.

File name length
This property must be specified by the application program at run time.

File options
An application must assign one of the file options to a file reference variable before the application can use that variable. File options are set by the INTEGER value in a field in the file reference variable construct. One of the following values must be specified for each file reference variable:

• Input (from application to database):

  SQL_FILE_READ
  A regular file that can be opened, read, and closed.

• Output (from database to application):

  SQL_FILE_CREATE
  If the file does not exists, a new file is created. If the file already exists, an error is returned.

  SQL_FILE_OVERWRITE
  If the file does not exists, a new file is created. If the file already exists, it is overwritten.

  SQL_FILE_APPEND
  If the file does not exists, a new file is created. If the file already exists, the output is appended to the existing file.

Data length
The length, in bytes, of the new data written to the file.
**Examples of declaring file reference variables**

You can declare a file reference variable in C, COBOL, and PL/I, and declare the file reference variable construct that Db2 generates.

**C Example:** Consider the following C declaration:

```c
EXEC SQL BEGIN DECLARE SECTION
SQL TYPE IS CLOB_FILE hv_text_file;
CHAR hv_thesis_title[64];
EXEC SQL END DECLARE SECTION
```

That declaration results in the following Db2-generated construct:

```sql
EXEC SQL BEGIN DECLARE SECTION
/* SQL TYPE IS CLOB_FILE hv_text_file; */
struct {
  unsigned long name_length;  // File name length
  unsigned long data_length;  // Data length
  unsigned long file_options; // File options
  char name [255];            // File name
} hv_text_file;
char hv_thesis_title[64]
```

With the Db2-generated construct, you can use the following code to select from a CLOB column in the database into a new file that is referenced by :hv_text_file. The file name must be an absolute path.

```sql
strcopy(hv_text_file.name, "/u/gainer/papers/sigmod.94");
hv_text_file.name_length = strlen("/u/gainer/papers/sigmod.94");
hv_text_file.file_options = SQL_FILE_CREATE;
EXEC SQL SELECT CONTENT INTO :hv_text_file FROM PAPERS
    WHERE TITLE = 'The Relational Theory Behind Juggling';
```

Similarly, you can use the following code to insert the data from a file that is referenced by :hv_text_file into a CLOB column. The file name must be an absolute path.

```sql
strcpy(hv_text_file.name, "/u/gainer/patents/chips.13");
hv_text_file.name_length = strlen("/u/gainer/patents/chips.13");
hv_text_file.file_options = SQL_FILE_READ;
strcpy(:hv_patent_title, "A Method for Pipelining Chip Consumption");
EXEC SQL INSERT INTO PATENTS(TITLE, TEXT)
    VALUES(:hv_patent_title, :hv_text_file);
```

**COBOL Example:** Consider the following COBOL declaration:

```cobol
01 MY-FILE SQL TYPE IS BLOB-FILE
```

That declaration results in the following Db2-generated construct:

```cobol
01 MY-FILE.
  49 MY-FILE-NAME-LENGTH PIC S9(9) COMP-5.
  49 MY-FILE-DATA-LENGTH PIC S9(9) COMP-5.
  49 MY-FILE-FILE-OPTION PIC S9(9) COMP-5.
  49 MY-FILE-NAME PIC(255);
```

**PL/I Example:** Consider the following PL/I declaration:

```pli
DCL MY_FILE SQL TYPE IS CLOB_FILE
```

That declaration results in the following Db2-generated construct:

```pli
DCL 1 MY_FILE.
  3 MY_FILE_NAME_LENGTH BINARY FIXED (31) UNALIGNED,
  3 MY_FILE_DATA_LENGTH BINARY FIXED (31) UNALIGNED,
  3 MY_FILE_FILE_OPTIONS BINARY FIXED (31) UNALIGNED,
  3 MY_FILE_NAME CHAR(255);
```
For examples of how to declare file reference variables for XML data in C, COBOL, and PL/I, see “Host variable data types for XML data in embedded SQL applications” on page 542.

Referencing a sequence object

A sequence object is a user-defined object that generates a sequence of numeric values according to the specification with which the sequence was created. You can retrieve the next or previous value in the sequence.

About this task

You reference a sequence by using the NEXT VALUE expression or the PREVIOUS VALUE expression, specifying the name of the sequence:

- A NEXT VALUE expression generates and returns the next value for the specified sequence. If a query contains multiple instances of a NEXT VALUE expression with the same sequence name, the sequence value increments only once for that query. The ROLLBACK statement has no effect on values already generated.

- A PREVIOUS VALUE expression returns the most recently generated value for the specified sequence for a previous NEXT VALUE expression that specified the same sequence within the current application process. The value of the PREVIOUS VALUE expression persists until the next value is generated for the sequence, the sequence is dropped, or the application session ends. The COMMIT statement and the ROLLBACK statement have no effect on this value.

You can specify a NEXT VALUE or PREVIOUS VALUE expression in a SELECT clause, within a VALUES clause of an insert operation, within the SET clause of an update operation (with certain restrictions), or within a SET host-variable statement.

Retrieving thousands of rows

When retrieving large numbers of rows, consider the possibilities for lock escalation and other locking issues.

About this task

Question: Are there any special techniques for fetching and displaying large volumes of data?

Answer: There are no special techniques; but for large numbers of rows, efficiency can become very important. In particular, you need to be aware of locking considerations, including the possibilities of lock escalation.

If your program allows input from a terminal before it commits the data and thereby releases locks, it is possible that a significant loss of concurrency results.

Determining when a row was changed

If a table has a ROW CHANGE TIMESTAMP column, you can determine when a row was changed.

Procedure

Issue a SELECT statement with the ROW CHANGE TIMESTAMP column in the column list.

If a qualifying row does not have a value for the ROW CHANGE TIMESTAMP column, Db2 returns the time that the page in which that row resides was updated.

Example

Suppose that you issue the following statements to create, populate, and alter a table:

```
CREATE TABLE T1 (C1 INTEGER NOT NULL);
INSERT INTO T1 VALUES (1);
ALTER TABLE T1 ADD COLUMN C2 NOT NULL GENERATED ALWAYS
```
FOR EACH ROW ON UPDATE AS ROW CHANGE TIMESTAMP;
SELECT T1.C2 FROM T1 WHERE T1.C1 = 1;

Because the ROW CHANGE TIMESTAMP column was added after the data was inserted, the following statement returns the time that the page was last modified:

SELECT T1.C2 FROM T1 WHERE T1.C1 = 1;

Assume that you then issue the following statement:

INSERT INTO T1(C1) VALUES (2);

Assume that this row is added to the same page as the first row. The following statement returns the time that value “2” was inserted into the table:

SELECT T1.C2 FROM T1 WHERE T1.C1 = 2;

Because the row with value “1” still does not have a value for the ROW CHANGE TIMESTAMP column, the following statement still returns the time that the page was last modified, which in this case is the time that value “2” was inserted:

SELECT T1.C2 FROM T1 WHERE T1.C1 = 1;

Related reference
CREATE TABLE (Db2 SQL)

Checking whether an XML column contains a certain value

You can determine which rows contain any fragment of XML data that you specify.

Procedure

Specify the XMLEXISTS predicate in the WHERE clause of your SQL statement.

Include the following parameters for the XMLEXISTS predicate:

- An XPath expression that is embedded in a character string literal. Specify an XPath expression that identifies the XML data that you are looking for. If the result of the XPath expression is an empty sequence, XMLEXISTS returns false. If the result is not empty, XMLEXISTS returns true. If the evaluation of the XPath expression returns an error, XMLEXISTS returns an error.
- The XML column name. Specify this value after the PASSING keyword.

Example

Suppose that you want to return only purchase orders that have a billing address. Assume that column XMLPO stores the XML purchase order documents and that the billTo nodes within these documents contain any billing addresses. You can use the following SELECT statement with the XMLEXISTS predicate:

```
SELECT XMLPO FROM T1
WHERE XMLEXISTS ('declare namespace ipo="http://www.example.com/IPO";
/ipo:purchaseOrder[billTo]' PASSING XMLPO);
```

Related reference
XMLEXISTS predicate (Db2 SQL)
Accessing Db2 data that is not in a table

You can access Db2 data that is not in a table by returning the value of an SQL expression in a host variable.

Procedure

To return the value of an SQL expression that does not include the value of a table column in host variable, use one of the following approaches:

- Use the SET host-variable assignment statement to set the contents of a host variable to the value of an expression.
  
  ```sql
  EXEC SQL SET :hvrandval = RAND(:hvrand); 
  ```

- Use the VALUES INTO statement to return the value of an expression in a host variable.
  
  ```sql
  EXEC SQL VALUES RAND(:hvrand)
  INTO :hvrandval; 
  ```

- Use the following statement to select the expression from the Db2-provided EBCDIC table, named SYSIBM.SYSDUMMY1, which consists of one row.
  
  ```sql
  EXEC SQL SELECT RAND(:hvrand)
  INTO :hvrandval 
  FROM SYSIBM.SYSDUMMY1; 
  ```

Related reference

- SET assignment-statement (Db2 SQL)
- VALUES INTO (Db2 SQL)
- SYSDUMMY1 catalog table (Db2 SQL)

Ensuring that queries perform sufficiently

It is important to make sure that any individual queries that are included in your program are not slowing down the performance of your program.

About this task

**Tip:** You can use tools such as IBM Data Server Manager and related Db2 Tools for z/OS solutions to simplify this task.

- IBM Data Server Manager (Introduction to Db2 for z/OS)
- Db2 Tools for z/OS

Procedure

To ensure that queries perform sufficiently:

1. Tune each query in your program by following the general tuning guidelines for how to write efficient queries.
2. If you suspect that a query is not as efficient as it could be, monitor its performance.
   
   You can use a number of different functions and techniques to monitor SQL performance, including the SQL EXPLAIN statement and SQL optimization tools.

Related concepts

- Investigating SQL performance by using EXPLAIN (Db2 Performance)
- Interpreting data access by using EXPLAIN (Db2 Performance)

Related tasks

- Programming applications for performance (Db2 Performance)
- Investigating access path problems (Db2 Performance)
Items to include in a batch DL/I program

When you use a batch DL/I program with Db2, you must include certain items in your program.

A batch DL/I program can issue:

- Any IMS batch call, except ROLS, SETS, and SYNC calls. ROLS and SETS calls provide intermediate backout point processing, which Db2 does not support. The SYNC call provides commit point processing without identifying the commit point with a value. IMS does not allow a SYNC call in batch, and neither does the Db2 DL/I batch support.

  Issuing a ROLS, SETS, or SYNC call in an application program causes a system abend X'04E' with the reason code X'00D44057' in register 15.

- GSAM calls.
- IMS system services calls.
- Any SQL statements, except COMMIT and ROLLBACK. IMS and CICS environments do not allow those SQL statements; however, IMS and CICS do allow ROLLBACK TO SAVEPOINT. You can use the IMS CHKRP call to commit data and the IMS ROLL or ROLB to roll back changes.

  Issuing a COMMIT statement causes SQLCODE -925; issuing a ROLLBACK statement causes SQLCODE -926. Those statements also return SQLSTATE '2D521'.

- Any call to a standard or traditional access method (for example, QSAM, VSAM, and so on).

The restart capabilities for Db2 and IMS databases, as well as for sequential data sets that are accessed through GSAM, are available through the IMS Checkpoint and Restart facility.

Db2 allows access to both Db2 and DL/I data through the use of the following Db2 and IMS facilities:

- IMS synchronization calls, which commit and abnormally terminate units of recovery
- The Db2 IMS attachment facility, which handles the two-phase commit protocol and enables both systems to synchronize a unit of recovery during a restart after a failure
- The IMS log, which is used to record the instant of commit

In a data sharing environment, DL/I batch supports group attachment or subgroup attachment. You can specify a group attachment name instead of a subsystem name in the SSN parameter of the DDITV02 data set for the DL/I batch job.

Requirements for using Db2 in a DL/I batch job

Using Db2 in a DL/I batch job requires the following changes to the application program and the job step JCL:

- Add SQL statements to your application program to gain access to Db2 data. You must then precompile the application program and bind the resulting DBRM into a package.
- Before you run the application program, use JOBLIB, STEPLIB, or link book to access the Db2 load library, so that Db2 modules can be loaded.
- In a data set that is specified by a DDITV02 DD statement, specify the program name and plan name for the application, and the connection name for the DL/I batch job.

  In an input data set or in a subsystem member, specify information about the connection between Db2 and IMS. The input data set name is specified with a DDITV02 DD statement. The subsystem member name is specified by the parameter SSM= on the DL/I batch invocation procedure.

  Optionally specify an output data set using the DDOTV02 DD statement. You might need this data set to receive messages from the IMS attachment facility about indoubt threads and diagnostic information.
Program design considerations for using DL/I batch

**Address spaces in DL/I batch:**
A DL/I batch region is independent of both the IMS control region and the CICS address space. The DL/I batch region loads the DL/I code into the application region along with the application program.

**Commits in DL/I batch:**
Commit IMS batch applications frequently so that you do not use resources for an extended time.

**SQL statements and IMS calls in DL/I batch:**
DL/I batch applications cannot use the SQL COMMIT and ROLLBACK statements; otherwise, you get an SQL error code. DL/I batch applications also cannot use ROLS, SETS, and SYNC calls; otherwise the application program abnormally terminates.

**Checkpoint calls in DL/I batch:**
Write your program with SQL statements and DL/I calls, and use checkpoint calls. The frequency of checkpoints depends on the application design. All checkpoints that are issued by a batch application program must be unique. At a checkpoint, DL/I positioning is lost, Db2 cursors are closed (with the possible exception of cursors that are defined as WITH HOLD), commit duration locks are freed (again with some exceptions), and database changes are considered permanent to both IMS and Db2.

**Application program synchronization in DL/I batch:**
You can design an application program without using IMS checkpoints. In that case, if the program abnormally terminates before completing, Db2 backs out any updates, and you can use the IMS batch backout utility to back out the DL/I changes.

You can also have IMS dynamically back out the updates within the same job. You must specify the BKO parameter as ‘Y’ and allocate the IMS log to DASD.

You could have a problem if the system on which the job is run fails after the program terminates but before the job step ends. If you do not have a checkpoint call before the program ends, Db2 commits the unit of work without involving IMS. If the system fails before DL/I commits the data, the Db2 data is out of synchronization with the DL/I changes. If the system fails during Db2 commit processing, the Db2 data could be indoubt. When you restart the application program, use the XRST call to obtain checkpoint information and resolve any Db2 indoubt work units.

**Recommendation:** Always issue a symbolic checkpoint at the end of any update job to coordinate the commit of the outstanding unit of work for IMS and Db2.

**Checkpoint and XRST considerations in DL/I batch:**
If you use an XRST call, Db2 assumes that any checkpoint that is issued is a symbolic checkpoint. The options of the symbolic checkpoint call differ from the options of a basic checkpoint call. Using the incorrect form of the checkpoint call can cause problems.

If you do not use an XRST call, Db2 assumes that any checkpoint call that is issued is a basic checkpoint.

To make restart easier, use EBCDIC characters for checkpoint IDs.

When an application program needs to be restartable, you must use symbolic checkpoint and XRST calls. If you use an XRST call, it must be the first IMS call that is issued, and it must occur before any SQL statement. Also, you must use only one XRST call.

**Synchronization call abends in DL/I batch:**
If the application program contains an incorrect IMS synchronization call (CHKP, ROLB, ROLL, or XRST), causing IMS to issue a bad status code in the PCB, Db2 abends the application program. Be sure to test these calls before placing the programs in production.

**Related concepts**
- Input and output data sets for DL/I batch jobs
- DL/I batch jobs require an input data set with DD name DDITV02 and an output data set with DD name DDOTV02.
- Multiple system consistency (Db2 Administration Guide)
Related tasks
Preparing an application to run on Db2 for z/OS
To prepare and run applications that contain embedded static SQL statements or dynamic SQL statements, you must precompile, compile, link-edit, and bind them.

Invoking a user-defined function

You can use a user-defined function wherever you can use a built-in function.

You can invoke a sourced or external user-defined scalar function in an SQL statement wherever you use an expression. For a table function, you can invoke the user-defined function only in the FROM clause of a SELECT statement. The invoking SQL statement can be in a stand-alone program, a stored procedure, a trigger body, or another user-defined function.

Recommendations for invoking user-defined functions:

Invoke user-defined functions with external actions and nondeterministic user-defined functions from select lists: Invoking user-defined functions with external action from a select list and nondeterministic user-defined functions from a select list is preferred to invoking these user-defined functions from a predicate.

The access path that Db2 chooses for a predicate determines whether a user-defined function in that predicate is invoked. To ensure that Db2 executes the external action for each row of the result table, put the user-defined function invocation in the SELECT list.

Invoking a nondeterministic user-defined function from a predicate can yield undesirable results. The following example demonstrates this idea.

Suppose that you execute this query:

```sql
SELECT COUNTER(), C1, C2 FROM T1 WHERE COUNTER() = 2;
```

Table T1 looks like this:

<table>
<thead>
<tr>
<th>C1</th>
<th>C2</th>
</tr>
</thead>
<tbody>
<tr>
<td>--</td>
<td>--</td>
</tr>
<tr>
<td>1</td>
<td>b</td>
</tr>
<tr>
<td>2</td>
<td>c</td>
</tr>
<tr>
<td>3</td>
<td>a</td>
</tr>
</tbody>
</table>

COUNTER is a user-defined function that increments a variable in the scratchpad each time it is invoked.

Db2 invokes an instance of COUNTER in the predicate 3 times. Assume that COUNTER is invoked for row 1 first, for row 2 second, and for row 3 third. Then COUNTER returns 1 for row 1, 2 for row 2, and 3 for row 3. Therefore, row 2 satisfies the predicate WHERE COUNTER()=2, so Db2 evaluates the SELECT list for row 2. Db2 uses a different instance of COUNTER in the select list from the instance in the predicate.

Because the instance of COUNTER in the select list is invoked only once, it returns a value of 1. Therefore, the result of the query is:

<table>
<thead>
<tr>
<th>COUNTER()</th>
<th>C1</th>
<th>C2</th>
</tr>
</thead>
<tbody>
<tr>
<td>---------</td>
<td>--</td>
<td>--</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>c</td>
</tr>
</tbody>
</table>

This is not the result you might expect.

The results can differ even more, depending on the order in which Db2 retrieves the rows from the table. Suppose that an ascending index is defined on column C2. Then Db2 retrieves row 3 first, row 1 second, and row 2 third. This means that row 1 satisfies the predicate WHERE COUNTER()=2. The value of COUNTER in the select list is again 1, so the result of the query in this case is:

<table>
<thead>
<tr>
<th>COUNTER()</th>
<th>C1</th>
<th>C2</th>
</tr>
</thead>
<tbody>
<tr>
<td>---------</td>
<td>--</td>
<td>--</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>b</td>
</tr>
</tbody>
</table>
Understand the interaction between scrollable cursors and nondeterministic user-defined functions or user-defined functions with external actions: When you use a scrollable cursor, you might retrieve the same row multiple times while the cursor is open. If the select list of the cursor's SELECT statement contains a user-defined function, that user-defined function is executed each time you retrieve a row. Therefore, if the user-defined function has an external action, and you retrieve the same row multiple times, the external action is executed multiple times for that row.

A similar situation occurs with scrollable cursors and nondeterministic functions. The result of a nondeterministic user-defined function can be different each time you execute the user-defined function. If the select list of a scrollable cursor contains a nondeterministic user-defined function, and you use that cursor to retrieve the same row multiple times, the results can differ each time you retrieve the row.

A nondeterministic user-defined function in the predicate of a scrollable cursor's SELECT statement does not change the result of the predicate while the cursor is open. Db2 evaluates a user-defined function in the predicate only once while the cursor is open.

Related concepts
Abnormal termination of an external user-defined function
If an external user-defined function abnormally terminates, your program receives SQLCODE -430 for invoking the statement.

Function invocation (Db2 SQL)
Related reference
from-clause (Db2 SQL)

How Db2 determines the authorization for invoking user-defined functions
Both the authorization used to invoke a user-defined function and the authorization used for executing each SQL statement in the function influence the processing of a user-defined function.

The authorization that is required to invoke a user-defined function depends on whether the function is invoked statically or dynamically:

- For static invocations, the authorization of the owner of the package that contains the invocation of the user-defined function is used.
- For dynamic invocations, the DYNAMICRULES bind option of the package that contains the invocation of the user-defined determines the authorization that is used. For more information about how Db2 applies the DYNAMICRULES bind option, see DYNAMICRULES bind option (Db2 Commands).

Similarly, the authorization that Db2 uses to process each SQL statement inside a user-defined function depends on whether the statement is a static or dynamic SQL statement:

- For static SQL statements, the authorization of the owner of the user-defined function is used.
- For dynamic SQL statements, the DYNAMICRULES option of the CREATE FUNCTION statement determines the authorization that is used.

Related concepts
Function invocation (Db2 SQL)
Related reference
CREATE FUNCTION (Db2 SQL)

Ensuring that Db2 executes the intended user-defined function
Multiple functions can with the same name can exist in the same schema or in different schemas. You can take certain actions so that Db2 chooses the correct function to execute.

About this task
The combination of the function name and the parameter list form the signature that Db2 uses to identify a function. For detailed information about the rules and process that Db2 uses to identify the function to invoke, see Function resolution (Db2 SQL).
If the signatures of two functions match, including built-in and user-defined functions, you must take appropriate action to ensure that Db2 invokes the correct intended function.

**Procedure**

To simplify the resolution of built-in and user-defined functions, use the following techniques:

- When you invoke a function, use the qualified name.
  
  This causes Db2 to search for functions only in the schema you specify. This approach has the following advantages:
  
  - Db2 is less likely to choose a function that you did not intend to use. Several functions might fit the invocation equally well. Db2 picks the function whose schema name is listed first in the SQL path, which might not be the function you want.
  
  - The number of candidate functions is smaller, so Db2 takes less time for function resolution.

- Cast parameters in a user-defined function invocation to the types in the user-defined function definition. For example, if an input parameter for user-defined function FUNC is defined as DECIMAL(13,2), and the value you want to pass to the user-defined function is an integer value, cast the integer value to DECIMAL(13,2):

  ```sql
  SELECT FUNC(CAST (INTCOL AS DECIMAL(13,2))) FROM T1;
  ```

- Use the data type BIGINT for numeric parameters in a user-defined function.

  When you invoke the function, you can pass in SMALLINT, INTEGER, or BIGINT values. If you use SMALLINT or REAL as the parameter type, you must pass parameters of the same types. For example, if user-defined function FUNC is defined with a parameter of type SMALLINT, only an invocation with a parameter of type SMALLINT resolves correctly. The following call does not resolve to FUNC because the constant 123 is of type INTEGER, not SMALLINT:

  ```sql
  SELECT FUNC(123) FROM T1;
  ```

- Avoid defining user-defined function string parameters with fixed-length string types.

  If you define a parameter with a fixed-length string type (CHAR, GRAPHIC, or BINARY), you can invoke the user-defined function only with a fixed-length string parameter. However, if you define the parameter with a varying-length string type (VARCHAR, VARGRAPHIC, or VARBINARY), you can invoke the user-defined function with either a fixed-length string parameter or a varying-length string parameter.

  If you must define parameters for a user-defined function as CHAR or BINARY, and you call the user-defined function from a C program or SQL procedure, you need to cast the corresponding parameter values in the user-defined function invocation to CHAR or BINARY to ensure that Db2 invokes the correct function. For example, suppose that a C program calls user-defined function CVRTNUM, which takes one input parameter of type CHAR(6). Also suppose that you declare host variable empnumbr as char empnumbr[6]. When you invoke CVRTNUM, cast empnumbr to CHAR:

  ```sql
  UPDATE EMP
  SET EMPNO=CVRTNUM(CHAR(:empnumbr))
  WHERE EMPNO = :empnumbr;
  ```

**Related concepts**

- Functions (Db2 SQL)
How Db2 resolves functions

Function resolution is the process by which Db2 determines which user-defined function or built-in function to execute. You need to understand the function resolution process that Db2 uses to ensure that you invoke the user-defined function that you want to invoke.

Several user-defined functions with the same name but different numbers or types of parameters can exist in a Db2 subsystem. Several user-defined functions with the same name can have the same number of parameters, as long as the data types of any of the first 30 parameters are different. In addition, several user-defined functions might have the same name as a built-in function. When you invoke a function, Db2 must determine which user-defined function or built-in function to execute.

Db2 performs these steps for function resolution:

1. Determines if any function instances are candidates for execution. If no candidates exist, Db2 issues an SQL error message.
2. Compares the data types of the input parameters to determine which candidates fit the invocation best.
   Db2 does not compare data types for input parameters that are untyped parameter markers.
   For a qualified function invocation, if there are no parameter markers in the invocation, the result of the data type comparison is one best fit. That best fit is the choice for execution. If there are parameter markers in the invocation, there might be more than one best fit. Db2 issues an error if there is more than one best fit.
   For an unqualified function invocation, Db2 might find multiple best fits because the same function name with the same input parameters can exist in different schemas, or because there are parameter markers in the invocation.
3. If two or more candidates fit the unqualified function invocation equally well because the same function name with the same input parameters exists in different schemas, Db2 chooses the user-defined function whose schema name is earliest in the SQL path.
   For example, suppose functions SCHEMA1.X and SCHEMA2.X fit a function invocation equally well. Assume that the SQL path is:

   "SCHEMA2", "SYSPROC", "SYSIBM", "SCHEMA1", "SYSFUN"

   Then Db2 chooses function SCHEMA2.X.
   If two or more candidates fit the unqualified function invocation equally well because the function invocation contains parameter markers, Db2 issues an error.

The remainder of this section discusses details of the function resolution process and gives suggestions on how you can ensure that Db2 picks the right function.

How Db2 chooses candidate functions:

An instance of a user-defined function is a candidate for execution only if it meets all of the following criteria:

- If the function name is qualified in the invocation, the schema of the function instance matches the schema in the function invocation.
  If the function name is unqualified in the invocation, the schema of the function instance matches a schema in the invoker’s SQL path.
- The name of the function instance matches the name in the function invocation.
- The number of input parameters in the function instance matches the number of input parameters in the function invocation.
- The function invoker is authorized to execute the function instance.
- The type of each of the input parameters in the function invocation matches or is promotable to the type of the corresponding parameter in the function instance.

Chapter 3. Db2 SQL programming 455
If an input parameter in the function invocation is an untyped parameter marker, Db2 considers that parameter to be a match or promotable.

For a function invocation that passes a transition table, the data type, length, precision, and scale of each column in the transition table must match exactly the data type, length, precision, and scale of each column of the table that is named in the function instance definition. For information about transition tables, see “Creating a trigger” on page 149.

- The create timestamp for a user-defined function must be older than the BIND or REBIND timestamp for the package or plan in which the user-defined function is invoked.

If Db2 authorization checking is in effect, and Db2 performs an automatic rebind on a plan or package that contains a user-defined function invocation, any user-defined functions that were created after the original BIND or REBIND of the invoking plan or package are not candidates for execution.

If you use an access control authorization exit routine, some user-defined functions that were not candidates for execution before the original BIND or REBIND of the invoking plan or package might become candidates for execution during the automatic rebind of the invoking plan or package.

If a user-defined function is invoked during an automatic rebind, and that user-defined function is invoked from a trigger body and receives a transition table, then the form of the invoked function that Db2 uses for function selection includes only the columns of the transition table that existed at the time of the original BIND or REBIND of the package or plan for the invoking program.

During an automatic rebind, Db2 does not consider built-in functions for function resolution if those built-in functions were introduced in a later release of Db2 than the release in which the BIND or REBIND of the invoking plan or package occurred.

When you explicitly bind or rebind a plan or package, the plan or package receives a release dependency marker. When Db2 performs an automatic rebind of a query that contains a function invocation, a built-in function is a candidate for function resolution only if the release dependency marker of the built-in function is the same as or lower than the release dependency marker of the plan or package that contains the function invocation.

**Example:** Suppose that in this statement, the data type of A is SMALLINT:

```sql
SELECT USER1.ADDTWO(A) FROM TABLEA;
```

Two instances of USER1.ADDTWO are defined: one with an input parameter of type INTEGER and one with an input parameter of type DECIMAL. Both function instances are candidates for execution because the SMALLINT type is promotable to either INTEGER or DECIMAL. However, the instance with the INTEGER type is a better fit because INTEGER is higher in the list than DECIMAL.

**How Db2 chooses the best fit among candidate functions:**

More than one function instance might be a candidate for execution. In that case, Db2 determines which function instances are the best fit for the invocation by comparing parameter data types.

If the data types of all parameters in a function instance are the same as those in the function invocation, that function instance is a best fit. If no exact match exists, Db2 compares data types in the parameter lists from left to right, using this method:

1. Db2 compares the data types of the first parameter in the function invocation to the data type of the first parameter in each function instance. If the first parameter in the invocation is an untyped parameter marker, Db2 does not do the comparison.
2. For the first parameter, if one function instance has a data type that fits the function invocation better than the data types in the other instances, that function is a best fit.
3. If the data types of the first parameter are the same for all function instances, or if the first parameter in the function invocation is an untyped parameter marker, Db2 repeats this process for the next parameter. Db2 continues this process for each parameter until it finds a best fit.
**Example of function resolution:** Suppose that a program contains the following statement:

```
SELECT FUNC(VCHARCOL,SMINTCOL,DECCOL) FROM T1;
```

In user-defined function `FUNC`, `VCHARCOL` has data type `VARCHAR`, `SMINTCOL` has data type `SMALLINT`, and `DECCOL` has data type `DECIMAL`. Also suppose that two function instances with the following definitions meet the appropriate criteria and are therefore candidates for execution.

Candidate 1:
```
CREATE FUNCTION FUNC(VARCHAR(20),INTEGER,DOUBLE)  
RETURNS DECIMAL(9,2)  
EXTERNAL NAME 'FUNC1'  
PARAMETER STYLE SQL  
LANGUAGE COBOL;
```

Candidate 2:
```
CREATE FUNCTION FUNC(VARCHAR(20),REAL,DOUBLE)  
RETURNS DECIMAL(9,2)  
EXTERNAL NAME 'FUNC2'  
PARAMETER STYLE SQL  
LANGUAGE COBOL;
```

Db2 compares the data type of the first parameter in the user-defined function invocation to the data types of the first parameters in the candidate functions. Because the first parameter in the invocation has data type `VARCHAR`, and both candidate functions also have data type `VARCHAR`, Db2 cannot determine the better candidate based on the first parameter. Therefore, Db2 compares the data types of the second parameters.

The data type of the second parameter in the invocation is `SMALLINT`. `INTEGER`, which is the data type of candidate 1, is a better fit to `SMALLINT` than `REAL`, which is the data type of candidate 2. Therefore, candidate 1 is the Db2 choice for execution.

**Related concepts**
Promotion of data types (Db2 SQL)

**Related tasks**
Creating a trigger

A **trigger** is a set of SQL statements that execute when a certain event occurs in a table or view. Use triggers to control changes in Db2 databases. Triggers are more powerful than constraints because they can monitor a broader range of changes and perform a broader range of actions. This topic describes support for advanced triggers.

**Related information**
Exit routines (Db2 Administration Guide)

### Checking how Db2 resolves functions by using DSN_FUNCTION_TABLE

Because multiple user-defined functions can have the same name, you should ensure that Db2 invokes the function that you intended to invoke. One way to check that the correct function was invoked is to use a function table called `DSN_FUNCTION_TABLE`.

**Procedure**

To check how Db2 resolves a function by using `DSN_FUNCTION_TABLE`:

1. If your `userID`.DSN_FUNCTION_TABLE does not already exist, create this table by following the instructions in `DSN_FUNCTION_TABLE` (Db2 Performance).

2. Populate your `userID`.DSN_FUNCTION_TABLE with information about which functions are invoked by a particular SQL statement by performing one of the following actions:
   - Execute the EXPLAIN statement on the SQL statement.
   - Ensure that the program that contains the SQL statement is bound with EXPLAIN(YES) and run the program.
Db2 puts a row in your_userID.DSN_FUNCTION_TABLE for each function that is referenced in each SQL statement.

3. Check the rows that were added to your_userID.DSN_FUNCTION_TABLE to ensure that the appropriate function was invoked. Use the following columns to help you find applicable rows: QUERYNO, APPLNAME, PROGNAM, COLLID, and EXPLAIN_TIME.

Related reference
BIND and REBIND options for packages, plans, and services (Db2 Commands)
EXPLAIN (Db2 SQL)

Restrictions when passing arguments with distinct types to functions

Because Db2 enforces strong typing when you pass arguments to a function, you must follow certain rules when passing arguments with distinct types to functions.

Adhere to the following rules:

- You can pass arguments that have distinct types to a function if either of the following conditions is true:
  - A version of the function that accepts those distinct types is defined.
    - This also applies to infix operators. If you want to use one of the five built-in infix operators (||, /, *, +, -) with your distinct types, you must define a version of that operator that accepts the distinct types.
  - You can cast your distinct types to the argument types of the function.
- If you pass arguments to a function that accepts only distinct types, the arguments you pass must have the same distinct types as in the function definition. If the types are different, you must cast your arguments to the distinct types in the function definition.
- If you pass constants or host variables to a function that accepts only distinct types, you must cast the constants or host variables to the distinct types that the function accepts.

The following examples demonstrate how to use distinct types as arguments in function invocations.

**Example: Defining a function with distinct types as arguments:** Suppose that you want to invoke the built-in function HOUR with a distinct type that is defined like this:

```sql
CREATE DISTINCT TYPE FLIGHT_TIME AS TIME;
```

The HOUR function takes only the TIME or TIMESTAMP data type as an argument, so you need a sourced function that is based on the HOUR function that accepts the FLIGHT_TIME data type. You might declare a function like this:

```sql
CREATE FUNCTION HOUR(FLIGHT_TIME)
RETURNS INTEGER
SOURCE SYSIBM.HOUR(TIME);
```

**Example: Casting function arguments to acceptable types:** Another way you can invoke the HOUR function is to cast the argument of type FLIGHT_TIME to the TIME data type before you invoke the HOUR function. Suppose table FLIGHT_INFO contains column DEPARTURE_TIME, which has data type FLIGHT_TIME, and you want to use the HOUR function to extract the hour of departure from the departure time. You can cast DEPARTURE_TIME to the TIME data type, and then invoke the HOUR function:

```sql
SELECT HOUR(CAST(DEPARTURE_TIME AS TIME)) FROM FLIGHT_INFO;
```

**Example: Using an infix operator with distinct type arguments:** Suppose you want to add two values of type US_DOLLAR. Before you can do this, you must define a version of the + function that accepts values of type US_DOLLAR as operands:

```sql
CREATE FUNCTION "+"(US_DOLLAR,US_DOLLAR)
RETURNS US_DOLLAR
SOURCE SYSIBM."+"(DECIMAL(9,2),DECIMAL(9,2));
```
Because the US_DOLLAR type is based on the DECIMAL(9,2) type, the source function must be the version of + with arguments of type DECIMAL(9,2).

**Example: Casting constants and host variables to distinct types to invoke a user-defined function:**
Suppose function CDN_TO_US is defined like this:

```sql
CREATE FUNCTION CDN_TO_US(EURO)
RETURNS US_DOLLAR
EXTERNAL NAME 'CDNCVT'
PARAMETER STYLE SQL
LANGUAGE C;
```

This means that CDN_TO_US accepts only the EURO type as input. Therefore, if you want to call CDN_TO_US with a constant or host variable argument, you must cast that argument to distinct type EURO:

```sql
SELECT * FROM US_SALES
WHERE TOTAL = CDN_TO_US(EURO(:H1));
```

```sql
SELECT * FROM US_SALES
WHERE TOTAL = CDN_TO_US(EURO(10000));
```

**Cases when Db2 casts arguments for a user-defined function**

In certain situations, when you invoke a user-defined function, Db2 casts your input argument values to different data types and lengths.

Whenever you invoke a user-defined function, Db2 assigns your input argument values to parameters with the data types and lengths in the user-defined function definition.

When you invoke a user-defined function that is sourced on another function, Db2 casts your arguments to the data types and lengths of the sourced function.

The following example demonstrates what happens when the parameter definitions of a sourced function differ from those of the function on which it is sourced.

Suppose that external user-defined function TAXFN1 is defined like this:

```sql
CREATE FUNCTION TAXFN1(DEC(6,0))
RETURNS DEC(5,2)
PARAMETER STYLE SQL
LANGUAGE C
EXTERNAL NAME TAXPROG;
```

Sourced user-defined function TAXFN2, which is sourced on TAXFN1, is defined like this:

```sql
CREATE FUNCTION TAXFN2(DEC(8,2))
RETURNS DEC(5,0)
SOURCE TAXFN1;
```

You invoke TAXFN2 using this SQL statement:

```sql
UPDATE TB1
SET SALESTAX2 = TAXFN2(PRICE2);
```

TB1 is defined like this:

```sql
CREATE TABLE TB1
(PRICE1 DEC(6,0),
SALESTAX1 DEC(5,2),
PRICE2 DEC(9,2),
SALESTAX2 DEC(7,2));
```

Now suppose that PRICE2 has the DECIMAL(9,2) value 0001234.56. Db2 must first assign this value to the data type of the input parameter in the definition of TAXFN2, which is DECIMAL(8,2). The input parameter value then becomes 001234.56. Next, Db2 casts the parameter value to a source function...
parameter, which is DECIMAL(6,0). The parameter value then becomes 001234. (When you cast a value, that value is truncated, rather than rounded.)

Now, if TAXFN1 returns the DECIMAL(5,2) value 123.45, Db2 casts the value to DECIMAL(5,0), which is the result type for TAXFN2, and the value becomes 00123. This is the value that Db2 assigns to column SALESTAX2 in the UPDATE statement.

**Casting of parameter markers**

You can use untyped parameter markers in a function invocation. However, Db2 cannot compare the data types of untyped parameter markers to the data types of candidate functions. Therefore, Db2 might find more than one function that qualifies for invocation. If this happens, an SQL error occurs. To ensure that Db2 picks the right function to execute, cast the parameter markers in your function invocation to the data types of the parameters in the function that you want to execute. For example, suppose that two versions of function FX exist. One version of FX is defined with a parameter of type of DECIMAL(9,2), and the other is defined with a parameter of type INTEGER. You want to invoke FX with a parameter marker, and you want Db2 to execute the version of FX that has a DECIMAL(9,2) parameter. You need to cast the parameter marker to a DECIMAL(9,2) type by using a CAST specification:

```sql
SELECT FX(CAST(?) AS DECIMAL(9,2))) FROM T1;
```

**Related concepts**

Assignment and comparison (Db2 SQL)
Chapter 4. Embedded SQL programming

Application programs written in host languages such as COBOL can contain SQL statements. The source form of a static SQL statement is embedded within application program, and the statement is prepared before the program is executed and the operational form of the statement persists beyond the execution of the program.

An application program can also contain dynamic SQL statements referred to as embedded dynamic SQL. Programs that contain embedded dynamic SQL statements must be precompiled like those that contain static SQL, but unlike static SQL, the dynamic statements are constructed and prepared at run time. The source form of a dynamic statement is a character string that is passed to Db2 by the program using the static SQL PREPARE or EXECUTE IMMEDIATE statement. A statement that is prepared using the PREPARE statement can be referenced in a DECLARE CURSOR, DESCRIBE, or EXECUTE statement. Whether the operational form of the statement is persistent depends on whether dynamic statement caching is enabled.

Overview of programming applications that access Db2 for z/OS data

Applications that interact with Db2 must first connect to Db2. They can then read, add, or modify data or manipulate Db2 objects.

About this task

A query is an SQL statement that returns data from a Db2 database. Your program can use several methods to communicate SQL statements to Db2 for z/OS. After processing the statement, Db2 issues a return code, which your program can test to determine the result of the operation.

Introductory concepts

Programming for Db2 for z/OS (Introduction to Db2 for z/OS)
Tools and IDEs for developing Db2 applications (Introduction to Db2 for z/OS)
Preparation process for an application program (Introduction to Db2 for z/OS)
Performance information for SQL application programming (Introduction to Db2 for z/OS)

Procedure

To include Db2 for z/OS queries in an application program:

1. Choose one of the following methods for communicating with Db2:

   Static SQL
   - The source form of a static SQL statement is embedded within an application program written in a host language. The statement is prepared before the program is executed and the operational form of the statement persists beyond the execution of the program.

   Embedded dynamic SQL
   - Dynamic SQL is prepared and executed while the program is running.

   Open Database Connectivity (ODBC)
   - You access data through ODBC function calls in your application. You execute SQL statements by passing them to Db2 through a ODBC function call. ODBC eliminates the need for precompiling and binding your application and increases the portability of your application by using the ODBC interface.

   JDBC application support
   - If you are writing your applications in Java, you can use JDBC application support to access Db2. JDBC is similar to ODBC but is designed specifically for use with Java.
SQLJ application support
You also can use SQLJ application support to access Db2. SQLJ is designed to simplify the coding of Db2 calls for Java applications.

Db2 for Linux, UNIX, and Windows drivers
You can use the client drivers to connect to Db2 for z/OS from application programming languages such as Node.js, Perl, Python, Ruby on Rails, PHP, and others.

2. Optional: Declare the tables and views that you use.
You can use DCLGEN to generate these declarations.

3. Define the items that your program can use to check whether an SQL statement executed successfully. You can either define an SQL communications area (SQLCA) or declare SQLSTATE and SQLCODE host variables.

4. Define at least one SQL descriptor area (SQLDA).

5. Declare any of the following data items for passing data between Db2 and a host language:
   - “Host variables” on page 477
   - “Host-variable arrays” on page 478
   - “Host structures” on page 479

   Ensure that you use the appropriate data types. For details, see “Compatibility of SQL and language data types” on page 484

6. Code SQL statements to access Db2 data. Make sure to delimit the statements correctly for the specific programming language.
   For more information about coding SQL statements in host languages, see the language-specific information for your programming language:
   • Assembler
   • C and C++
   • COBOL
   • Fortran
   • Java
   • ODBC
   • PL/I
   • REXX

   Consider using cursors to select a set of rows and then process the set either one row at a time or one rowset at a time.

7. Check the execution of the SQL statements.
8. Handle any SQL error codes.

What to do next
“Writing applications that enable users to create and modify tables” on page 540
“Saving SQL statements that are translated from user requests” on page 541

Related concepts
Example programs that call stored procedures
Examples can be used as models when you write applications that call stored procedures. In addition, prefix.SDSNSAMP contains sample jobs DSNTEJ6P and DSNTEJ6S and programs DSN8EP1 and DSN8EP2, which you can run.

XML data in embedded SQL applications (Db2 Programming for XML)
Introduction to Db2 ODBC (Db2 Programming for ODBC)
JDBC application programming (Db2 Application Programming for Java)
SQLJ application programming (Db2 Application Programming for Java)
Declaring table and view definitions

Before your program issues SQL statements that select, insert, update, or delete data, the program needs to declare the tables and views that those statements access.

About this task

Your program is not required to declare tables or views, but doing so offers the following advantages:

- **Clear documentation in the program**
  The declaration specifies the structure of the table or view and the data type of each column. You can refer to the declaration for the column names and data types in the table or view.

- **Assurance that your program uses the correct column names and data types**
  The Db2 precompiler uses your declarations to make sure that you have used correct column names and data types in your SQL statements. The Db2 precompiler issues a warning message when the column names and data types in SQL statements do not correspond to the table and view declarations in your program.

Procedure

To declare table and view definitions, use one of the following methods:

- Include an SQL DECLARE TABLE statement in your program. Specify the name of the table or view and list each column and its data type.

  When you declare a table or view that contains a column with a distinct type, declare that column with the source type of the distinct type rather than with the distinct type itself. When you declare the column with the source type, Db2 can check embedded SQL statements that reference that column at precompile time.

  In a COBOL program, code the DECLARE TABLE statement in the WORKING-STORAGE SECTION or LINKAGE SECTION within the DATA DIVISION.

  For example, the following DECLARE TABLE statement in a COBOL program defines the DSN8C10.DEPT table:

  ```sql
  EXEC SQL
  DECLARE DSN8C10.DEPT TABLE
  (DEPTNO    CHAR(3)           NOT NULL,
   DEPTNAME  VARCHAR(36)       NOT NULL,
   MGRNO     CHAR(6)                   ,
   ADMRDEPT  CHAR(3)           NOT NULL,
   LOCATION  CHAR(16)                  )
END-EXEC.
  ```

- Use DCLGEN, the declarations generator that is supplied with Db2, to create these declarations for you and then include them in your program.

  **Restriction:** You can use DCLGEN for only C, COBOL, and PL/I programs.

Related concepts

DCLGEN (declarations generator)
Your program should declare the tables and views that it accesses. The Db2 declarations generator, DCLGEN, produces these DECLARE statements for C, COBOL, and PL/I programs, so that you do not need to code the statements yourself. DCLGEN also generates corresponding host variable structures.

**Related reference**

DECLARE TABLE (Db2 SQL)
DCLGEN (DECLARATIONS GENERATOR) (DSN) (Db2 Commands)

**DCLGEN (declarations generator)**

Your program should declare the tables and views that it accesses. The Db2 declarations generator, DCLGEN, produces these DECLARE statements for C, COBOL, and PL/I programs, so that you do not need to code the statements yourself. DCLGEN also generates corresponding host variable structures.

DCLGEN generates a table or view declaration and puts it into a member of a partitioned data set that you can include in your program. When you use DCLGEN to generate a table declaration, Db2 gets the relevant information from the Db2 catalog. The catalog contains information about the table or view definition and the definition of each column within the table or view. DCLGEN uses this information to produce an SQL DECLARE TABLE statement for the table or view and a corresponding PL/I or C structure declaration or COBOL record description.

**Related reference**

DCLGEN (DECLARATIONS GENERATOR) (DSN) (Db2 Commands)

**Generating table and view declarations by using DCLGEN**

Your program should declare the tables and views that it accesses. For C, COBOL, and PL/I programs, you can use DCLGEN to produce these declarations, so that you do not need to code the statements yourself. DCLGEN also generates corresponding host variable structures.

**Before you begin**

**Requirements:**

- Db2 must be active before you can use DCLGEN.
- You can use DCLGEN for table declarations only if the table or view that you are declaring already exists.
- If you use DCLGEN, you must use it before you precompile your program.

**Procedure**

To generate table and view declarations by using DCLGEN:

1. Invoke DCLGEN by performing one of the following actions:

   - **To start DCLGEN from ISPF through DB2I:** Select the DCLGEN option on the DB2I Primary Option Menu panel. Then follow the detailed instructions for generating table and view declarations by using DCLGEN from DB2I.
   - **To start DCLGEN directly from TSO:** Sign on to TSO, issue the TSO command DSN, and then issue the subcommand DCLGEN.
   - **To start DCLGEN directly from a CLIST:** From a CLIST, running in TSO foreground or background, issue DSN and then DCLGEN.
   - **To start DCLGEN with JCL:** Supply the required information in JCL and run DCLGEN in batch. Use the sample jobs DSNTEJ2C and DSNTEJ2P in the prefix.SDSNSAMP library as models.

   **Requirement:** If you want to start DCLGEN in the foreground and your table names include DBCS characters, you must provide and display double-byte characters. If you do not have a terminal that displays DBCS characters, you can enter DBCS characters by using the hex mode of ISPF edit.

DCLGEN creates the declarations in the specified data set.
DCLGEN generates a table or column name in the DECLARE statement as a non-delimited identifier unless at least one of the following conditions is true:

- The name contains special characters and is not a DBCS string.
- The name is a DBCS string, and you have requested delimited DBCS names.

2. If you use an SQL reserved word as an identifier, edit the DCLGEN output to add the appropriate SQL delimiters.

3. Make any other necessary edits to the DCLGEN output.

DCLGEN produces output that is intended to meet the needs of most users, but occasionally, you need to edit the DCLGEN output to work in your specific case. For example, DCLGEN is unable to determine whether a column that is defined as NOT NULL also contains the DEFAULT clause, so you must edit the DCLGEN output to add the DEFAULT clause to the appropriate column definitions.

DCLGEN produces declarations based on the encoding scheme of the source table. Therefore, if your application uses a different encoding scheme, you might need to manually adjust the declarations. For example, if your source table is in EBCDIC with CHAR columns and your application is in COBOL, DCLGEN produces declarations of type PIC X. However, suppose your host variables in your COBOL application are UTF-16. In this case, you will need to manually change the declarations to be type PIC N USAGE NATIONAL.

Related reference
DCLGEN (DECLARATIONS GENERATOR) (DSN) (Db2 Commands)
DSN (TSO) (Db2 Commands)
Reserved words (Db2 SQL)

Generating table and view declarations by using DCLGEN from DB2I

DCLGEN generates table and view declarations and the corresponding variable declarations for C, COBOL, and PL/I programs so that you do not need to code these statements yourself. The easiest way to start DCLGEN is through DB2I.

Procedure

To generate table and view declarations by using DCLGEN from DB2I:
1. From the DB2I Primary Option Menu panel, select the DCLGEN option.
   The following DCLGEN panel is displayed:

<table>
<thead>
<tr>
<th>DSNEPD01</th>
<th>DCLGEN</th>
<th>SSID: DSN</th>
</tr>
</thead>
<tbody>
<tr>
<td>==&gt;</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

   Enter table name for which declarations are required:
   1  SOURCE TABLE NAME ==>  
   2  TABLE OWNER ..... ==>  
   3  AT LOCATION ..... ==> (Optional)
   Enter destination data set: (Can be sequential or partitioned)
   4  DATA SET NAME ... ==>  
   5  DATA SET PASSWORD ==> (If password protected)
   Enter options as desired:
   6  ACTION ............ ==> ADD (ADD new or REPLACE old declaration)
   7  COLUMN LABEL .... ==> NO (Enter YES for column label)
   8  STRUCTURE NAME .. ==> (Optional)
   9  FIELD NAME PREFIX ==> (Optional)
   10 DELIMIT DBCS .... ==> YES (Enter YES to delimit DBCS identifiers)
   11 COLUMN SUFFIX ... ==> NO (Enter YES to append column name)
   12 INDICATOR VARS .. ==> NO (Enter YES for indicator variables)
   13 ADDITIONAL OPTIONS==> YES (Enter YES to change additional options)

   PRESS: ENTER to process  END to exit  HELP for more information

   Figure 23. DCLGEN panel

   2. Complete the following fields on the DCLGEN panel:
1 SOURCE TABLE NAME
Is the unqualified name of the table, view, or created temporary table for which you want DCLGEN to produce SQL data declarations. The table can be stored at your Db2 location or at another Db2 location. To specify a table name at another Db2 location, enter the table qualifier in the TABLE OWNER field and the location name in the AT LOCATION field. DCLGEN generates a three-part table name from the SOURCE TABLE NAME, TABLE OWNER, and AT LOCATION fields. You can also use an alias for a table name.

To specify a table name that contains special characters or blanks, enclose the name in apostrophes. If the name contains apostrophes, you must double each one(' '). For example, to specify a table named DON’S TABLE, enter the following text:

```
'DON’S TABLE'
```

The underscore is not handled as a special character in DCLGEN. For example, the table name JUNE_PROFITS does not need to be enclosed in apostrophes. Because COBOL field names cannot contain underscores, DCLGEN substitutes hyphens (-) for single-byte underscores in COBOL field names that are built from the table name.

You do not need to enclose DBCS table names in apostrophes.

If you do not enclose the table name in apostrophes, Db2 converts lowercase characters to uppercase.

2 TABLE OWNER
Is the schema qualifier of the source table. If you do not specify this value and the table is a local table, Db2 assumes that the table qualifier is your TSO logon ID. If the table is at a remote location, you must specify this value.

3 AT LOCATION
Is the location of a table or view at another Db2 subsystem. The value of the AT LOCATION field becomes a prefix for the table name on the SQL DECLARE statement, as follows: `location_name, schema_name, table_name` For example, if the location name is PLAINS_GA, the schema name is CARTER, and the table name is CROP_YIELD_89, the following table name is included in the SQL DECLARE statement: PLAINS_GA.CARTER.CROP_YIELD_89

The default is the local location name. This field applies to Db2 private protocol access only. The location must be another Db2 for z/OS subsystem.

4 DATA SET NAME
Is the name of the data set that you allocated to contain the declarations that DCLGEN produces. You must supply a name; no default exists.

The data set must already exist and be accessible to DCLGEN. The data set can be either sequential or partitioned. If you do not enclose the data set name in apostrophes, DCLGEN adds a standard TSO prefix (user ID) and suffix (language). DCLGEN determines the host language from the DB2I defaults panel.

For example, for library name LIBNAME(MEMBNAME), the name becomes `userid.libname.language(membname)` For library name LIBNAME, the name becomes `userid.libname.language`.

If this data set is password protected, you must supply the password in the DATA SET PASSWORD field.

5 DATA SET PASSWORD
Is the password for the data set that is specified in the DATA SET NAME field, if the data set is password protected. The password is not displayed on your terminal, and it is not recognized if you issued it from a previous session.

6 ACTION
Specifies what DCLGEN is to do with the output when it is sent to a partitioned data set. (The option is ignored if the data set you specify in the DATA SET NAME field is sequential.) You can specify one of the following values:
ADD
Indicates that an old version of the output does not exist and creates a new member with the
specified data set name. ADD is the default.

REPLACE
Replaces an old version, if it already exists. If the member does not exist, this option creates a
new member.

7 COLUMN LABEL
Specifies whether DCLGEN is to include labels that are declared on any columns of the table or
view as comments in the data declarations. (The SQL LABEL statement creates column labels to
use as supplements to column names.) You can specify one of the following values:

YES
Include column labels.

NO
Ignore column labels. NO is the default.

8 STRUCTURE NAME
Is the name of the generated data structure. The name can be up to 31 characters. If the name
is not a DBCS string, and the first character is not alphabetic, enclose the name in apostrophes. If
you use special characters, be careful to avoid name conflicts.

If you leave this field blank, DCLGEN generates a name that contains the table or view name with
a prefix of DCL. If the language is COBOL or PL/I and the table or view name consists of a DBCS
string, the prefix consists of DBCS characters.

For C, lowercase characters that you enter in this field are not converted to uppercase.

9 FIELD NAME PREFIX
Specifies a prefix that DCLGEN uses to form field names in the output. For example, if you choose
ABCDE, the field names generated are ABCDE1, ABCDE2, and so on.

You can specify a field name prefix of up to 28 bytes that can include special and double-byte
characters. If you specify a single-byte or mixed-string prefix and the first character is not
alphabetic, enclose the prefix in apostrophes. If you use special characters, be careful to avoid
name conflicts.

For COBOL and PL/I, if the name is a DBCS string, DCLGEN generates DBCS equivalents of the
suffix numbers.

For C, lowercase characters that you enter in this field do not converted to uppercase.

If you leave this field blank, the field names are the same as the column names in the table or view.

10 DELIMIT DBCS
Specifies whether DCLGEN is to delimit DBCS table names and column names in the table
declaration. You can specify one of the following values:

YES
Specifies that DCLGEN is to enclose the DBCS table and column names with SQL delimiters.

NO
Specifies that DCLGEN is not to delimit the DBCS table and column names.

11 COLUMN SUFFIX
Specifies whether DCLGEN is to form field names by attaching the column name as a suffix to the
value that you specify in FIELD NAME PREFIX. You can specify one of the following values:

YES
Specifies that DCLGEN is to use the column name as a suffix. For example, if you specify YES,
the field name prefix is NEW, and the column name is EMPNO, the field name is NEWEMPNO.

If you specify YES, you must also enter a value in FIELD NAME PREFIX. If you do not enter a
field name prefix, DCLGEN issues a warning message and uses the column names as the field
names.
NO

Specifies that DCLGEN is not to use the column name as a suffix. The default is NO.

12 INDICATOR VARS

Specifies whether DCLGEN is to generate an array of indicator variables for the host variable structure. You can specify one of the following values:

YES

Specifies that DCLGEN is to generate an array of indicator variables for the host variable structure.

If you specify YES, the array name is the table name with a prefix of I (or DBCS letter <I> if the table name consists solely of double-byte characters). The form of the data declaration depends on the language, as shown in the following table. \( n \) is the number of columns in the table.

<table>
<thead>
<tr>
<th>Language</th>
<th>Declaration form</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>short int Itable-name[n];</td>
</tr>
<tr>
<td>COBOL</td>
<td>01 Itable-name PIC S9(4) USAGE COMP OCCURS n TIMES.</td>
</tr>
<tr>
<td>PL/I</td>
<td>DCL Itable-name(n) BIN FIXED(15);</td>
</tr>
</tbody>
</table>

For example, suppose that you define the following table:

CREATE TABLE HASNULLS (CHARCOL1 CHAR(1), CHARCOL2 CHAR(1));

If you request an array of indicator variables for a COBOL program, DCLGEN might generate the following host variable declaration:

01 DCLHASNULLS.
10 CHARCOL1        PIC X(1).
10 CHARCOL2        PIC X(1).
01 IHASNULLS PIC S9(4) USAGE COMP OCCURS 2 TIMES.

NO

Specifies that DCLGEN is not to generate an array of indicator variables. The default is NO.

13 ADDITIONAL OPTIONS

Indicates whether to display the panel for additional DCLGEN options, including the break point for statement tokens and whether to generate DECLARE VARIABLE statements for FOR BIT DATA columns. You can specify YES or NO. The default is YES.

If you specified YES in the ADDITIONAL OPTIONS field, the following ADDITIONAL DCLGEN OPTIONS panel is displayed:

```
DSNEDP02 ADDITIONAL DCLGEN OPTIONS SSID: DSN

Enter options as desired:
1 RIGHT MARGIN .... ===> 72 (Enter 72 or 80)
2 FOR BIT DATA .... ===> NO (Enter YES to declare SQL variables for FOR BIT DATA columns)

PRESS: ENTER to process END to exit HELP for more information
```

Figure 24. ADDITIONAL DCLGEN OPTIONS panel

Otherwise, DCLGEN creates the declarations in the specified data set.

3. If the ADDITIONAL DCLGEN OPTIONS panel is displayed, complete the following fields on that panel:
1 RIGHT MARGIN
Specifications the break point for statement tokens that must be wrapped to one or more subsequent records. You can specify column 72 or column 80.

The default is 72.

2 FOR BIT DATA
Specifies whether DCLGEN is to generate a DECLARE VARIABLE statement for SQL variables for columns that are declared as FOR BIT DATA. This statement is required in Db2 applications that meet all of the following criteria:

- are written in COBOL
- have host variables for FOR BIT DATA columns
- are prepared with the SQLCCSID option of the Db2 coprocessor.

You can specify YES or NO. The default is NO.

If the table or view does not have FOR BIT DATA columns, DCLGEN does not generate this statement.

DCLGEN creates the declarations in the specified data set.

Related reference
The DB2I primary option menu (Introduction to Db2 for z/OS)
LABEL (Db2 SQL)

Data types that DCLGEN uses for variable declarations
DCLGEN produces declarations for tables and views and the corresponding host variable structures for C, COBOL, and PL/I programs. DCLGEN derives the variable names and data types for these declarations based on the source tables in the database.

The following table lists the C, COBOL, and PL/I data types that DCLGEN uses for variable declarations based on the corresponding SQL data types that are used in the source tables. var represents a variable name that DCLGEN provides.

<table>
<thead>
<tr>
<th>SQL data type¹</th>
<th>C</th>
<th>COBOL</th>
<th>PL/I</th>
</tr>
</thead>
<tbody>
<tr>
<td>SMALLINT</td>
<td>short int</td>
<td>PIC S9(4) USAGE COMP-5</td>
<td>BIN FIXED(15)</td>
</tr>
<tr>
<td>INTEGER</td>
<td>long int</td>
<td>PIC S9(9) USAGE COMP-5</td>
<td>BIN FIXED(31)</td>
</tr>
<tr>
<td>BIGINT</td>
<td>long long int</td>
<td>PIC S9(18) USAGE COMP-5</td>
<td>FIXED BIN(63)</td>
</tr>
<tr>
<td>DECIMAL(p,s) or NUMERIC(p,s)</td>
<td>decimal(p,s)²</td>
<td>PIC S9(p-s)V9(s) USAGE COMP-3</td>
<td>DEC FIXED(p,s)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>REAL or FLOAT(n) 1 &lt;= n &lt;= 21</td>
<td>float</td>
<td>USAGE COMP-1</td>
<td>BIN FLOAT(n)</td>
</tr>
<tr>
<td>DOUBLE PRECISION, DOUBLE, or FLOAT(n)</td>
<td>double</td>
<td>USAGE COMP-2</td>
<td>BIN FLOAT(n)</td>
</tr>
</tbody>
</table>

If p>15, the PL/I compiler must support this precision, or a warning is generated.
### Table 86. Type declarations that DCLGEN generates (continued)

<table>
<thead>
<tr>
<th>SQL data type¹</th>
<th>C</th>
<th>COBOL</th>
<th>PL/I</th>
</tr>
</thead>
<tbody>
<tr>
<td>CHAR(1)</td>
<td>char</td>
<td>PIC X(1)</td>
<td>CHAR(1)</td>
</tr>
<tr>
<td>CHAR(n)</td>
<td>char var [n+1]</td>
<td>PIC X(n)</td>
<td>CHAR(n)</td>
</tr>
<tr>
<td>VARCHAR(n)</td>
<td>struct</td>
<td>var [n+1];</td>
<td>CHAR(n) VAR</td>
</tr>
<tr>
<td></td>
<td>{short int var_len;</td>
<td>char var_data[n];</td>
<td></td>
</tr>
<tr>
<td></td>
<td>char</td>
<td>} var;</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>10 var.</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>49 var_LEN PIC 9(4)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>USAGE COMP-5.</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>49 var_TEXT PIC X(n).</td>
<td></td>
</tr>
<tr>
<td>CLOB(n)³</td>
<td>SQL TYPE IS</td>
<td>USAGE SQL TYPE IS</td>
<td>SQL TYPE IS</td>
</tr>
<tr>
<td></td>
<td>CLOB_LOCATOR</td>
<td>CLOB_LOCATOR</td>
<td>CLOB_LOCATOR</td>
</tr>
<tr>
<td>GRAPHIC(1)</td>
<td>sqldbchar</td>
<td>PIC G(1)</td>
<td>GRAPHIC(1)</td>
</tr>
<tr>
<td>GRAPHIC(n)</td>
<td>sqldbchar var[n+1];</td>
<td>PIC G(n) USAGE</td>
<td></td>
</tr>
<tr>
<td>n &gt; 1</td>
<td></td>
<td>DISPLAY-1.⁴</td>
<td>or PIC N(n).⁴</td>
</tr>
<tr>
<td>VARGRAPHIC(n)</td>
<td>struct VARGRAPH</td>
<td>{short len;</td>
<td>GRAPHIC(n) VAR</td>
</tr>
<tr>
<td></td>
<td></td>
<td>sqldbchar data[n];</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>} var;</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>10 var.</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>49 var_LEN PIC 9(4)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>USAGE COMP-5.</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>49 var_TEXT PIC G(n)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>or</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>10 var.</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>49 var_LEN PIC 9(4)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>USAGE DISPLAY-1.⁴</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>or PIC N(n).⁴</td>
<td></td>
</tr>
<tr>
<td>DBCLOB(n)³</td>
<td>SQL TYPE IS</td>
<td>USAGE SQL TYPE IS</td>
<td>SQL TYPE IS</td>
</tr>
<tr>
<td></td>
<td>DBCLOB_LOCATOR</td>
<td>DBCLOB_LOCATOR</td>
<td>DBCLOB_LOCATOR</td>
</tr>
<tr>
<td>BINARY(n)</td>
<td>SQL TYPE IS</td>
<td>USAGE SQL TYPE IS</td>
<td>SQL TYPE IS</td>
</tr>
<tr>
<td></td>
<td>BINARY(n)</td>
<td>BINARY(n)</td>
<td>BINARY(n)</td>
</tr>
<tr>
<td>VARBINARY(n)</td>
<td>SQL TYPE IS</td>
<td>USAGE SQL TYPE IS</td>
<td>SQL TYPE IS</td>
</tr>
<tr>
<td></td>
<td>VARBINARY(n)</td>
<td>VARBINARY(n)</td>
<td>VARBINARY(n)</td>
</tr>
<tr>
<td>BLOB(n)³</td>
<td>SQL TYPE IS</td>
<td>USAGE SQL TYPE IS</td>
<td>SQL TYPE IS</td>
</tr>
<tr>
<td></td>
<td>BLOB_LOCATOR</td>
<td>BLOB_LOCATOR</td>
<td>BLOB_LOCATOR</td>
</tr>
<tr>
<td>DATE</td>
<td>char var[11]⁵</td>
<td>PIC X(10)⁵</td>
<td>CHAR(10)⁵</td>
</tr>
<tr>
<td>TIME</td>
<td>char var[9]⁶</td>
<td>PIC X(8)⁶</td>
<td>CHAR(8)⁶</td>
</tr>
<tr>
<td>TIMESTAMP</td>
<td>char var[27]</td>
<td>PIC X(26)</td>
<td>CHAR(26)</td>
</tr>
<tr>
<td>TIMESTAMP(0)</td>
<td>char var[20]</td>
<td>PIC X(19)</td>
<td>CHAR(19)</td>
</tr>
<tr>
<td>TIMESTAMP(p) p &gt; 0</td>
<td>char var[21+p]</td>
<td>PIC X(20+p)</td>
<td>CHAR(20+p)</td>
</tr>
<tr>
<td>TIMESTAMP(0) WITH TIME ZONE</td>
<td>struct {short int var_len;</td>
<td>var [147];</td>
<td>DCL var CHAR(147) VAR;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>char var_data[147];</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>} var;</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>01 var.</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>49 var_LEN PIC S9(4)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>COMP-5.</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>49 var_TEXT PIC X(147).</td>
<td></td>
</tr>
</tbody>
</table>
Table 86. Type declarations that DCLGEN generates (continued)

<table>
<thead>
<tr>
<th>SQL data type</th>
<th>C</th>
<th>COBOL</th>
<th>PL/I</th>
</tr>
</thead>
<tbody>
<tr>
<td>TIMESTAMP(p) WITH TIME ZONE</td>
<td>struct short int var_len;</td>
<td>01 var.</td>
<td>DCL var CHAR(148 + p) VAR;</td>
</tr>
<tr>
<td></td>
<td>char var_data[148 + p];</td>
<td>49 var_LEN PIC S9(4) COMP-5.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>} var;</td>
<td>49 var_TEXT PIC X(148 + p).</td>
<td></td>
</tr>
<tr>
<td>ROWID</td>
<td>SQL TYPE IS ROWID</td>
<td>SQL TYPE IS ROWID</td>
<td>SQL TYPE IS ROWID</td>
</tr>
<tr>
<td>XML</td>
<td>SQL TYPE IS XML AS CLOB(1M)</td>
<td>SQL TYPE IS XML AS CLOB(1M)</td>
<td>SQL TYPE IS XML AS CLOB(1M)</td>
</tr>
</tbody>
</table>

Notes:

1. For a distinct type, DCLGEN generates the host language equivalent of the source data type.
2. If your C compiler does not support the decimal data type, edit your DCLGEN output and replace the decimal data declarations with declarations of type double.
3. For a BLOB, CLOB, or DBCLOB data type, DCLGEN generates a LOB locator.
4. DCLGEN chooses the format based on the character that you specify as the DBCS symbol on the COBOL Defaults panel.
5. This declaration is used unless a date installation exit routine exists for formatting dates, in which case the length is that specified for the LOCAL DATE LENGTH installation option.
6. This declaration is used unless a time installation exit routine exists for formatting times, in which case the length is that specified for the LOCAL TIME LENGTH installation option.
7. The default setting for XML is 1M; however, you might need to adjust it.

Including declarations from DCLGEN in your program

After you use DCLGEN to produce declarations for tables, views, and variables for your C, COBOL, or PL/I program, you should include these declarations in your program.

Before you begin

Recommendation: To ensure that your program uses a current description of the table, use DCLGEN to generate the table's declaration and store it as a member in a library (usually a partitioned data set) just before you precompile the program.

Procedure

Code the following SQL INCLUDE statement in your program:

```sql
EXEC SQL
INCLUDE member-name
END-EXEC.
```

*member-name* is the name of the data set member where the DCLGEN output is stored.

Example

Suppose that you used DCLGEN to generate a table declaration and corresponding COBOL record description for the table DSN8C10.EMP, and those declarations were stored in the data set member DECEMP. (A COBOL record description is a two-level host structure that corresponds to the columns of
To include those declarations in your program, include the following statement in your COBOL program:

```
EXEC SQL
INCLUDE DECEMP
END-EXEC.
```

**Related reference**

INCLUDE (Db2 SQL)

---

### Example: Adding DCLGEN declarations to a library

You can use DCLGEN to generate table and variable declarations for C, COBOL, and PL/I programs. If you store these declarations in a library, you can later integrate them into your program with a single SQL INCLUDE statement.

This example adds a table declaration and a corresponding host-variable structure to a library. This example is based on the following scenario:

- The library name is `prefix.TEMP.COBOL`.
- The member is a new member named `VPHONE`.
- The table is a local table named `DSN8C10.VPHONE`.
- The host-variable structure is for COBOL.
- The structure receives the default name `DCLVPHONE`.

Throughout this example, information that you must enter on each panel is in bold-faced type.

In this scenario, to add a table declaration and a corresponding host variable structure for `DSN8C10.VPHONE` to the library `prefix.TEMP.COBOL`, complete the following steps:

1. Specify COBOL as the host language by completing the following actions:
   a. On the ISPF/PDF menu, select option D to display the DB2I DEFAULTS PANEL 1 panel.
   b. Specify `IBMCOB` as the application language, as shown in the following figure and press Enter.

```
DSNEOP01                   DB2I DEFAULTS PANEL 1
COMMAND ===> _
Change defaults as desired:
1  DB2 NAME ............. ===> DSN (Subsystem identifier)
2  DB2 CONNECTION RETRIES ===> 0 (How many retries for DB2 connection)
3  APPLICATION LANGUAGE ====> IBMCOB (ASM, C, CPP, IBMCOB, FORTRAN, PLI)
4  LINES/PAGE OF LISTING ===> 80 (A number from 5 to 999)
5  MESSAGE LEVEL ......... ===> I (Information, Warning, Error, Severe)
6  SQL STRING DELIMITER ` ===> DEFAULT (DEFAULT, ' or ')
7  DECIMAL POINT ......... ===> . (. or ,)
8  STOP IF RETURN CODE >= ====> 8 (Lowest terminating return code)
9  NUMBER OF ROWS ===> 20 (For ISPF Tables)
10 CHANGE HELP BOOK NAMES? ===> NO (YES to change HELP data set names)
11 AS USER ===> (Userid to associate with the trusted connection)

PRESS: ENTER to process      END to cancel      HELP for more information
```

*Figure 25. DB2I defaults panel—changing the application language*

The DB2I DEFAULTS PANEL 2 panel for COBOL is then displayed.

2. Complete the DB2I DEFAULTS PANEL 2 panel, shown in the following figure, as needed and press Enter to save the new defaults, if any.
Change defaults as desired:

1. **DB2I JOB STATEMENT**: (Optional if your site has a SUBMIT exit)

   ```
   ===> //ADMF001A JOB (ACCOUNT), 'NAME'
   ===> //**
   ===> //**
   ===> //**
   ```

2. **COBOL DEFAULTS**: (For IBMCOB)

   2. **COBOL STRING DELIMITER ===> ** DEFAULT (DEFAULT, ' or ')
   3. **DBC5 SYMBOL FOR DCLGEN ===> G** (G/N - Character in PIC clause)

**Figure 26. The COBOL defaults panel**

The DB2I Primary Option menu is displayed.

2. Generate the table and host structure declarations by completing the following actions:

   a. On the DB2I Primary Option menu, select the DCLGEN option and press Enter to display the DCLGEN panel.

   b. Complete the fields as shown in the following figure and press Enter.

   ```
   DSNEDP01                  DCLGEN                             SSID: DSN
   ===>
   Enter table name for which declarations are required:
   1  SOURCE TABLE NAME ===>
   DSN8C10.VPHONE
   2  TABLE OWNER ..... ===>
   (Optional)
   Enter destination data set: (Can be sequential or partitioned)
   4  DATA SET NAME ... ===>
   TEMP(VPHONEC)
   5  DATA SET PASSWORD ===> (If password protected)
   Enter options as desired:
   6  ACTION ............ ===> ADD (ADD new or REPLACE old declaration)
   7  COLUMN LABEL .... ===> NO (Enter YES for column label)
   8  STRUCTURE NAME .. ===> (Optional)
   9  FIELD NAME PREFIX ===> (Optional)
   10  DELIMIT DBCS ...... ===> YES (Enter YES to delimit DBCS identifiers)
   11  COLUMN SUFFIX ... ===> NO (Enter YES to append column name)
   12  INDICATOR VARS .. ===> NO (Enter YES for indicator variables)
   13  ADDITIONAL OPTIONS===>
   (Enter YES to change additional options)
   ```

   **Figure 27. DCLGEN panel—selecting source table and destination data set**

   A successful completion message, such as the one in the following figure, is displayed at the top of your screen.

   ```
   DSN6905I EXECUTION COMPLETE, MEMBER VPHONEC ADDED
   ***
   ```

   **Figure 28. Successful completion message**

   Db2 again displays the DCLGEN screen, as shown in the following figure.
DSNEDP01                   DCLGEN                             SSID: DSN

Enter table name for which declarations are required:
1  SOURCE TABLE NAME ===> DSN8C10.VPHONE

2  TABLE OWNER ..... ===> (Optional)

Enter destination data set: (Can be sequential or partitioned)
4  DATA SET NAME ... ===> TEMP(VPHONEC)

Enter options as desired:
6  ACTION .......... ===> ADD (ADD new or REPLACE old declaration)
7  COLUMN LABEL .... ===> NO (Enter YES for column label)
8  STRUCTURE NAME .. ===> (Optional)
9  FIELD NAME PREFIX ===> (Optional)
10  DELIMIT DBCS .... ===> YES (Enter YES to delimit DBCS identifiers)
11  COLUMN SUFFIX ... ===> NO (Enter YES to append column name)
12  INDICATOR VARS .. ===> NO (Enter YES for indicator variables)
13  ADDITIONAL OPTIONS===> NO (Enter YES to change additional options)

PRESS: ENTER to process    END to exit      HELP for more information

Figure 29. DCLGEN panel—displaying system and user return codes

c. Press Enter to return to the DB2I Primary Option menu.

3. Exit from DB2I.

4. Examine the DCLGEN output by selecting either the browse or the edit option from the ISPF/PDF menu to view the results in the specified data set member.

For this example, the data set to edit is prefix.TEMP.COBOL(VPHONEC). This data set member contains the following information.

***** DCLGEN TABLE(DSN8C10.VPHONE) ******
*****        LIBRARY(SYSADM.TEMP.COBOL(VPHONEC)) ******
*****        QUOTE ******
***** ... IS THE DCLGEN COMMAND THAT MADE THE FOLLOWING STATEMENTS ******
EXEC SQL DECLARE DSN8C10.VPHONE TABLE
  ( LASTNAME  VARCHAR(15) NOT NULL,
    FIRSTNAME VARCHAR(12) NOT NULL,
    MIDDLEINITIAL CHAR(1) NOT NULL,
    PHONENUMBER VARCHAR(4) NOT NULL,
    EMPLOYEENUMBER CHAR(6) NOT NULL,
    DEPTNUMBER   CHAR(3) NOT NULL,
    DEPTNAME      VARCHAR(36) NOT NULL ) END-EXEC.
***** COBOL DECLARATION FOR TABLE DSN8C10.VPHONE ******

01 DCLVPHONE.
  10 LASTNAME.
  49 LASTNAME-LEN PIC S9(4) USAGE COMP.
  49 LASTNAME-TEXT PIC X(15).
  10 FIRSTNAME.
  49 FIRSTNAME-LEN PIC S9(4) USAGE COMP.
  49 FIRSTNAME-TEXT PIC X(12).
  10 MIDDLEINITIAL PIC X(1).
  10 PHONENUMBER.
  49 PHONENUMBER-LEN PIC S9(4) USAGE COMP.
  49 PHONENUMBER-TEXT PIC X(4).
  10 EMPLOYEENUMBER PIC X(6).
  10 DEPTNUMBER PIC X(3).
  10 DEPTNAME.
  49 DEPTNAME-LEN PIC S9(4) USAGE COMP.
  49 DEPTNAME-TEXT PIC X(36).

***** THE NUMBER OF COLUMNS DESCRIBED BY THIS DECLARATION IS 7 ******

You can now pull these declarations into your program by using an SQL INCLUDE statement.
Defining the items that your program can use to check whether an SQL statement executed successfully

If your program contains SQL statements, the program should define some infrastructure so that it can check whether the statements executed successfully. You can either include an SQL communications area (SQLCA), which contains SQLCODE and SQLSTATE variables, or declare individual SQLCODE and SQLSTATE host variables.

**About this task**

Whether you define the SQLCODE or SQLSTATE variables or an SQLCA in your program depends on what you specify for the SQL processing option STDSQL.

If your application contains SQL statements and does not include an SQL communications area (SQLCA), you must declare individual SQLCODE and SQLSTATE host variables. Your program can use these variables to check whether an SQL statement executed successfully.

**Related tasks**

- Defining the SQL communications area, SQLSTATE, and SQLCODE in assembler
- Defining the SQL communications area, SQLSTATE, and SQLCODE in C and C++
- Defining the SQL communications area, SQLSTATE, and SQLCODE in COBOL
- Defining the SQL communications area, SQLSTATE, and SQLCODE in Fortran
- Defining the SQL communications area, SQLSTATE, and SQLCODE in PL/I
- Defining the SQL communications area, SQLSTATE, and SQLCODE in REXX

**Related reference**

- Descriptions of SQL processing options
- Description of SQLCA fields (Db2 SQL)
- The REXX SQLCA (Db2 SQL)
Defining SQL descriptor areas (SQLDA)

If your program includes certain SQL statements, you must define at least one SQL descriptor area (SQLDA). Depending on the context in which it is used, the SQLDA stores information about prepared SQL statements or host variables. This information can then be read by either the application program or Db2.

About this task

If your program includes any of the following statement variations, you must include an SQLDA in your program:

- CALL ... USING DESCRIPTOR descriptor-name
- DESCRIBE statement-name INTO descriptor-name
- DESCRIBE CURSOR host-variable INTO descriptor-name
- DESCRIBE INPUT statement-name INTO descriptor-name
- DESCRIBE PROCEDURE host-variable INTO descriptor-name
- DESCRIBE TABLE host-variable INTO descriptor-name
- EXECUTE ... USING DESCRIPTOR descriptor-name
- FETCH ... INTO DESCRIPTOR descriptor-name
- OPEN ... USING DESCRIPTOR descriptor-name
- PREPARE ... INTO descriptor-name

Unlike the SQLCA, a program can have more than one SQLDA, and an SQLDA can have any valid name.

Procedure

Take the actions that are appropriate for the programming language that you use:

- “Defining SQL descriptor areas (SQLDA) in assembler” on page 556
- “Defining SQL descriptor areas (SQLDA) in C and C++” on page 583
- “Defining SQL descriptor areas (SQLDA) in COBOL” on page 649
- “Defining SQL descriptor areas in (SQLDA) Fortran” on page 687
- “Defining SQL descriptor areas (SQLDA) in PL/I” on page 705
- “Defining SQL descriptor areas (SQLDA) in REXX” on page 743

Related reference

SQL descriptor area (SQLDA) (Db2 SQL)
Descriptions of SQL processing options
You can specify any SQL processing options regardless of whether you use the Db2 precompiler or the Db2 coprocessor. However, the Db2 coprocessor might ignore certain options because host language compiler options exist that provide the same information.

Description of SQLCA fields (Db2 SQL)
The REXX SQLCA (Db2 SQL)

Declaring host variables and indicator variables

You can use host variables and indicator variables in SQL statements in your program to pass data between Db2 and your application.

Procedure

Use the techniques that are appropriate for the programming language that you use.
Related tasks
Accessing data by using a rowset-positioned cursor
A rowset-positioned cursor is a cursor that can return one or more rows for a single fetch operation. The cursor is positioned on the set of rows that are to be fetched.

Determining whether a retrieved value in a host variable is null or truncated
Before your application manipulates the data that was retrieved from Db2 into a host variable, determine if the value is null. Also determine if it was truncated when assigned to the variable. You can use indicator variables to obtain this information.

Related reference
Descriptions of SQL processing options
You can specify any SQL processing options regardless of whether you use the Db2 precompiler or the Db2 coprocessor. However, the Db2 coprocessor might ignore certain options because host language compiler options exist that provide the same information.

Host variables
Use host variables to pass a single data item between Db2 and your application.

A host variable is a single data item that is declared in the host language to be used within an SQL statement. You can use host variables in application programs that are written in the following languages: assembler, C, C++, COBOL, Fortran, and PL/I to perform the following actions:

- Retrieve data into the host variable for your application program's use
- Place data into the host variable to insert into a table or to change the contents of a row
- Use the data in the host variable when evaluating a WHERE or HAVING clause
- Assign the value that is in the host variable to a special register, such as CURRENT SQLID and CURRENT DEGREE
- Insert null values into columns by using a host indicator variable that contains a negative value
- Use the data in the host variable in statements that process dynamic SQL, such as EXECUTE, PREPARE, and OPEN

Related concepts
Using host variables in SQL statements
Use scalar host variables in embedded SQL statements to represent a single value. Host variables are useful for storing retrieved data or for passing values that are to be assigned or used for comparisons.

Related reference
Host variables in assembler
In assembler programs, you can specify numeric, character, graphic, binary, LOB, XML, and ROWID host variables. You can also specify result set, table, and LOB locators and LOB and XML file reference variables.

Host variables in C and C++
In C and C++ programs, you can specify numeric, character, graphic, binary, LOB, XML, and ROWID host variables. You can also specify result set, table, and LOB locators and LOB and XML file reference variables.

Host variables in COBOL
In COBOL programs, you can specify numeric, character, graphic, binary, LOB, XML, and ROWID host variables. You can also specify result set and table locators and LOB and XML file reference variables.

Host variables in Fortran
In Fortran programs, you can specify numeric, character, LOB, and ROWID host variables. You can also specify result set and LOB locators.

Host variables in PL/I
In PL/I programs, you can specify numeric, character, graphic, binary, LOB, XML, and ROWID host variables. You can also specify result set, table, and LOB locators and LOB and XML file reference variables.

**Host-variable arrays**

You can use host-variable arrays to pass a data array between Db2 and your application. A *host-variable array* is a data array that is declared in the host language to be used within an SQL statement.

You can use host-variable arrays for the following actions:

- Retrieve data into host-variable arrays for your application use by your application
- Place data into host-variable arrays to insert rows into a table
- Retrieve data for the source of a merge operation.

Host-variable arrays can be referenced only as a simple reference in the following contexts. In syntax diagrams, *host-variable-array* designates a reference to a host-variable array.

- In a FETCH statement for a multiple-row fetch. See FETCH (Db2 SQL).
- In an INSERT statement with multiple rows of source data. This is also referred to as multiple-row insert. See INSERT (Db2 SQL).
- In a MERGE statement with multiple rows of source data. See MERGE (Db2 SQL).
- In an EXECUTE statement to provide a value for a parameter marker in a dynamic multi-row INSERT or MERGE. See EXECUTE (Db2 SQL).

Host-variable arrays are defined by statements of the host language, as explained in the following topics:

- “Host-variable arrays in C and C++” on page 595
- “Host-variable arrays in COBOL” on page 660
- “Host-variable arrays in PL/I” on page 712

**Tip:** Host-variable arrays are not supported for assembler, FORTRAN, or REXX programs. However, you can use SQL descriptor areas (SQLDA) to achieve similar results in any host language. For more information see “Defining SQL descriptor areas (SQLDA)” on page 476.

**Example**

The following statement uses the main host-variable array, COL1, and the corresponding indicator array, COL1IND. Assume that COL1 has 10 elements. The first element in the array corresponds to the first value, and so on. COL1IND must have at least 10 entries.

```
EXEC SQL
  SQL FETCH FIRST ROWSET FROM C1 FOR 5 ROWS
    INTO :COL1 :COL1IND
END-EXEC.
```

**Related concepts**

Host-variable arrays in PL/I, C, C++, and COBOL (Db2 SQL)
Using host-variable arrays in SQL statements
Use host-variable arrays in embedded SQL statements to represent values that the program does not know until the query is executed. Host-variable arrays are useful for storing a set of retrieved values or for passing a set of values that are to be inserted into a table.

**Related tasks**

Inserting multiple rows of data from host-variable arrays
Use host-variable arrays in your INSERT statement when you do not know at least some of the values to insert until the program runs.

Retrieving multiple rows of data into host-variable arrays
If you know that your query returns multiple rows, you can specify host-variable arrays to store the retrieved column values.

Related reference
Host-variable arrays in C and C++
In C and C++ programs, you can specify numeric, character, graphic, binary, LOB, XML, and ROWID host-variable arrays. You can also specify LOB locators and LOB and XML file reference variables.

Host-variable arrays in COBOL
In COBOL programs, you can specify numeric, character, graphic, LOB, XML, and ROWID host-variable arrays. You can also specify LOB locators and LOB and XML file reference variables.

Host-variable arrays in PL/I
In PL/I programs, you can specify numeric, character, graphic, binary, LOB, XML, and ROWID host-variable arrays. You can also specify LOB locators and LOB and XML file reference variables.

Host structures
Use host structures to pass a group of host variables between Db2 and your application.

A host structure is a group of host variables that can be referenced with a single name. You can use host structures in all host languages except REXX. You define host structures with statements in the host language. You can refer to a host structure in any context where you want to refer to the list of host variables in the structure. A host structure reference is equivalent to a reference to each of the host variables within the structure in the order in which they are defined in the structure declaration. You can also use indicator variables (or indicator structures) with host structures.

Related tasks
Retrieving a single row of data into a host structure
If you know that your query returns multiple column values for only one row, you can specify a host structure to contain the column values.

Related reference
Host structures in C and C++
A C host structure contains an ordered group of data fields.

Host structures in COBOL
A COBOL host structure is a named set of host variables that are defined in your program’s WORKING-STORAGE SECTION or LINKAGE SECTION.

Host structures in PL/I
A PL/I host structure is a structure that contains subordinate levels of scalars. You can use the name of the structure as shorthand notation to reference the list of scalars.

Indicator variables, arrays, and structures
An indicator variable is associated with a particular host variable. Each indicator variable contains a small integer value that indicates some information about the associated host variable. Indicator arrays and structures serve the same purpose for host-variable arrays and structures.

You can use indicator variables to perform the following actions:

• Determine whether the value of an associated output host variable is null or indicate that the value of an input host variable is null
• Determine the original length of a character string that was truncated when it was assigned to a host variable
• Determine that a character value could not be converted when it was assigned to a host variable
• Determine the seconds portion of a time value that was truncated when it was assigned to a host variable.
• Indicate that the target column of the host variable is to be set to its defined DEFAULT value, or that the host variable's value is UNASSIGNED and its target column is to be treated as if it had not appeared in the statement.

You can use indicator variable arrays and indicator structures to perform these same actions for individual items in host data arrays and structures.

If you provide an indicator variable for the variable X, when Db2 retrieves a null value for X, it puts a negative value in the indicator variable and does not update X. Your program should check the indicator variable before using X. If the indicator variable is negative, you know that X is null and any value that you find in X is irrelevant. When your program uses variable X to assign a null value to a column, the program should set the indicator variable to a negative number. Db2 then assigns a null value to the column and ignores any value in X.

An indicator variable array contains a series of small integers to help you determine the associated information for the corresponding item in a host data array. When you retrieve data into a host-variable array, you can check the values in the associated indicator array to determine how to handle each data item. If a value in the associated indicator array is negative, you can disregard the contents of the corresponding element in the host-variable array. Values in indicator arrays have the following meanings:

On output to the application, the normal indicator variable can contain the following values:

0
A 0 (zero), or positive value of the indicator variable specifies that the first host-identifier provides the value of this host variable reference.

-1
A -1 value indicates that the value that was selected was the null value.

-2
A -2 value of the indicator variable indicates that a numeric conversion error (such as a divide by 0 or overflow) has occurred. Or indicates a null result because of character string conversion warnings.

-3
A -3 value of the indicator variable indicates that no value was returned. A -3 value of the indicator variable can also indicate a null result because the cursor's current row is on a hole that was detected during a multiple row FETCH.

**positive integer**
If the indicator variable contains a positive integer, the retrieved value is truncated, and the integer is the original length of the string.

**positive integer**
The seconds portion of a time if the time is truncated on assignment to a host variable.

On input to Db2, normal indicator variables or extended indicator variables can contain the following values:

0, or positive integer
Specifies a non-null value. A 0 (zero), or positive value of the indicator variable specifies that the first host-identifier provides the value of this host variable reference.

-1, -2, -3, -4, -6
Specifies a null value.

-5
If extended indicator variables are not enabled, a -5 value specifies the NULL value.

-7
If extended indicator variables are not enabled, a -7 value specifies the NULL value.
• If extended indicator variables are enabled, a -7 value specifies the UNASSIGNED value. A -7 value specifies that the target column for this host variable is to be treated as if it had not been specified in the statement.

An indicator structure is an array of halfword integer variables that supports a specified host structure. If the column values that your program retrieves into a host structure can be null, you can attach an indicator structure name to the host structure name. This name enables Db2 to notify your program about each null value it returns to a host variable in the host structure.

**Related concepts**
Holes in the result table of a scrollable cursor
A hole in the result table means that the result table does not shrink to fill the space of deleted rows. It also does not shrink to fill the space of rows that have been updated and no longer satisfy the search condition. You cannot access a delete or update hole. However, you can remove holes in specific situations.

**Related tasks**
Executing SQL statements by using a rowset cursor
You can use rowset cursors to execute multiple-row FETCH statements, positioned UPDATE statements, and positioned DELETE statements.

**Related reference**
Indicator variables in assembler
An indicator variable is a 2-byte integer (DS HL2). You declare indicator variables in the same way as host variables. You can mix the declarations of the two types of variables.

Indicator variables, indicator arrays, and host structure indicator arrays in C and C++
An indicator variable is a 2-byte integer (short int). An indicator variable array is an array of 2-byte integers (short int). You declare indicator variables in the same way as host variables. You can mix the declarations of the two types of variables.

Indicator variables, indicator arrays, and host structure indicator arrays in COBOL
A COBOL indicator variable is a 2-byte binary integer. A COBOL indicator variable array is an array in which each element is declared as a 2-byte binary integer. You can use indicator variable arrays to support COBOL host structures.

Indicator variables in Fortran
An indicator variable is a 2-byte integer (INTEGER*2). You declare indicator variables in the same way as host variables. You can mix the declarations of the two types of variables.

Indicator variables in PL/I
An indicator variable is a 2-byte integer (or an integer declared as BIN FIXED(15)). An indicator variable array is an array of 2-byte integers. You declare indicator variables in the same way as host variables. You can mix the declarations of the two types of variables.

**Setting the CCSID for host variables**
All Db2 string data, other than binary data, has an encoding scheme and a coded character set ID (CCSID) associated with it. You can associate an encoding scheme and a CCSID with individual host variables. Any data in those host variable is then associated with that encoding scheme and CCSID.

**Procedure**
Specify the DECLARE VARIABLE statement after the corresponding host variable declaration and before your first reference to that host variable.

This statement associates an encoding scheme and a CCSID with individual host variables. You can use this statement in static or dynamic SQL applications.

**Restriction:** You cannot use the DECLARE VARIABLE statement to control the CCSID and encoding scheme of data that you retrieve or update by using an SQLDA.

The DECLARE VARIABLE statement has the following effects on a host variable:
When you use the host variable to update a table, the local subsystem or the remote server assumes that the data in the host variable is encoded with the CCSID and encoding scheme that the DECLARE VARIABLE statement assigns.

When you retrieve data from a local or remote table into the host variable, the retrieved data is converted to the CCSID and encoding scheme that are assigned by the DECLARE VARIABLE statement.

**Example**

Suppose that you are writing a C program that runs on a Db2 for z/OS subsystem. The subsystem has an EBCDIC application encoding scheme. The C program retrieves data from the following columns of a local table that is defined with the CCSID UNICODE option:

<table>
<thead>
<tr>
<th>PARTNUM</th>
<th>CHAR(10)</th>
</tr>
</thead>
<tbody>
<tr>
<td>JPNNAME</td>
<td>GRAPHIC(10)</td>
</tr>
<tr>
<td>ENGNAME</td>
<td>VARCHAR(30)</td>
</tr>
</tbody>
</table>

Because the application encoding scheme for the subsystem is EBCDIC, the retrieved data is EBCDIC. To make the retrieved data Unicode, use DECLARE VARIABLE statements to specify that the data that is retrieved from these columns is encoded in the default Unicode CCSIDs for the subsystem.

Suppose that you want to retrieve the character data in Unicode CCSID 1208 and the graphic data in Unicode CCSID 1200. Use the following DECLARE VARIABLE statements:

```sql
EXEC SQL BEGIN DECLARE SECTION;
char hvpartnum[11];
EXEC SQL DECLARE :hvpartnum VARIABLE CCSID 1208;
sqldbchar hvjpnname[11];
EXEC SQL DECLARE :hvjpnname VARIABLE CCSID 1200;
struct {
    short len;
    char d[30];
} hvengname;
EXEC SQL DECLARE :hvengname VARIABLE CCSID 1208;
EXEC SQL END DECLARE SECTION;
```

**Related reference**

DECLARE VARIABLE (Db2 SQL)

**Determining what caused an error when retrieving data into a host variable**

Errors that occur when Db2 passes data to host variables in an application are usually caused by a problem in converting from one data type to another. These errors do not affect the position of the cursor.

**About this task**

For example, suppose that you fetch an integer value of 32768 into a host variable of type SMALLINT. The conversion might cause an error if you do not provide sufficient conversion information to Db2.

The variable to which Db2 assigns the data is called the **output host variable**. If you provide an indicator variable for the output host variable or if data type conversion is not required, Db2 returns a positive SQLCODE for the row in most cases. In other cases where data conversion problems occur, Db2 returns a negative SQLCODE for that row. Regardless of the SQLCODE for the row, no new values are assigned to the host variable or to subsequent variables for that row. Any values that are already assigned to variables remain assigned. Even when a negative SQLCODE is returned for a row, statement processing continues and Db2 returns a positive SQLCODE for the statement (SQLSTATE 01668, SQLCODE +354).

**Procedure**

To determine what caused an error when retrieving data into a host variable:

1. When Db2 returns SQLCODE = +354, use the GET DIAGNOSTICS statement with the NUMBER option to determine the number of errors and warnings.

   For example, suppose that no indicator variables are provided for the values that are returned by the following statement:
FETCH FIRST ROWSET FROM C1 FOR 10 ROWS INTO :hva_col1, :hva_col2;

For each row with an error, Db2 records a negative SQLCODE and continues processing until the 10 rows are fetched. When SQLCODE = +354 is returned for the statement, you can use the GET DIAGNOSTICS statement to determine which errors occurred for which rows. The following statement returns num_rows = 10 and num_cond = 3:

GET DIAGNOSTICS :num_rows = ROW_COUNT, :num_cond = NUMBER;

2. To investigate the errors and warnings, use additional GET DIAGNOSTIC statements with the CONDITION option. For example, to investigate the three conditions that were reported in the example in the previous step, use the following statements:

<table>
<thead>
<tr>
<th>Table 87. GET DIAGNOSTIC statements to investigate conditions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Statement</td>
</tr>
<tr>
<td>---------------------------------</td>
</tr>
<tr>
<td>GET DIAGNOSTICS CONDITION 3 :sqlstate = RETURNED_SQLSTATE, :sqlcode = DB2_RETURNED_SQLCODE, :row_num = DB2_ROW_NUMBER;</td>
</tr>
<tr>
<td>GET DIAGNOSTICS CONDITION 2 :sqlstate = RETURNED_SQLSTATE, :sqlcode = DB2_RETURNED_SQLCODE, :row_num = DB2_ROW_NUMBER;</td>
</tr>
<tr>
<td>GET DIAGNOSTICS CONDITION 1 :sqlstate = RETURNED_SQLSTATE, :sqlcode = DB2_RETURNED_SQLCODE, :row_num = DB2_ROW_NUMBER;</td>
</tr>
</tbody>
</table>

This output shows that the fifth row has a data mapping error (-304) for column 1 and that the seventh row has a data mapping error (-802) for column 2. These rows do not contain valid data, and they should not be used.

Related concepts
Indicator variables, arrays, and structures
An indicator variable is associated with a particular host variable. Each indicator variable contains a small integer value that indicates some information about the associated host variable. Indicator arrays and structures serve the same purpose for host-variable arrays and structures.

Related reference
GET DIAGNOSTICS (Db2 SQL)

Related information
+354 (Db2 Codes)

Accessing an application defaults module

If your application program currently uses LOAD DSNHDECP, consider changing the application program to use the DECP address that is returned by IFCID 373, DSNALI, or DSNRLI.

About this task
By using the DECP address that is returned by IFCID 373, DSNALI, or DSNRLI, guarantees that you are using the same DECP module that was used to start Db2. It also allows the code to skip the LOAD entirely, only after successfully connecting to Db2. DSNHDECP is loaded by Db2 into Global, pageable storage, so all programs can share it.
Compatibility of SQL and language data types

The host variable data types that are used in SQL statements must be compatible with the data types of the columns with which you intend to use them.

When deciding the data types of host variables, consider the following rules and recommendations:

• Numeric data types are compatible with each other:

  **Assembler**
  A SMALLINT, INTEGER, BIGINT, DECIMAL, or FLOAT column is compatible with a numeric assembler host variable.

  **Fortran**
  An INTEGER column is compatible with any Fortran host variable that is defined as INTEGER*2, INTEGER*4, REAL, REAL*4, REAL*8, or DOUBLE PRECISION.

  **PL/I**
  A SMALLINT, INTEGER, BIGINT, DECIMAL, or FLOAT column is compatible with a PL/I host variable of BIN FIXED(15), BIN FIXED(31), DECIMAL(s,p), or BIN FLOAT(n), where n is from 1 to 53, or DEC FLOAT(m) where m is from 1 to 16.

• Character data types are compatible with each other:

  **Assembler**
  A CHAR, VARCHAR, or CLOB column is compatible with a fixed-length or varying-length assembler character host variable.

  **C/C++**
  A CHAR, VARCHAR, or CLOB column is compatible with a single-character, NUL-terminated, or VARCHAR structured form of a C character host variable.

  **COBOL**
  A CHAR, VARCHAR, or CLOB column is compatible with a fixed-length or varying-length COBOL character host variable.

  **Fortran**
  A CHAR, VARCHAR, or CLOB column is compatible with Fortran character host variable.

  **PL/I**
  A CHAR, VARCHAR, or CLOB column is compatible with a fixed-length or varying-length PL/I character host variable.

• Character data types are partially compatible with CLOB locators. You can perform the following assignments:
  - Assign a value in a CLOB locator to a CHAR or VARCHAR column
  - Use a SELECT INTO statement to assign a CHAR or VARCHAR column to a CLOB locator host variable.
  - Assign a CHAR or VARCHAR output parameter from a user-defined function or stored procedure to a CLOB locator host variable.
  - Use a SET assignment statement to assign a CHAR or VARCHAR transition variable to a CLOB locator host variable.
  - Use a VALUES INTO statement to assign a CHAR or VARCHAR function parameter to a CLOB locator host variable.

  However, you cannot use a FETCH statement to assign a value in a CHAR or VARCHAR column to a CLOB locator host variable.

• Graphic data types are compatible with each other:

  **Assembler**
  A GRAPHIC, VARGRAPHIC, or DBCLOB column is compatible with a fixed-length or varying-length assembler graphic character host variable.
C/C++
A GRAPHIC, VARGRAPHIC, or DBCLOB column is compatible with a single character, NUL-terminated, or VARGRAPHIC structured form of a C graphic host variable.

COBOL
A GRAPHIC, VARGRAPHIC, or DBCLOB column is compatible with a fixed-length or varying-length COBOL graphic string host variable.

PL/I
A GRAPHIC, VARGRAPHIC, or DBCLOB column is compatible with a fixed-length or varying-length PL/I graphic character host variable.

- Graphic data types are partially compatible with DBCLOB locators. You can perform the following assignments:
  - Assign a value in a DBCLOB locator to a GRAPHIC or VARGRAPHIC column
  - Use a SELECT INTO statement to assign a GRAPHIC or VARGRAPHIC column to a DBCLOB locator host variable.
  - Assign a GRAPHIC or VARGRAPHIC output parameter from a user-defined function or stored procedure to a DBCLOB locator host variable.
  - Use a SET assignment statement to assign a GRAPHIC or VARGRAPHIC transition variable to a DBCLOB locator host variable.
  - Use a VALUES INTO statement to assign a GRAPHIC or VARGRAPHIC function parameter to a DBCLOB locator host variable.

However, you cannot use a FETCH statement to assign a value in a GRAPHIC or VARGRAPHIC column to a DBCLOB locator host variable.

- Binary data types are compatible with each other.
- Binary data types are partially compatible with BLOB locators. You can perform the following assignments:
  - Assign a value in a BLOB locator to a BINARY or VARBINARY column.
  - Use a SELECT INTO statement to assign a BINARY or VARBINARY column to a BLOB locator host variable.
  - Assign a BINARY or VARBINARY output parameter from a user-defined function or stored procedure to a BLOB locator host variable.
  - Use a SET assignment statement to assign a BINARY or VARBINARY transition variable to a BLOB locator host variable.
  - Use a VALUES INTO statement to assign a BINARY or VARBINARY function parameter to a BLOB locator host variable.

However, you cannot use a FETCH statement to assign a value in a BINARY or VARBINARY column to a BLOB locator host variable.

- Datetime data types are compatible with character host variables.

Fortran
A BINARY, VARBINARY, or BLOB column or BLOB locator is compatible only with a BLOB host variable.

C:
For varying-length BIT data, use BINARY. Some C string manipulation functions process NUL-terminated strings and other functions process strings that are not NUL-terminated. The C string manipulation functions that process NUL-terminated strings cannot handle bit data because these functions might misinterpret a NUL character to be a NUL-terminator.

Assembler
A DATE, TIME, or TIMESTAMP column is compatible with a fixed-length or varying-length assembler character host variable.
C/C++  
A DATE, TIME, or TIMESTAMP column is compatible with a single-character, NUL-terminated, or VARCHAR structured form of a C character host variable.

COBOL  
A DATE, TIME, or TIMESTAMP column is compatible with a fixed-length or varying length COBOL character host variable.

Fortran  
A DATE, TIME, or TIMESTAMP column is compatible with a Fortran character host variable.

PL/I  
A DATE, TIME, or TIMESTAMP column is compatible with a fixed-length or varying-length PL/I character host variable.

•  
The ROWID column is compatible only with a ROWID host variable.

• A host variable is compatible with a distinct type if the host variable type is compatible with the source type of the distinct type.

• XML columns are compatible with the XML host variable types, character types, and binary string types.  

Recommendation: Use the XML host variable types for data from XML columns.

• Assembler  
You can assign LOB data to a file reference variable (BLOB_FILE, CLOB_FILE, and DBCLOB_FILE).

When necessary, Db2 automatically converts a fixed-length string to a varying-length string, or a varying-length string to a fixed-length string.

Related concepts  
Distinct types  
A distinct type is a user-defined data type that shares its internal representation with a built-in data type (its source type), but is considered to be a separate and incompatible data type for most operations.

Host variable data types for XML data in embedded SQL applications (Db2 Programming for XML)  
Related reference  
Equivalent SQL and assembler data types  
When you declare host variables in your assembler programs, the precompiler uses equivalent SQL data types. When you retrieve data of a particular SQL data type into a host variable, ensure that the host variable is of an equivalent data type.

Equivalent SQL and C data types  
When you declare host variables in your C programs, the precompiler uses equivalent SQL data types. When you retrieve data of a particular SQL data type into a host variable, you need to ensure that the host variable is of an equivalent data type.

Equivalent SQL and COBOL data types  
When you declare host variables in your COBOL programs, the precompiler uses equivalent SQL data types. When you retrieve data of a particular SQL data type into a host variable, you need to ensure that the host variable is of an equivalent data type.

Equivalent SQL and Fortran data types  
When you declare host variables in your Fortran programs, the precompiler uses equivalent SQL data types. When you retrieve data of a particular SQL data type into a host variable, ensure that the host variable is of an equivalent data type.

Equivalent SQL and PL/I data types  
When you declare host variables in your PL/I programs, the precompiler uses equivalent SQL data types. When you retrieve data of a particular SQL data type into a host variable, you need to ensure that the host variable is of an equivalent data type.

Equivalent SQL and REXX data types
All REXX data is string data. Therefore, when a REXX program assigns input data to a column, Db2 converts the data from a string type to the column type. When a REXX program assigns column data to an output variable, Db2 converts the data from the column type to a string type.

Using host variables in SQL statements

Use scalar host variables in embedded SQL statements to represent a single value. Host variables are useful for storing retrieved data or for passing values that are to be assigned or used for comparisons.

When you use host variables, adhere to the following requirements:

• You must declare the name of the host variable in the host program before you use it. Host variables follow the naming conventions of the host language.
• You can use a host variable to represent a data value, but you cannot use it to represent a table, view, or column name. You can specify table, view, or column names at run time by using dynamic SQL.
• To use a host variable in an SQL statement, you can specify any valid host variable name that is declared according to the rules of the host language.
• A colon (:) must precede host variables that are used in SQL statements so that Db2 can distinguish a variable name from a column name. When host variables are used outside of SQL statements, do not precede them with a colon. PL/I programs have the following exceptions: If the SQL statement meets any of the following conditions, do not precede a host variable or host variable array in that statement with a colon:
  – The SQL statement is in a program that also contains a DECLARE VARIABLE statement.
  – The host variable is part of a string expression, but the host variable is not the only component of the string expression.
• To optimize performance, make sure that the host language declaration maps as closely as possible to the data type of the associated data in the database.
• For assignments and comparisons between a Db2 column and a host variable of a different data type or length, expect conversions to occur.

Related concepts
Host variables
Use host variables to pass a single data item between Db2 and your application.

Assignment and comparison (Db2 SQL)
Host variables (Db2 SQL)

Related tasks
Including dynamic SQL in your program
Dynamic SQL is prepared and executed while the program is running.

Retrieving a single row of data into host variables

If you know that your query returns only one row, you can specify one or more host variables to contain the column values of the retrieved row.

About this task

Restriction: These instructions do not apply if you do not know how many rows Db2 will return or if you expect Db2 to return more than one row. In these situations, use a cursor. A cursor enables an application to return a set of rows and fetch either one row at a time or one rowset at a time from the result table.

Procedure

In the SELECT statement specify the INTO clause with the name of one or more host variables to contain the retrieved values. Specify one variable for each value that is to be retrieved. The retrieved value can be a column value, a value of a host variable, the result of an expression, or the result of an aggregate function.
Recommendation: If you want to ensure that only one row is returned, specify the `FETCH FIRST 1 ROW ONLY` clause. Consider using the `ORDER BY` clause to control which row is returned. If you specify both the `ORDER BY` clause and the `FETCH FIRST` clause, ordering is performed on the entire result set before the first row is returned.

Db2 assigns the first value in the result row to the first variable in the list, the second value to the second variable, and so on.

If the `SELECT` statement returns more than one row, Db2 returns an error, and any data that is returned is undefined and unpredictable.

Examples

Example: Retrieving a single row into a host variable
Suppose that you are retrieving the LASTNAME and WORKDEPT column values from the DSN8C10.EMP table for a particular employee. You can define a host variable in your program to hold each column value and then name the host variables in the INTO clause of the `SELECT` statement, as shown in the following COBOL example.

```
MOVE '000110' TO CBLEMPNO.
EXEC SQL
   SELECT LASTNAME, WORKDEPT
   INTO :CBLNAME, :CBLDEPT
   FROM DSN8C10.EMP
   WHERE EMPNO = :CBLEMPNO
END-EXEC.
```

In this example, the host variable CBLEMPNO is preceded by a colon (:) in the SQL statement, but it is not preceded by a colon in the COBOL MOVE statement.

This example also uses a host variable to specify a value in a search condition. The host variable CBLEMPNO is defined for the employee number, so that you can retrieve the name and the work department of the employee whose number is the same as the value of the host variable, CBLEMPNO; in this case, 000110.

In the DATA DIVISION section of a COBOL program, you must declare the host variables CBLEMPNO, CBLNAME, and CBLDEPT to be compatible with the data types in the columns EMPNO, LASTNAME, and WORKDEPT of the DSN8C10.EMP table.

Example: Ensuring that a query returns only a single row
You can use the `FETCH FIRST 1 ROW ONLY` clause in a `SELECT` statement to ensure that only one row is returned. This action prevents undefined and unpredictable data from being returned when you specify the INTO clause of the `SELECT` statement. The following example `SELECT` statement ensures that only one row of the DSN8C10.EMP table is returned.

```
EXEC SQL
   SELECT LASTNAME, WORKDEPT
   INTO :CBLNAME, :CBLDEPT
   FROM DSN8C10.EMP
   FETCH FIRST 1 ROW ONLY
END-EXEC.
```

You can include an `ORDER BY` clause in the preceding example to control which row is returned. The following example `SELECT` statement ensures that the only row returned is the one with a last name that is first alphabetically.

```
EXEC SQL
   SELECT LASTNAME, WORKDEPT
   INTO :CBLNAME, :CBLDEPT
   FROM DSN8B10.EMP
   ORDER BY LASTNAME
   FETCH FIRST 1 ROW ONLY
END-EXEC.
```

Example: Retrieving the results of host variable values and expressions into host variables
When you specify a list of items in the `SELECT` clause, that list can include more than the column names of tables and views. You can request a set of column values mixed with host variable values.
and constants. For example, the following query requests the values of several columns (EMPNO, LASTNAME, and SALARY), the value of a host variable (RAISE), and the value of the sum of a column and a host variable (SALARY and RAISE). For each of these five items in the SELECT list, a host variable is listed in the INTO clause.

```
MOVE 4476 TO RAISE.
MOVE '000220' TO PERSON.
EXEC SQL
  SELECT EMPNO, LASTNAME, SALARY, :RAISE, SALARY + :RAISE
  INTO :EMP-NUM, :PERSON-NAME, :EMP-SAL, :EMP-RAISE, :EMP-TTL
  FROM DSN8C10.EMP
  WHERE EMPNO = :PERSON
END-EXEC.
```

The preceding SELECT statement returns the following results. The column headings represent the names of the host variables.

<table>
<thead>
<tr>
<th>EMP-NUM</th>
<th>PERSON-NAME</th>
<th>EMP-SAL</th>
<th>EMP-RAISE</th>
<th>EMP-TTL</th>
</tr>
</thead>
<tbody>
<tr>
<td>000220</td>
<td>LUTZ</td>
<td>29840</td>
<td>4476</td>
<td>34316</td>
</tr>
</tbody>
</table>

Example: Retrieving the result of an aggregate function into a host variable

A query can request summary values to be returned from aggregate functions and store those values in host variables. For example, the following query requests that the result of the AVG function be stored in the AVG-SALARY host variable.

```
MOVE 'D11' TO DEPTID.
EXEC SQL
  SELECT WORKDEPT, AVG(SALARY)
  INTO :WORK-DEPT, :AVG-SALARY
  FROM DSN8C10.EMP
  WHERE WORKDEPT = :DEPTID
END-EXEC.
```

### Related tasks
- **Retrieving a set of rows by using a cursor**
In an application program, you can retrieve a set of rows from a table or a result table that is returned by a stored procedure. You can retrieve one or more rows at a time.

### Related reference
- **SELECT INTO (Db2 SQL)**

### Determining whether a retrieved value in a host variable is null or truncated

Before your application manipulates the data that was retrieved from Db2 into a host variable, determine if the value is null. Also determine if it was truncated when assigned to the variable. You can use indicator variables to obtain this information.

### Before you begin

Before you determine whether a retrieved column value is null or truncated, you must have defined the appropriate indicator variables, arrays, and structures.

### About this task

An error occurs if you do not use an indicator variable and Db2 retrieves a null value.

### Procedure

Determine the value of the indicator variable, array, or structure that is associated with the host variable, array, or structure.

Those values have the following meanings:
### Table 88. Meanings of values in indicator variables

<table>
<thead>
<tr>
<th>Value of indicator variable</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Less than zero</td>
<td>The column value is null. The value of the host variable does not change from its previous value. If the indicator variable value is -2, the column value is null because of a numeric or character conversion error,</td>
</tr>
<tr>
<td>Zero</td>
<td>The column value is nonnull. If the column value is a character string, the retrieved value is not truncated.</td>
</tr>
<tr>
<td>Positive integer</td>
<td>The retrieved value is truncated. The integer is the original length of the string.</td>
</tr>
</tbody>
</table>

#### Examples

**Example of testing an indicator variable**

Assume that you have defined the following indicator variable INDNULL for the host variable CBLPHONE.

```sql
EXEC SQL
SELECT PHONENO INTO :CBLPHONE:INDNULL FROM DSN8C10.EMP WHERE EMPNO = :EMPID END-EXEC.
```

You can then test INDNULL for a negative value. If the value is negative, the corresponding value of PHONENO is null, and you can disregard the contents of CBLPHONE.

**Example of testing an indicator variable array**

Suppose that you declare the following indicator array INDNULL for the host-variable array CBLPHONE.

```sql
EXEC SQL
FETCH NEXT ROWSET CURS1 FOR 10 ROWS INTO :CBLPHONE :INDNULL END-EXEC.
```

After the multiple-row FETCH statement, you can test each element of the INDNULL array for a negative value. If an element is negative, you can disregard the contents of the corresponding element in the CBLPHONE host-variable array.

**Example of testing an indicator structure in COBOL**

The following example defines the indicator structure EMP-IND as an array that contains six values and corresponds to the PEMP-ROW host structure.

```cobol
01 PEMP-ROW.
   10 EMPNO PIC X(6).
   10 FIRSTNME.
      49 FIRSTNME-LEN PIC S9(4) USAGE COMP.
      49 FIRSTNME-TEXT PIC X(12).
   10 MIDINIT PIC X(1).
   10 LASTNAME.
      49 LASTNAME-LEN PIC S9(4) USAGE COMP.
      49 LASTNAME-TEXT PIC X(15).
   10 WORKDEPT PIC X(3).
   10 EMP-BIRTHDATE PIC X(10).
01 INDICATOR-TABLE.
   02 EMP-IND PIC S9(4) COMP OCCURS 6 TIMES.
   ... MOVE '000230' TO EMPNO.
   ... EXEC SQL
       SELECT EMPNO, FIRSTNME, MIDINIT, LASTNAME, WORKDEPT, BIRTHDATE
```
You can test the indicator structure EMP-IND for negative values. If, for example, EMP-IND(6) contains a negative value, the corresponding host variable in the host structure (EMP-BIRTHDATE) contains a null value.

**Related concepts**

Arithmetic and conversion errors
You can track arithmetic and conversion errors by using indicator variables. An indicator variable contains a small integer value that indicates some information about the associated host variable.

**Related tasks**

Declaring host variables and indicator variables
You can use host variables and indicator variables in SQL statements in your program to pass data between Db2 and your application.

### Updating data by using host variables

When you want to update a value in a Db2 table, but you do not know the exact value until the program runs, use host variables. Db2 can change a table value to match the current value of the host variable.

**Procedure**

To update data by using host variables:
1. Declare the necessary host variables.
2. Specify an UPDATE statement with the appropriate host variable names in the SET clause.

**Examples**

**Example of updating a single row by using a host variable**

The following COBOL example changes an employee's phone number to the value in the NEWPHONE host variable. The employee ID value is passed through the EMPID host variable.

```
MOVE '4246' TO NEWPHONE.
MOVE '000110' TO EMPID.
EXEC SQL
  UPDATE DSN8C10.EMP
    SET PHONENO = :NEWPHONE
    WHERE EMPNO = :EMPID
END-EXEC.
```

**Example of updating a single row by using a host variable**

The following example gives the employees in a particular department a salary increase of 10%. The department value is passed through the DEPTID host variable.

```
MOVE 'D11' TO DEPTID.
EXEC SQL
  UPDATE DSN8C10.EMP
    SET SALARY = 1.10 * SALARY
    WHERE WORKDEPT = :DEPTID
END-EXEC.
```

**Related reference**

UPDATE (Db2 SQL)
Inserting a single row by using a host variable

Use host variables in your INSERT statement when you don't know at least some of the values to insert until the program runs.

About this task

Restriction: These instructions apply only to inserting a single row. If you want to insert multiple rows, use host variable arrays or the form of the INSERT statement that selects values from another table or view.

Procedure

Specify an INSERT statement with column values in the VALUES clause. Specify host variables or a combination of host variables and constants as the column values.

Db2 inserts the first value into the first column in the list, the second value into the second column, and so on.

Example

The following example uses host variables to insert a single row into the activity table.

EXEC SQL
  INSERT INTO DSN8C10.ACT
  VALUES (:HV-ACTNO, :HV-ACTKWD, :HV-ACTDESC)
END-EXEC.

Related tasks

Inserting multiple rows of data from host-variable arrays

Use host-variable arrays in your INSERT statement when you do not know at least some of the values to insert until the program runs.

Related reference

INSERT (Db2 SQL)

Using host-variable arrays in SQL statements

Use host-variable arrays in embedded SQL statements to represent values that the program does not know until the query is executed. Host-variable arrays are useful for storing a set of retrieved values or for passing a set of values that are to be inserted into a table.

To use a host-variable array in an SQL statement, specify any valid host-variable array that is declared according to the host language rules. You can specify host-variable arrays in C or C++, COBOL, and PL/I. You must declare the array in the host program before you use it.

Host-variable arrays are defined by statements of the host language, as explained in the following topics:

- “Host-variable arrays in C and C++” on page 595
- “Host-variable arrays in COBOL” on page 660
- “Host-variable arrays in PL/I” on page 712

Tip: Host-variable arrays are not supported for assembler, FORTRAN, or REXX programs. However, you can use SQL descriptor areas (SQLDA) to achieve similar results in any host language. For more information see “Defining SQL descriptor areas (SQLDA)” on page 476.

Host-variable arrays can be referenced only as a simple reference in the following contexts. In syntax diagrams, host-variable-array designates a reference to a host-variable array.

- In a FETCH statement for a multiple-row fetch. See FETCH (Db2 SQL).
- In an INSERT statement with multiple rows of source data. This is also referred to as multiple-row insert. See INSERT (Db2 SQL).
• In a MERGE statement with multiple rows of source data. See MERGE (Db2 SQL).
• In an EXECUTE statement to provide a value for a parameter marker in a dynamic multi-row INSERT or MERGE. See EXECUTE (Db2 SQL).

Related concepts
Host-variable arrays in PL/I, C, C++, and COBOL (Db2 SQL)

Host-variable arrays
You can use host-variable arrays to pass a data array between Db2 and your application. A host-variable array is a data array that is declared in the host language to be used within an SQL statement.

Related tasks
Inserting multiple rows of data from host-variable arrays
Use host-variable arrays in your INSERT statement when you do not know at least some of the values to insert until the program runs.

Retrieving multiple rows of data into host-variable arrays
If you know that your query returns multiple rows, you can specify host-variable arrays to store the retrieved column values.

Retrieving multiple rows of data into host-variable arrays
If you know that your query returns multiple rows, you can specify host-variable arrays to store the retrieved column values.

About this task
You can use host-variable arrays to specify a program data area to contain multiple rows of column values. A Db2 rowset cursor enables an application to retrieve and process a set of rows from the result table of the cursor.

Related concepts
Host-variable arrays
You can use host-variable arrays to pass a data array between Db2 and your application. A host-variable array is a data array that is declared in the host language to be used within an SQL statement.

Host-variable arrays in PL/I, C, C++, and COBOL (Db2 SQL)

Related tasks
Accessing data by using a rowset-positioned cursor
A rowset-positioned cursor is a cursor that can return one or more rows for a single fetch operation. The cursor is positioned on the set of rows that are to be fetched.

Executing SQL statements by using a rowset cursor
You can use rowset cursors to execute multiple-row FETCH statements, positioned UPDATE statements, and positioned DELETE statements.

Inserting multiple rows of data from host-variable arrays
Use host-variable arrays in your INSERT statement when you do not know at least some of the values to insert until the program runs.

Inserting multiple rows of data from host-variable arrays
Use host-variable arrays in your INSERT statement when you do not know at least some of the values to insert until the program runs.

About this task
You can use a form of the INSERT statement or MERGE statement to insert multiple rows from values that are provided in host-variable arrays. Each array contains values for a column of the target table. The first value in an array corresponds to the value for that column for the first inserted row, the second value in the array corresponds to the value for the column in the second inserted row, and so on. Db2 determines the attributes of the values based on the declaration of the array.
Example

You can insert the number of rows that are specified in the host variable NUM-ROWS by using the following INSERT statement:

```sql
EXEC SQL
  INSERT INTO DSN8C10.ACT
  (ACTNO, ACTKWD, ACTDESC)
  VALUES (:HVA1, :HVA2, :HVA3)
FOR :NUM-ROWS ROWS
END-EXEC.
```

Assume that the host-variable arrays HVA1, HVA2, and HVA3 have been declared and populated with the values that are to be inserted into the ACTNO, ACTKWD, and ACTDESC columns. The NUM-ROWS host variable specifies the number of rows that are to be inserted, which must be less than or equal to the dimension of each host-variable array.

Related concepts
Host-variable arrays in PL/I, C, C++, and COBOL (Db2 SQL)

Related tasks
Retrieving multiple rows of data into host-variable arrays

If you know that your query returns multiple rows, you can specify host-variable arrays to store the retrieved column values.

Related reference
INSERT (Db2 SQL)
MERGE (Db2 SQL)

Inserting null values into columns by using indicator variables or arrays

If you need to insert null values into a column, using an indicator variable or array is an easy way to do so. An indicator variable or array is associated with a particular host variable or host-variable array.

Procedure

To insert null values into columns by using indicator variables or arrays:

1. Define an indicator variable or array for a particular host variable or array.
2. Assign a negative value to the indicator variable or array.
3. Issue the appropriate INSERT, UPDATE, or MERGE statement with the host variable or array and its indicator variable or array.

When Db2 processes INSERT, UPDATE, and MERGE statements, it checks the indicator variable if one exists. If the indicator variable is negative, the column value is null. If the indicator variable is greater than -1, the associated host variable contains a value for the column.

Examples

Example of setting a column value to null by using an indicator variable

Suppose your program reads an employee ID and a new phone number and must update the employee table with the new number. The new number could be missing if the old number is incorrect, but a new number is not yet available. If the new value for column PHONENO might be null, you can use an indicator variable, as shown in the following UPDATE statement.

```sql
EXEC SQL
  UPDATE DSN8C10.EMP
  SET PHONENO = :NEWPHONE:PHONEIND
  WHERE EMPNO = :EMPID
END-EXEC.
```
When NEWPHONE contains a non-null value, set the indicator variable PHONEIND to zero by preceding the UPDATE statement with the following line:

```sql
MOVE 0 TO PHONEIND.
```

When NEWPHONE contains a null value, set PHONEIND to a negative value by preceding the UPDATE statement with the following line:

```sql
MOVE -1 TO PHONEIND.
```

**Example of setting a column value to null by using an indicator variable array**

Assume that host-variable arrays hva1 and hva2 have been populated with values that are to be inserted into the ACTNO and ACTKWD columns. Assume the ACTDESC column allows nulls. To set the ACTDESC column to null, assign -1 to the elements in its indicator array, ind3, as shown in the following example:

```sql
/* Initialize each indicator array */
for (i=0; i<10; i++) {
  ind1[i] = 0;
  ind2[i] = 0;
  ind3[i] = -1;
}
EXEC SQL
  INSERT INTO DSN8C10.ACT
    (ACTNO, ACTKWD, ACTDESC)
  VALUES (:hva1:ind1, :hva2:ind2, :hva3:ind3)
  FOR 10 ROWS;
```

Db2 ignores the values in the hva3 array and assigns the values in the ACTDESC column to null for the 10 rows that are inserted.

**Related tasks**

**Declaring host variables and indicator variables**

You can use host variables and indicator variables in SQL statements in your program to pass data between Db2 and your application.

**Retrieving a single row of data into a host structure**

If you know that your query returns multiple column values for only one row, you can specify a host structure to contain the column values.

**About this task**

In the following example, assume that your COBOL program includes the following SQL statement:

```sql
EXEC SQL
  SELECT EMPNO, FIRSTNAME, MIDINIT, LASTNAME, WORKDEPT
  INTO :EMPNO, :FIRSTNAME, :MIDINIT, :LASTNAME, :WORKDEPT
  FROM DSN8C10.VEMP
  WHERE EMPNO = :EMPID
END-EXEC.
```

If you want to avoid listing host variables, you can substitute the name of a structure, say :PEMP, that contains :EMPNO, :FIRSTNAME, :MIDINIT, :LASTNAME, and :WORKDEPT. The example then reads:

```sql
EXEC SQL
  SELECT EMPNO, FIRSTNAME, MIDINIT, LASTNAME, WORKDEPT
  INTO :PEMP
  FROM DSN8C10.VEMP
  WHERE EMPNO = :EMPID
END-EXEC.
```

You can declare a host structure yourself, or you can use DCLGEN to generate a COBOL record description, PL/I structure declaration, or C structure declaration that corresponds to the columns of a table.
**Related concepts**

**DCLGEN (declarations generator)**
Your program should declare the tables and views that it accesses. The Db2 declarations generator, DCLGEN, produces these DECLARE statements for C, COBOL, and PL/I programs, so that you do not need to code the statements yourself. DCLGEN also generates corresponding host variable structures.

**Host structures**
Use host structures to pass a group of host variables between Db2 and your application.

**Example: Adding DCLGEN declarations to a library**
You can use DCLGEN to generate table and variable declarations for C, COBOL, and PL/I programs. If you store these declarations in a library, you can later integrate them into your program with a single SQL INCLUDE statement.

## Including dynamic SQL in your program

Dynamic SQL is prepared and executed while the program is running.

### Before you begin

Before you use dynamic SQL, consider whether static SQL or dynamic SQL is the best technique for your application, and consider the type of dynamic SQL that you want to use. Also consider the performance implications of using dynamic SQL in application programs. For information about methods that you can use to improve the performance of dynamic SQL statements, see [Improving dynamic SQL performance (Db2 Performance)](http://www.ibm.com/support/knowledgecenter/SSEPGG_12.1.0/com.ibm.db2.luw.dataset.sqlref.doc/doc/cpg_optimize.html).

### About this task

**Introductory concepts**

- Submitting SQL statements to Db2 (Introduction to Db2 for z/OS)
- Dynamic SQL applications (Introduction to Db2 for z/OS)

Dynamic SQL prepares and executes the SQL statements within a program, while the program is running. You can issue dynamic SQL statements in the following contexts:

**Interactive SQL**
A user enters SQL statements through SPUFI, the command line processor, or an interactive tool, such as QMF for Workstation. Db2 prepares and executes those statements as dynamic SQL statements.

**Embedded dynamic SQL**
Your application puts the SQL source in host variables and includes PREPARE and EXECUTE statements that tell Db2 to prepare and run the contents of those host variables at run time. You must precompile and bind programs that include embedded dynamic SQL.

**Deferred embedded SQL**
Deferred embedded SQL statements are neither fully static nor fully dynamic. Like static statements, deferred embedded SQL statements are embedded within applications; however, like dynamic statements, they are prepared at run time. Db2 processes the deferred embedded SQL statements with bind-time rules. For example, Db2 uses the authorization ID and qualifier (that are determined at bind time) as the plan or package owner.

**Dynamic SQL executed through ODBC or JDBC functions**
Your application contains ODBC function calls that pass dynamic SQL statements as arguments. You do not need to precompile and bind programs that use ODBC function calls.

JDBC application support lets you write dynamic SQL applications in Java.

For most Db2 users, **static SQL**, which is embedded in a host language program and bound before the program runs, provides a straightforward, efficient path to Db2 data. You can use static SQL when you know before run time what SQL statements your application needs to execute.
Related tasks
- Setting limits for system resource usage by using the resource limit facility (Db2 Performance)
- Improving dynamic SQL performance by enabling the dynamic statement cache (Db2 Performance)

Related information
- Dynamic Statement Cache (white paper)

Differences between static and dynamic SQL
Static and dynamic SQL are each appropriate for different circumstances. You should consider the differences between the two when determining whether static SQL or dynamic SQL is best for your application.

Flexibility of static SQL with host variables

Introductory concepts
- Static SQL (Introduction to Db2 for z/OS)
- Static SQL applications (Introduction to Db2 for z/OS)
- Submitting SQL statements to Db2 (Introduction to Db2 for z/OS)
- Dynamic SQL applications (Introduction to Db2 for z/OS)

When you use static SQL, you cannot change the form of SQL statements unless you make changes to the program. However, you can increase the flexibility of static statements by using host variables.

Example: In the following example, the UPDATE statement can update the salary of any employee. At bind time, you know that salaries must be updated, but you do not know until run time whose salaries should be updated, and by how much.

```cobol
01  IOAREA.
   02  EMPID              PIC X(06).
   02  NEW-SALARY         PIC S9(7)V9(2) COMP-3.
   : (Other declarations)
READ CARDIN RECORD INTO IOAREA
   AT END MOVE 'N' TO INPUT-SWITCH.
   : (Other COBOL statements)
EXEC SQL
   UPDATE DSN8C10.EMP
   SET SALARY = :NEW-SALARY
   WHERE EMPNO = :EMPID
END-EXEC.
```

The statement (UPDATE) does not change, nor does its basic structure, but the input can change the results of the UPDATE statement.

Flexibility of dynamic SQL
What if a program must use different types and structures of SQL statements? If there are so many types and structures that it cannot contain a model of each one, your program might need dynamic SQL.

You can use one of the following programs to execute dynamic SQL:

Db2 Query Management Facility (QMF)
- Provides an alternative interface to Db2 that accepts almost any SQL statement

SPUFI
- Accepts SQL statements from an input data set, and then processes and executes them dynamically

command line processor
- Accepts SQL statements from a UNIX System Services environment.

Limitations of dynamic SQL
You cannot use some of the SQL statements dynamically.
**Dynamic SQL processing**

A program that provides for dynamic SQL accepts as input, or generates, an SQL statement in the form of a character string. You can simplify the programming if you can plan the program not to use SELECT statements, or to use only those that return a known number of values of known types. In the most general case, in which you do not know in advance about the SQL statements that will execute, the program typically takes these steps:

1. Translates the input data, including any parameter markers, into an SQL statement
2. Prepares the SQL statement to execute and acquires a description of the result table
3. Obtains, for SELECT statements, enough main storage to contain retrieved data
4. Executes the statement or fetches the rows of data
5. Processes the information returned
6. Handles SQL return codes.

**Performance of static and dynamic SQL**

To access Db2 data, an SQL statement requires an access path. Two big factors in the performance of an SQL statement are the amount of time that Db2 uses to determine the access path at runtime and whether the access path is efficient. Db2 determines the access path for a statement at either of these times:

- When you bind the plan or package that contains the SQL statement
- When the SQL statement executes

The time at which Db2 determines the access path depends on these factors:

- Whether the statement is executed statically or dynamically
- Whether the statement contains input host variables
- Whether the statement contains a declared global temporary table.

**Static SQL statements with no input host variables**

For static SQL statements that do not contain input host variables, Db2 determines the access path when you bind the plan or package. This combination yields the best performance because the access path is already determined when the program executes.

**Static SQL statements with input host variables**

For static SQL statements that have input host variables, the time at which Db2 determines the access path depends on the REOPT bind option that you specify: REOPT(NONE) or REOPT(ALWAYS). REOPT(NONE) is the default. Do not specify REOPT(AUTO) or REOPT(ONCE); these options are applicable only to dynamic statements. Db2 ignores REOPT(ONCE) and REOPT(AUTO) for static SQL statements, because Db2 caches only dynamic SQL statements.

If you specify REOPT(NONE), Db2 determines the access path at bind time, just as it does when there are no input variables.

If you specify REOPT(ALWAYS), Db2 determines the access path at bind time and again at runtime, using the values of the following types of input variables:

- Host variables
- Parameter markers
- Special registers

Db2 must spend extra time determining the access path for statements at runtime. However if Db2 determines a significantly better access path using the variable values, you might see an overall performance improvement. With REOPT(ALWAYS), Db2 optimizes statements using known literal values. Knowing the literal values can help Db2 to choose a more efficient access path when the columns contain...
skewed data. Db2 can also recognize which partitions qualify if there are search conditions with host variables on the limit keys of partitioned table spaces.

With REOPT(ALWAYS) Db2 does not start the optimization over from the beginning. For example Db2 does not perform query transformations based on the literal values. Consequently, static SQL statements that use host variables optimized with REOPT(ALWAYS) and similar SQL statements that use explicit literal values might result in different access paths.

**Dynamic SQL statements**

For dynamic SQL statements, Db2 determines the access path at run time, when the statement is prepared. The repeating cost of preparing a dynamic statement can make the performance worse than that of static SQL statements. However, if you execute the same SQL statement often, you can use the dynamic statement cache to decrease the number of times that those dynamic statements must be prepared.

**Dynamic SQL statements with input host variables**

When you bind applications that contain dynamic SQL statements with input host variables, consider using the REOPT(ALWAYS), REOPT(ONCE), or REOPT(AUTO) bind options, instead of the REOPT(NONE) option.

Use REOPT(ALWAYS) when you are not using the dynamic statement cache. Db2 determines the access path for statements at each EXECUTE or OPEN of the statement. This option ensures the best access path for a statement, but using REOPT(ALWAYS) can increase the cost of frequently used dynamic SQL statements.

Consequently, the REOPT(ALWAYS) option is not a good choice for high-volume sub-second queries. For high-volume fast running queries, the repeating cost of prepare can exceed the execution cost of the statement. Statements that are processed under the REOPT(ALWAYS) option are excluded from the dynamic statement cache even if dynamic statement caching is enabled because Db2 cannot reuse access paths when REOPT(ALWAYS) is specified.

Use REOPT(ONCE) or REOPT(AUTO) when you are using the dynamic statements cache:

- If you specify REOPT(ONCE), Db2 determines and the access path for statements only at the first EXECUTE or OPEN of the statement. It saves that access path in the dynamic statement cache and uses it until the statement is invalidated or removed from the cache. This reuse of the access path reduces the prepare cost of frequently used dynamic SQL statements that contain input host variables; however, it does not account for changes to parameter marker values for dynamic statements.

  The REOPT(ONCE) option is ideal for ad-hoc query applications such as SPUFI, DSNTEP2, DSNTEP4, DSNTIAUL, and QMF Db2 can better optimize statements knowing the literal values for special registers such as CURRENT DATE and CURRENT TIMESTAMP, rather than using default filter factor estimates.

- If you specify REOPT(AUTO), Db2 determines the access path at run time. For each execution of a statement with parameter markers, Db2 generates a new access path if it determines that a new access path is likely to improve performance.

**Coding PREPARE statements for efficient optimization**

You should code your PREPARE statements to minimize overhead. With REOPT(AUTO), REOPT(ALWAYS), and REOPT(ONCE), Db2 prepares an SQL statement at the same time as it processes OPEN or EXECUTE for the statement. That is, Db2 processes the statement as if you specify DEFER(PREPARE). However, Db2 prepares the statement twice in the following situations:

- Your program issues the DESCRIBE statement before the OPEN statement
- You issue the PREPARE statement with the INTO parameter

For the first prepare, Db2 determines the access path without using input variable values. For the second prepare, Db2 uses the input variable values to determine the access path. This extra prepare can decrease performance.
If you specify REOPT(ALWAYS), Db2 prepares the statement twice each time it is run.

If you specify REOPT(ONCE), Db2 prepares the statement twice only when the statement has never been saved in the cache. If the statement has been prepared and saved in the cache, Db2 will use the saved version of the statement to complete the DESCRIBE statement.

If you specify REOPT(AUTO), Db2 initially prepares the statement without using input variable values. If the statement has been saved in the cache, for the subsequent OPEN or EXECUTE, Db2 determines if a new access path is needed according to the input variable values.

For a statement that uses a cursor, you can avoid the double prepare by placing the DESCRIBE statement after the OPEN statement in your program.

If you use predictive governing, and a dynamic SQL statement that is bound with either REOPT(ALWAYS) or REOPT(ONCE) exceeds a predictive governing warning threshold, your application does not receive a warning SQLCODE. However, it will receive an error SQLCODE from the OPEN or EXECUTE statement.

Related tasks
Reoptimizing SQL statements at run time (Db2 Performance)
Improving dynamic SQL performance by enabling the dynamic statement cache (Db2 Performance)
Related reference
Characteristics of SQL statements in Db2 for z/OS (Db2 SQL)
REOPT bind option (Db2 Commands)

Possible host languages for dynamic SQL applications

Programs that use dynamic SQL are usually written in assembler, C, PL/I, REXX, and COBOL. All SQL statements in REXX programs are considered dynamic SQL.

You can write non-SELECT and fixed-list SELECT statements in any of the Db2 supported languages. A program containing a varying-list SELECT statement is more difficult to write in Fortran, because the program cannot run without the help of a subroutine to manage address variables (pointers) and storage allocation.

Most of the examples in this topic are in PL/I. Longer examples in the form of complete programs are available in the sample applications:

**DSNTEP2**
Processes both SELECT and non-SELECT statements dynamically. (PL/I).

**DSNTIAD**
Processes only non-SELECT statements dynamically. (Assembler).

**DSNTIAUL**
Processes SELECT statements dynamically. (Assembler).

Library `prefix.SDSNSAMP` contains the sample programs. You can view the programs online, or you can print them using ISPF, IEBPTPCH, or your own printing program.

You can use all forms of dynamic SQL in all supported versions of COBOL.

Related concepts
Sample COBOL dynamic SQL program
You can code dynamic varying-list SELECT statements in a COBOL program. *Varying-List SELECT statements* are statements for which you do not know the number or data types of columns that are to be returned when you write the program.

### Including dynamic SQL for non-SELECT statements in your program

The easiest way to use dynamic SQL is to use non-SELECT statements. Because you do not need to dynamically allocate any main storage, you can write your program in any host language, including Fortran.

#### Procedure

Your program must take the following steps:

1. Include an SQLCA. The requirements for an SQL communications area (SQLCA) are the same as for static SQL statements. For REXX, Db2 includes the SQLCA automatically.
2. Load the input SQL statement into a data area. The procedure for building or reading the input SQL statement is not discussed here; the statement depends on your environment and sources of information. You can read in complete SQL statements, or you can get information to build the statement from data sets, a user at a terminal, previously set program variables, or tables in the database.
   
   If you attempt to execute an SQL statement dynamically that Db2 does not allow, you get an SQL error.
3. Execute the statement. You can use either of these methods:
   
   - EXECUTE IMMEDIATE
   - PREPARE and EXECUTE
4. Handle any errors that might result. The requirements are the same as those for static SQL statements. The return code from the most recently executed SQL statement appears in the host variables SQLCODE and SQLSTATE or corresponding fields of the SQLCA.

#### Related concepts

- **Sample dynamic and static SQL in a C program**
  Programs that access Db2 can contain static SQL, dynamic SQL, or both.

- **Assembler applications that issue SQL statements**
  You can code SQL statements in assembler programs wherever you can use executable statements.

- **C and C++ applications that issue SQL statements**
  You can code SQL statements in a C or C++ program wherever you can use executable statements.

- **COBOL applications that issue SQL statements**
  You can code SQL statements in certain COBOL program sections.

- **Fortran applications that issue SQL statements**
  You can code SQL statements in a Fortran program wherever you can place executable statements. If the SQL statement is within an IF statement, the precompiler generates any necessary THEN and END IF statements.

- **PL/I applications that issue SQL statements**
  You can code SQL statements in a PL/I program wherever you can use executable statements.

- **REXX applications that issue SQL statements**
  You can code SQL statements in a REXX programs wherever you can use REXXX commands.

#### Related tasks

- **Checking the execution of SQL statements**
  After executing an SQL statement, your program should check for any errors before you commit the data and handle the errors that they represent.

- **Dynamically executing an SQL statement by using EXECUTE IMMEDIATE**
In certain situations, you might want your program to prepare and dynamically execute a statement immediately after reading it.

Dynamically executing an SQL statement by using PREPARE and EXECUTE
As an alternative to executing an SQL statement immediately after it is read, you can prepare and execute the SQL statement in two steps. This two-step method is useful when you need to execute an SQL statement multiple times with different values.

Including dynamic SQL for fixed-list SELECT statements in your program
A fixed-list SELECT statement returns rows that contain a known number of values of a known type. When you use this type of statement, you know in advance exactly what kinds of host variables you need to declare to store the results.

About this task
The term "fixed-list" does not imply that you must know in advance how many rows of data will be returned. However, you must know the number of columns and the data types of those columns. A fixed-list SELECT statement returns a result table that can contain any number of rows; your program looks at those rows one at a time, using the FETCH statement. Each successive fetch returns the same number of values as the last, and the values have the same data types each time. Therefore, you can specify host variables as you do for static SQL.

An advantage of the fixed-list SELECT is that you can write it in any of the programming languages that Db2 supports. Varying-list dynamic SELECT statements require assembler, C, PL/I, and COBOL.

For example, suppose that your program retrieves last names and phone numbers by dynamically executing SELECT statements of this form:

```
SELECT LASTNAME, PHONENO FROM DSN8C10.EMP
WHERE ... ;
```

The program reads the statements from a terminal, and the user determines the WHERE clause.

As with non-SELECT statements, your program puts the statements into a varying-length character variable; call it DSTRING. Eventually you prepare a statement from DSTRING, but first you must declare a cursor for the statement and give it a name.

Procedure
To execute a fixed-list SELECT statement dynamically, your program must:

1. Include an SQLCA.
2. Load the input SQL statement into a data area.
   The preceding two steps are exactly the same including dynamic SQL for non-SELECT statements in your program.
3. Declare a cursor for the statement name.
   Dynamic SELECT statements cannot use INTO. Therefore, you must use a cursor to put the results into host variables.
   For example, when you declare the cursor, use the statement name (call it STMT), and give the cursor itself a name (for example, C1):
   ```
   EXEC SQL DECLARE C1 CURSOR FOR STMT;
   ```
4. Prepare the statement.
   Prepare a statement (STMT) from DSTRING. This is one possible PREPARE statement:
   ```
   EXEC SQL PREPARE STMT FROM :DSTRING ATTRIBUTES :ATTRVAR;
   ```
   ATTRVAR contains attributes that you want to add to the SELECT statement, such as FETCH FIRST 10 ROWS ONLY or OPTIMIZE for 1 ROW. In general, if the SELECT statement has attributes that
conflict with the attributes in the PREPARE statement, the attributes on the SELECT statement take precedence over the attributes on the PREPARE statement. However, in this example, the SELECT statement in DSTRING has no attributes specified, so Db2 uses the attributes in ATTRVAR for the SELECT statement.

As with non-SELECT statements, the fixed-list SELECT could contain parameter markers. However, this example does not need them.

5. Open the cursor.
   The OPEN statement evaluates the SELECT statement named STMT.
   For example, without parameter markers, use this statement:

   ```sql
   EXEC SQL OPEN C1;
   ```

   If STMT contains parameter markers, you must use the USING clause of OPEN to provide values for all of the parameter markers in STMT. If four parameter markers are in STMT, you need the following statement:

   ```sql
   EXEC SQL OPEN C1 USING :PARM1, :PARM2, :PARM3, :PARM4;
   ```

6. Fetch rows from the result table.
   For example, your program could repeatedly execute a statement such as this:

   ```sql
   EXEC SQL FETCH C1 INTO :NAME, :PHONE;
   ```

   The key feature of this statement is the use of a list of host variables to receive the values returned by FETCH. The list has a known number of items (in this case, two items, :NAME and :PHONE) of known data types (both are character strings, of lengths 15 and 4, respectively).

   You can use this list in the FETCH statement only because you planned the program to use only fixed-list SELECTs. Every row that cursor C1 points to must contain exactly two character values of appropriate length. If the program is to handle anything else, it must use the techniques for including dynamic SQL for varying-list SELECT statements in your program.

7. Close the cursor.
   This step is the same as for static SQL.
   A WHENEVER NOT FOUND statement in your program can name a routine that contains this statement:

   ```sql
   EXEC SQL CLOSE C1;
   ```

8. Handle any resulting errors. This step is the same as for static SQL, except for the number and types of errors that can result.

Related concepts
Sample dynamic and static SQL in a C program
Programs that access Db2 can contain static SQL, dynamic SQL, or both.

Assembler applications that issue SQL statements
You can code SQL statements in assembler programs wherever you can use executable statements.

C and C++ applications that issue SQL statements
You can code SQL statements in a C or C++ program wherever you can use executable statements.

COBOL applications that issue SQL statements
You can code SQL statements in certain COBOL program sections.

Fortran applications that issue SQL statements
You can code SQL statements in a Fortran program wherever you can place executable statements. If the SQL statement is within an IF statement, the precompiler generates any necessary THEN and END IF statements.

PL/I applications that issue SQL statements
You can code SQL statements in a PL/I program wherever you can use executable statements.

REXX applications that issue SQL statements
You can code SQL statements in a REXX program wherever you can use REXX commands.

Related tasks
Including dynamic SQL for non-SELECT statements in your program
The easiest way to use dynamic SQL is to use non-SELECT statements. Because you do not need to dynamically allocate any main storage, you can write your program in any host language, including Fortran.

Including dynamic SQL for varying-list SELECT statements in your program
A varying-list SELECT statement returns rows that contain an unknown number of values of unknown type. When you use this type of statement, you do not know in advance exactly what kinds of host variables you need to declare for storing the results.

Including dynamic SQL for varying-list SELECT statements in your program
A varying-list SELECT statement returns rows that contain an unknown number of values of unknown type. When you use this type of statement, you do not know in advance exactly what kinds of host variables you need to declare for storing the results.

About this task
Because the varying-list SELECT statement requires pointer variables for the SQL descriptor area, you cannot issue it from a Fortran program. A Fortran program can call a subroutine written in a language that supports pointer variables (such as PL/I or assembler), if you need to use a varying-list SELECT statement.

Procedure
To execute a varying-list SELECT statement dynamically, your program must follow these steps:
1. Include an SQLCA.
   Db2 performs this step for a REXX program.
2. Load the input SQL statement into a data area.
3. Prepare and execute the statement. This step is more complex than for fixed-list SELECTs.
   It involves the following steps:
   a) Include an SQLDA (SQL descriptor area).
      Db2 performs this step for a REXX program.
   b) Declare a cursor and prepare the variable statement.
   c) Obtain information about the data type of each column of the result table.
   d) Determine the main storage needed to hold a row of retrieved data.
      You do not perform this step for a REXX program.
   e) Put storage addresses in the SQLDA to tell where to put each item of retrieved data.
   f) Open the cursor.
   g) Fetch a row.
   h) Eventually close the cursor and free main storage.
   Additional complications exist for statements with parameter markers.
4. Handle any errors that might result.
Examples

Preparing a varying-list SELECT statement

Suppose that your program dynamically executes SQL statements, but this time without any limits on their form. Your program reads the statements from a terminal, and you know nothing about them in advance. They might not even be SELECT statements.

As with non-SELECT statements, your program puts the statements into a varying-length character variable; call it DSTRING. Your program goes on to prepare a statement from the variable and then give the statement a name; call it STMT.

Now, the program must find out whether the statement is a SELECT. If it is, the program must also find out how many values are in each row, and what their data types are. The information comes from an SQL descriptor area (SQLDA).

SQL descriptor area (SQLDA)

The SQLDA is a structure that is used to communicate with your program, and storage for it is usually allocated dynamically at run time.

To include the SQLDA in a PL/I or C program, use:

```sql
EXEC SQL INCLUDE SQLDA;
```

For assembler, use this in the storage definition area of a CSECT:

```sql
EXEC SQL INCLUDE SQLDA
```

For COBOL, use:

```sql
EXEC SQL INCLUDE SQLDA END-EXEC.
```

You cannot include an SQLDA in a Fortran, or REXX program.

Obtaining information about the SQL statement

An SQLDA can contain a variable number of occurrences of SQLVAR, each of which is a set of five fields that describe one column in the result table of a SELECT statement.

The number of occurrences of SQLVAR depends on the following factors:

- The number of columns in the result table you want to describe.
- Whether you want the PREPARE or DESCRIBE to put both column names and labels in your SQLDA. This is the option USING BOTH in the PREPARE or DESCRIBE statement.
- Whether any columns in the result table are LOB types or distinct types.

The following table shows the minimum number of SQLVAR instances you need for a result table that contains \( n \) columns.

<table>
<thead>
<tr>
<th>Type of DESCRIBE and contents of result table</th>
<th>Not USING BOTH</th>
<th>USING BOTH</th>
</tr>
</thead>
<tbody>
<tr>
<td>No distinct types or LOBs</td>
<td>( n )</td>
<td>( 2 \times n )</td>
</tr>
<tr>
<td>Distinct types but no LOBs</td>
<td>( 2 \times n )</td>
<td>( 3 \times n )</td>
</tr>
<tr>
<td>LOBs but no distinct types</td>
<td>( 2 \times n )</td>
<td>( 2 \times n )</td>
</tr>
<tr>
<td>LOBs and distinct types</td>
<td>( 2 \times n )</td>
<td>( 3 \times n )</td>
</tr>
</tbody>
</table>

An SQLDA with \( n \) occurrences of SQLVAR is referred to as a single SQLDA, an SQLDA with \( 2 \times n \) occurrences of SQLVAR a double SQLDA, an SQLDA with \( 3 \times n \) occurrences of SQLVAR a triple SQLDA.

A program that admits SQL statements of every kind for dynamic execution has two choices:
• Provide the largest SQLDA that it could ever need. The maximum number of columns in a result table is 750, so an SQLDA for 750 columns occupies 33 016 bytes for a single SQLDA, 66 016 bytes for a double SQLDA, or 99 016 bytes for a triple SQLDA. Most SELECT statements do not retrieve 750 columns, so the program does not usually use most of that space.

• Provide a smaller SQLDA, with fewer occurrences of SQLVAR. From this the program can find out whether the statement was a SELECT and, if it was, how many columns are in its result table. If more columns are in the result than the SQLDA can hold, Db2 returns no descriptions. When this happens, the program must acquire storage for a second SQLDA that is long enough to hold the column descriptions, and ask Db2 for the descriptions again. Although this technique is more complicated to program than the first, it is more general.

How many columns should you allow? You must choose a number that is large enough for most of your SELECT statements, but not too wasteful of space; 40 is a good compromise. To illustrate what you must do for statements that return more columns than allowed, the example in this discussion uses an SQLDA that is allocated for at least 100 columns.

Declaring a cursor for the statement
As before, you need a cursor for the dynamic SELECT. For example, write:

```sql
EXEC SQL
  DECLARE C1 CURSOR FOR STMT;
```

Preparing the statement using the minimum SQLDA
Suppose that your program declares an SQLDA structure with the name MINSQLDA, having 100 occurrences of SQLVAR and SQLN set to 100. To prepare a statement from the character string in DSTRING and also enter its description into MINSQLDA, write this:

```sql
EXEC SQL PREPARE STMT FROM :DSTRING;
EXEC SQL DESCRIBE STMT INTO :MINSQLDA;
```

Equivalently, you can use the INTO clause in the PREPARE statement:

```sql
EXEC SQL
  PREPARE STMT INTO :MINSQLDA FROM :DSTRING;
```

Do not use the USING clause in either of these examples. At the moment, only the minimum SQLDA is in use. The following figure shows the contents of the minimum SQLDA in use.

![Figure 30. The minimum SQLDA structure](image)

**SQLN determines what SQLVAR gets**
The SQLN field, which you must set before using DESCRIBE (or PREPARE INTO), tells how many occurrences of SQLVAR the SQLDA is allocated for. If DESCRIBE needs more than that, the results of the DESCRIBE depend on the contents of the result table. Let \( n \) indicate the number of columns in the result table. Then:

• If the result table contains at least one distinct type column but no LOB columns, you do not specify USING BOTH, and \( n <= \text{SQLN} < 2^*n \), then Db2 returns base SQLVAR information in the first \( n \) SQLVAR occurrences, but no distinct type information. Base SQLVAR information includes:
  - Data type code
  - Length attribute (except for LOBs)
  - Column name or label
  - Host variable address
  - Indicator variable address

• Otherwise, if SQLN is less than the minimum number of SQLVARs specified in the table above, then Db2 returns no information in the SQLVARs.
Regardless of whether your SQLDA is big enough, whenever you execute DESCRIBE, Db2 returns the following values, which you can use to build an SQLDA of the correct size:

- SQLD is 0 if the SQL statement is not a SELECT. Otherwise, SQLD is the number of columns in the result table. The number of SQLVAR occurrences you need for the SELECT depends on the value in the seventh byte of SQLDAID.
- The seventh byte of SQLDAID is 2 if each column in the result table requires two SQLVAR entries. The seventh byte of SQLDAID is 3 if each column in the result table requires three SQLVAR entries.

**If the statement is not a SELECT**

To find out if the statement is a SELECT, your program can query the SQLD field in MINSQLDA. If the field contains 0, the statement is not a SELECT, the statement is already prepared, and your program can execute it. If no parameter markers are in the statement, you can use:

```sql
EXEC SQL EXECUTE STMT;
```

(If the statement does contain parameter markers, you must use an SQL descriptor area)

**Acquiring storage for a second SQLDA if needed**

Now you can allocate storage for a second, full-size SQLDA; call it FULSQLDA. The following figure shows its structure.

```
<table>
<thead>
<tr>
<th>16-byte fixed header</th>
<th>SQLDAID</th>
<th>SQLDABC</th>
<th>S LN</th>
<th>S Q LD</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQLVAR set of fields</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(element 1, 44 bytes)</td>
<td>SQLTYPE</td>
<td>SQLLEN</td>
<td>SQLDATA</td>
<td>S Q L N D</td>
</tr>
<tr>
<td>Other SQLVAR sets of</td>
<td>SQLLONGL</td>
<td>Reserved</td>
<td>S Q LD A T AL</td>
<td>m**</td>
</tr>
<tr>
<td>fields (elements 2, 3,</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>and so on, 44 bytes each)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SQLVAR2 set of fields</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(element 1, 44 bytes)</td>
<td>SQLNAME</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Other SQLVAR2 sets of</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>fields (elements 2, 3,</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>and so on, 44 bytes each)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

* The length of the character string in SQLNAME. SQLNAME is a 30-byte area immediately following the length field.

** The length of the character string in SQLTNAME. SQLTNAME is a 30-byte area immediately following the length field.

**Figure 31. The full-size SQLDA structure**

FULSQLDA has a fixed-length header of 16 bytes in length, followed by a varying-length section that consists of structures with the SQLVAR format. If the result table contains LOB columns or distinct type columns, a varying-length section that consists of structures with the SQLVAR2 format follows the structures with SQLVAR format. All SQLVAR structures and SQLVAR2 structures are 44 bytes long. The number of SQLVAR and SQLVAR2 elements you need is in the SQLD field of MINSQLDA, and the total length you need for FULSQLDA (16 + SQLD * 44) is in the SQLDABC field of MINSQLDA. Allocate that amount of storage.

**Describing the SELECT statement again**

After allocating sufficient space for FULSQLDA, your program must take these steps:

1. Put the total number of SQLVAR and SQLVAR2 occurrences in FULSQLDA into the S LN field of FULSQLDA. This number appears in the SQLD field of MINSQLDA.
2. Describe the statement again into the new SQLDA:
After the DESCRIBE statement executes, each occurrence of SQLVAR in the full-size SQLDA (FULSQLDA in our example) contains a description of one column of the result table in five fields. If an SQLVAR occurrence describes a LOB column or distinct type column, the corresponding SQLVAR2 occurrence contains additional information specific to the LOB or distinct type.

The following figure shows an SQLDA that describes two columns that are not LOB columns or distinct type columns.

<table>
<thead>
<tr>
<th>SQLDA header</th>
<th>SQLDA</th>
<th>8816</th>
<th>200</th>
<th>200</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQLVAR element 1 (44 bytes)</td>
<td>452 3 Undefined 0 8 WORKDEPT</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SQLVAR element 2 (44 bytes)</td>
<td>453 4 Undefined 0 7 PHONENO</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

*Figure 32. Contents of FULSQLDA after executing DESCRIBE*

**Acquiring storage to hold a row**

Before fetching rows of the result table, your program must:

1. Analyze each SQLVAR description to determine how much space you need for the column value.
2. Derive the address of some storage area of the required size.
3. Put this address in the SQLDATA field.

If the SQLTYPE field indicates that the value can be null, the program must also put the address of an indicator variable in the SQLIND field. The following figures show the SQL descriptor area after you take certain actions.

In the previous figure, the DESCRIBE statement inserted all the values except the first occurrence of the number 200. The program inserted the number 200 before it executed DESCRIBE to tell how many occurrences of SQLVAR to allow. If the result table of the SELECT has more columns than this, the SQLVAR fields describe nothing.

The first SQLVAR pertains to the first column of the result table (the WORKDEPT column). SQLVAR element 1 contains fixed-length character strings and does not allow null values (SQLTYPE=452); the length attribute is 3.

The following figure shows the SQLDA after your program acquires storage for the column values and their indicators, and puts the addresses in the SQLDATA fields of the SQLDA.

<table>
<thead>
<tr>
<th>SQLDA header</th>
<th>SQLDA</th>
<th>8816</th>
<th>200</th>
<th>200</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQLVAR element 1 (44 bytes)</td>
<td>452 3 Addr FLDA Addr FLDAI 8 WORKDEPT</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SQLVAR element 2 (44 bytes)</td>
<td>453 4 Addr FLDB Addr FLDBI 7 PHONENO</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

*Figure 33. SQL descriptor area after analyzing descriptions and acquiring storage*

The following figure shows the SQLDA after your program executes a FETCH statement.
The following table describes the values in the descriptor area.

<table>
<thead>
<tr>
<th>Value</th>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQLDA</td>
<td>SQLDAID</td>
<td>An &quot;eye-catcher&quot;</td>
</tr>
<tr>
<td>8816</td>
<td>SQLDABC</td>
<td>The size of the SQLDA in bytes (16 + 44 * 200)</td>
</tr>
<tr>
<td>200</td>
<td>SQLN</td>
<td>The number of occurrences of SQLVAR, set by the program</td>
</tr>
<tr>
<td>200</td>
<td>SQLD</td>
<td>The number of occurrences of SQLVAR actually used by the DESCRIBE statement</td>
</tr>
<tr>
<td>452</td>
<td>SQLTYPE</td>
<td>The value of SQLTYPE in the first occurrence of SQLVAR.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>It indicates that the first column contains fixed-length character strings, and does not allow nulls.</td>
</tr>
<tr>
<td>3</td>
<td>SQLLEN</td>
<td>The length attribute of the column</td>
</tr>
<tr>
<td>Undefined or CCSID value</td>
<td>SQLDATA</td>
<td>Bytes 3 and 4 contain the CCSID of a string column. Undefined for other types of columns.</td>
</tr>
<tr>
<td>Undefined</td>
<td>SQLIND</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>SQLNAME</td>
<td>The number of characters in the column name</td>
</tr>
<tr>
<td>WORKDEPT</td>
<td>SQLNAME+2</td>
<td>The column name of the first column</td>
</tr>
</tbody>
</table>

### Putting storage addresses in the SQLDA
After analyzing the description of each column, your program must replace the content of each SQLDATA field with the address of a storage area large enough to hold values from that column. Similarly, for every column that allows nulls, the program must replace the content of the SQLIND field. The content must be the address of a halfword that you can use as an indicator variable for the column. The program can acquire storage for this purpose, of course, but the storage areas used do not have to be contiguous.

Figure 33 on page 508 shows the content of the descriptor area before the program obtains any rows of the result table. Addresses of fields and indicator variables are already in the SQLVAR.

### Changing the CCSID for retrieved data
All Db2 string data has an encoding scheme and CCSID associated with it. When you select string data from a table, the selected data generally has the same encoding scheme and CCSID as the table. If the application uses some method, such as issuing the DECLARE VARIABLE statement, to change the CCSID of the selected data, the data is converted from the CCSID of the table to the CCSID that is specified by the application.

You can set the default application encoding scheme for a plan or package by specifying the value in the APPLICATION ENCODING field of the panel DEFAULTS FOR BIND PACKAGE or DEFAULTS FOR
BIND PLAN. The default application encoding scheme for the Db2 subsystem is the value that was specified in the APPLICATION ENCODING field of installation panel DSNTIPF.

If you want to retrieve the data in an encoding scheme and CCSID other than the default values, you can use one of the following techniques:

- For dynamic SQL, set the CURRENT APPLICATION ENCODING SCHEME special register before you execute the SELECT statements. For example, to set the CCSID and encoding scheme for retrieved data to the default CCSID for Unicode, execute this SQL statement:

  EXEC SQL SET CURRENT APPLICATION ENCODING SCHEME = 'UNICODE';

  The initial value of this special register is the application encoding scheme that is determined by the BIND option.

- For static and dynamic SQL statements that use host variables and host-variable arrays, use the DECLARE VARIABLE statement to associate CCSIDs with the host variables into which you retrieve the data. See “Setting the CCSID for host variables” on page 481 for information about this technique.

- For static and dynamic SQL statements that use a descriptor, set the CCSID for the retrieved data in the SQLDA. The following text describes that technique.

To change the encoding scheme for SQL statements that use a descriptor, set up the SQLDA, and then make these additional changes to the SQLDA:

1. Put the character + in the sixth byte of field SQLDAID.
2. For each SQLVAR entry:
   a. Set the length field of SQLNAME to 8.
   b. Set the first two bytes of the data field of SQLNAME to X'0000'.
   c. Set the third and fourth bytes of the data field of SQLNAME to the CCSID, in hexadecimal, in which you want the results to display, or to X'0000'. X'0000' indicates that Db2 should use the default CCSID If you specify a nonzero CCSID, it must meet one of the following conditions:
      - A row in catalog table SYSSTRINGS has a matching value for OUTCCSID.
      - The Unicode conversion services support conversion to that CCSID. See Building and using Dynamic Link Libraries (DLLs) (XL C/C++ Programming Guide) for information about the conversions supported.

If you are modifying the CCSID to retrieve the contents of an ASCII, EBCDIC, or Unicode table on a Db2 for z/OS system, and you previously executed a DESCRIBE statement on the SELECT statement that you are using to retrieve the data, the SQLDATA fields in the SQLDA that you used for the DESCRIBE contain the ASCII or Unicode CCSID for that table. To set the data portion of the SQLNAME fields for the SELECT, move the contents of each SQLDATA field in the SQLDA from the DESCRIBE to each SQLNAME field in the SQLDA for the SELECT. If you are using the same SQLDA for the DESCRIBE and the SELECT, be sure to move the contents of the SQLDATA field to SQLNAME before you modify the SQLDATA field for the SELECT.

For REXX, you set the CCSID in the stem.n.SQLUSECCSID field instead of setting the SQLDAID and SQLNAME fields.

For example, suppose that the table that contains WORKDEPT and PHONENO is defined with CCSID ASCII. To retrieve data for columns WORKDEPT and PHONENO in ASCII CCSID 437 (X'01B5'), change the SQLDA as shown in the following figure.
Specifying that DESCRIBE use column labels in the SQLNAME field

By default, DESCRIBE describes each column in the SQLNAME field by the column name. You can tell it to use column labels instead.

Restriction: You cannot use column labels with set operators (UNION, INTERSECT, and EXCEPT).

To specify that DESCRIBE use column labels in the SQLNAME field, specify one of the following options when you issue the DESCRIBE statement:

**USING LABELS**

Specifies that SQLNAME is to contain labels. If a column has no label, SQLNAME contains nothing.

**USING ANY**

Specifies that SQLNAME is to contain labels wherever they exist. If a column has no label, SQLNAME contains the column name.

**USING BOTH**

Specifies that SQLNAME is to contain both labels and column names, when both exist.

In this case, FULSQLDA must contain a second set of occurrences of SQLVAR. The first set contains descriptions of all the columns with column names; the second set contains descriptions with column labels.

If you choose this option, perform the following actions:

- Allocate a longer SQLDA for the second DESCRIBE statement ((16 + SQLD * 88 bytes) instead of (16 + SQLD * 44))
- Put double the number of columns (SLQD * 2) in the SQLN field of the second SQLDA.

These actions ensure that enough space is available. Otherwise, if not enough space is available, DESCRIBE does not enter descriptions of any of the columns.

```sql
EXEC SQL
DESCRIBE STMT INTO :FULSQLDA USING LABELS;
```

Some columns, such as those derived from functions or expressions, have neither name nor label; SQLNAME contains nothing for those columns. For example, if you use a UNION to combine two columns that do not have the same name and do not use a label, SQLNAME contains a string of length zero.

Describing tables with LOB and distinct type columns

In general, the steps that you perform when you prepare an SQLDA to select rows from a table with LOB and distinct type columns are similar to the steps that you perform if the table has no columns of this type. The only difference is that you need to analyze some additional fields in the SQLDA for LOB or distinct type columns.

For example, Suppose that you want to execute this SELECT statement:

```sql
SELECT USER, A_DOC FROM DOCUMENTS;
```
The USER column cannot contain nulls and is of distinct type ID, defined like this:

```
CREATE DISTINCT TYPE SCHEMA1.ID AS CHAR(20);
```

The A_DOC column can contain nulls and is of type CLOB(1M).

The result table for this statement has two columns, but you need four SQLVAR occurrences in your SQLDA because the result table contains a LOB type and a distinct type. Suppose that you prepare and describe this statement into FULSQLDA, which is large enough to hold four SQLVAR occurrences. FULSQLDA looks like the following figure.

![Figure 36. SQL descriptor area after describing a CLOB and distinct type](image)

The next steps are the same as for result tables without LOBs or distinct types:

1. Analyze each SQLVAR description to determine the maximum amount of space you need for the column value.
   - For a LOB type, retrieve the length from the SQLLONG field instead of the SQLLEN field.
2. Derive the address of some storage area of the required size.
   - For a LOB data type, you also need a 4-byte storage area for the length of the LOB data. You can allocate this 4-byte area at the beginning of the LOB data or in a different location.
3. Put this address in the SQLDATA field.
   - For a LOB data type, if you allocated a separate area to hold the length of the LOB data, put the address of the length field in SQLDATAL. If the length field is at beginning of the LOB data area, put 0 in SQLDATAL. When you use a file reference variable for a LOB column, the indicator variable indicates whether the data in the file is null, not whether the data to which SQLDATA points is null.
4. If the SQLTYPE field indicates that the value can be null, the program must also put the address of an indicator variable in the SQLIND field.

The following figure shows the contents of FULSQLDA after you enter pointers to the storage locations.

![Figure 37. SQL descriptor area after analyzing CLOB and distinct type descriptions and acquiring storage](image)

---

512 Db2 12 for z/OS: Application Programming and SQL Guide
The following figure shows the contents of FULSQLDA after you execute a FETCH statement.

![SQL descriptor area after executing FETCH on a table with CLOB and distinct type columns](image)

**Figure 38. SQL descriptor area after executing FETCH on a table with CLOB and distinct type columns**

### Setting an XML host variable in an SQLDA

Instead of specifying host variables to store XML values from a table, you can create an SQLDA to point to the data areas where Db2 puts the retrieved data. The SQLDA needs to describe the data type for each data area.

To set an XML host variable in an SQLDA:

1. Allocate an appropriate SQLDA.
2. Issue a DESCRIBE statement for the SQL statement whose result set you want to store. The DESCRIBE statement populates the SQLDA based on the column definitions. In the SQLDA, an SQLVAR entry is populated for each column in the result set. (Multiple SQLVAR entries are populated for LOB columns and columns with distinct types.) For columns of type XML the associated SQLVAR entry is populated as follows:

   ![Table 91. SQLVAR field values for XML columns](image)

<table>
<thead>
<tr>
<th>SQLVAR field</th>
<th>Value for an XML column</th>
</tr>
</thead>
<tbody>
<tr>
<td>sqltype</td>
<td>988 for a column that is not nullable or 989 for a nullable column</td>
</tr>
<tr>
<td>sqllen</td>
<td>0</td>
</tr>
<tr>
<td>sqldata</td>
<td>0</td>
</tr>
<tr>
<td>sqlind</td>
<td>0</td>
</tr>
<tr>
<td>sqlname</td>
<td>The unqualified name or label of the column</td>
</tr>
</tbody>
</table>

3. Check the SQLTYPE field of each SQLVAR entry. If the SQLTYPE field is 988 or 989, the column in the result set is an XML column.

4. For each XML column, make the following changes to the associated SQLVAR entry:
a. Change the SQLTYPE field to indicate the data type of the host variable to receive the XML data. You can retrieve the XML data into a host variable of type XML AS BLOB, XML AS CLOB, or XML AS DBCLOB, or a compatible string data type.

If the target host variable type is XML AS BLOB, XML AS CLOB, or XML AS DBCLOB, set the SQLTYPE field to one of the following values:

404  XML AS BLOB
405  nullable XML AS BLOB
408  XML AS CLOB
409  nullable XML AS CLOB
412  XML AS DBCLOB
413  nullable XML AS DBCLOB

If the target host variable type is a string data type, set the SQLTYPE field to a valid string value.

Restriction: You cannot use the XML type (988/989) as a target host variable type.

b. If the target host variable type is XML AS BLOB, XML AS CLOB, or XML AS DBCLOB, change the first two bytes in the SQLNAME field to X'0000' and the fifth and sixth bytes to X'0100'. These bytes indicate that the value to be received is an XML value.

5. Populate the extended SQLVAR fields for each XML column as you would for a LOB column, as indicated in the following table.

<table>
<thead>
<tr>
<th>SQLVAR field</th>
<th>Value for an XML host variable</th>
</tr>
</thead>
<tbody>
<tr>
<td>len.sqllonglen SQLLONGL SQLLONGLEN</td>
<td>length attribute for the XML host variable</td>
</tr>
<tr>
<td>*</td>
<td>Reserved</td>
</tr>
<tr>
<td>sqldatalen</td>
<td>pointer to the length of the XML host variable</td>
</tr>
<tr>
<td>SQLDATAL SQLDATALEN</td>
<td></td>
</tr>
<tr>
<td>sqldatatype_name SQLTNAME SQLDATATYPENAME</td>
<td>not used</td>
</tr>
</tbody>
</table>

You can now use the SQLDA to retrieve the XML data into a host variable of type XML AS BLOB, XML AS CLOB, or XML AS DBCLOB, or a compatible string data type.

**Executing a varying-list SELECT statement dynamically**

You can easily retrieve rows of the result table using a varying-list SELECT statement. The statements differ only a little from those for the fixed-list example.

1. Open the cursor. If the SELECT statement contains no parameter marker, this step is simple enough. For example:

```sql
EXEC SQL OPEN C1;
```
2. Fetch rows from the result table. This statement differs from the corresponding one for the case of a fixed-list select. Write:

```
EXEC SQL
FETCH C1 USING DESCRIPTOR :FULSQLDA;
```

The key feature of this statement is the clause USING DESCRIPTOR :FULSQLDA. That clause names an SQL descriptor area in which the occurrences of SQLVAR point to other areas. Those other areas receive the values that FETCH returns. It is possible to use that clause only because you previously set up FULSQLDA to look like Figure 32 on page 508.

Figure 34 on page 509 shows the result of the FETCH. The data areas identified in the SQLVAR fields receive the values from a single row of the result table.

Successive executions of the same FETCH statement put values from successive rows of the result table into these same areas.

3. Close the cursor. This step is the same as for the fixed-list case. When no more rows need to be processed, execute the following statement:

```
EXEC SQL CLOSE C1;
```

When COMMIT ends the unit of work containing OPEN, the statement in STMT reverts to the unprepared state. Unless you defined the cursor using the WITH HOLD option, you must prepare the statement again before you can reopen the cursor.

**Executing arbitrary statements with parameter markers**

Consider, as an example, a program that executes dynamic SQL statements of several kinds, including varying-list SELECT statements, any of which might contain a variable number of parameter markers. This program might present your users with lists of choices: choices of operation (update, select, delete); choices of table names; choices of columns to select or update. The program also enables the users to enter lists of employee numbers to apply to the chosen operation. From this, the program constructs SQL statements of several forms, one of which looks like this:

```
SELECT ..... FROM DSN8C10.EMP
WHERE EMPNO IN (?,?,?,...?);
```

The program then executes these statements dynamically.

**When the number and types of parameters are known**

In the preceding example, you do not know in advance the number of parameter markers, and perhaps the kinds of parameter they represent. You can use techniques described previously if you know the number and types of parameters, as in the following examples:

- If the SQL statement is not SELECT, name a list of host variables in the EXECUTE statement:

<table>
<thead>
<tr>
<th>WRONG:</th>
<th>EXEC SQL EXECUTE STMT;</th>
</tr>
</thead>
<tbody>
<tr>
<td>RIGHT:</td>
<td>EXEC SQL EXECUTE STMT USING :VAR1, :VAR2, :VAR3;</td>
</tr>
</tbody>
</table>

- If the SQL statement is SELECT, name a list of host variables in the OPEN statement:

<table>
<thead>
<tr>
<th>WRONG:</th>
<th>EXEC SQL OPEN C1;</th>
</tr>
</thead>
<tbody>
<tr>
<td>RIGHT:</td>
<td>EXEC SQL OPEN C1 USING :VAR1, :VAR2, :VAR3;</td>
</tr>
</tbody>
</table>

In both cases, the number and types of host variables named must agree with the number of parameter markers in STMT and the types of parameter they represent. The first variable (VAR1 in the examples) must have the type expected for the first parameter marker in the statement, the second variable must have the type expected for the second marker, and so on. There must be at least as many variables as parameter markers.

**When the number and types of parameters are not known**

When you do not know the number and types of parameters, you can adapt the SQL descriptor area. Your program can include an unlimited number of SQLDAs, and you can use them for
different purposes. Suppose that an SQLDA, arbitrarily named DPARM, describes a set of parameters.

The structure of DPARM is the same as that of any other SQLDA. The number of occurrences of SQLVAR can vary, as in previous examples. In this case, every parameter marker must have one SQLVAR. Each occurrence of SQLVAR describes one host variable that replaces one parameter marker at run time. Db2 replaces the parameter markers when a non-SELECT statement executes or when a cursor is opened for a SELECT statement.

You must enter certain fields in DPARM before using EXECUTE or OPEN; you can ignore the other fields.

Field
Use when describing host variables for parameter markers

SQLDAAID
The seventh byte indicates whether more than one SQLVAR entry is used for each parameter marker. If this byte is not blank, at least one parameter marker represents a distinct type or LOB value, so the SQLDA has more than one set of SQLVAR entries.

You do not set this field for a REXX SQLDA.

SQLDABC
The length of the SQLDA, which is equal to SQLN * 44 + 16. You do not set this field for a REXX SQLDA.

SQLN
The number of occurrences of SQLVAR allocated for DPARM. You do not set this field for a REXX SQLDA.

SQLD
The number of occurrences of SQLVAR actually used. This number must not be less than the number of parameter markers. In each occurrence of SQLVAR, put information in the following fields: SQLTYPE, SQLLEN, SQLDATA, SQLIND.

SQLTYPE
The code for the type of variable, and whether it allows nulls.

SQLLEN
The length of the host variable.

SQLDATA
The address of the host variable.

For REXX, this field contains the value of the host variable.

SQLIND
The address of an indicator variable, if needed.

For REXX, this field contains a negative number if the value in SQLDATA is null.

SQLNAME
Ignore.

Using the SQLDA with EXECUTE or OPEN
To indicate that the SQLDA called DPARM describes the host variables substituted for the parameter markers at run time, use a USING DESCRIPTOR clause with EXECUTE or OPEN.

• For a non-SELECT statement, write:

```sql
EXEC SQL EXECUTE STMT USING DESCRIPTOR :DPARM;
```

• For a SELECT statement, write:

```sql
EXEC SQL OPEN C1 USING DESCRIPTOR :DPARM;
```
How bind options REOPT(ALWAYS), REOPT(AUTO) and REOPT(ONCE) affect dynamic SQL

When you specify the bind option REOPT(ALWAYS), Db2 reoptimizes the access path at run time for SQL statements that contain host variables, parameter markers, or special registers. The option REOPT(ALWAYS) has the following effects on dynamic SQL statements:

- When you specify the option REOPT(ALWAYS), Db2 automatically uses DEFER(PREPARE), which means that Db2 waits to prepare a statement until it encounters an OPEN or EXECUTE statement.

- When you execute a DESCRIBE statement and then an EXECUTE statement on a non-SELECT statement, Db2 prepares the statement twice: Once for the DESCRIBE statement and once for the EXECUTE statement. Db2 uses the values in the input variables only during the second PREPARE. These multiple PREPAREs can cause performance to degrade if your program contains many dynamic non-SELECT statements. To improve performance, consider putting the code that contains those statements in a separate package and then binding that package with the option REOPT(NONE).

- If you execute a DESCRIBE statement before you open a cursor for that statement, Db2 prepares the statement twice. If, however, you execute a DESCRIBE statement after you open the cursor, Db2 prepares the statement only once. To improve the performance of a program bound with the option REOPT(ALWAYS), execute the DESCRIBE statement after you open the cursor. To prevent an automatic DESCRIBE before a cursor is opened, do not use a PREPARE statement with the INTO clause.

- If you specify the bind option REOPT(AUTO), Db2 optimizes the access path for SQL statements at the first EXECUTE or OPEN. Each time a statement is executed, Db2 determines if a new access path is needed to improve the performance of the statement. If a new access path will improve the performance, Db2 generates one. The option REOPT(AUTO) has the following effects on dynamic SQL statements:

  - When you specify the bind option REOPT(AUTO), Db2 optimizes the access path for SQL statements at the first EXECUTE or OPEN. Each time a statement is executed, Db2 determines if a new access path is needed to improve the performance of the statement. If a new access path will improve the performance, Db2 generates one.

  - When you specify the option REOPT(ONCE), Db2 automatically uses DEFER(PREPARE), which means that Db2 waits to prepare a statement until it encounters an OPEN or EXECUTE statement.

  - When Db2 prepares a statement using REOPT(AUTO), it saves the access path in the dynamic statement cache. This access path is used each time the statement is run, until Db2 determines that a new access path is needed to improve the performance or the statement that is in the cache is invalidated (or removed from the cache) and needs to be rebound.

  - The DESCRIBE statement has the following effects on dynamic statements that are bound with REOPT(AUTO):

    - When you execute a DESCRIBE statement before an EXECUTE statement on a non-SELECT statement, Db2 prepares the statement an extra time if it is not already saved in the cache: Once for the DESCRIBE statement and once for the EXECUTE statement. Db2 uses the values of the input variables only during the second time the statement is prepared. It then saves the statement in the cache. If you execute a DESCRIBE statement before an EXECUTE statement on a non-SELECT statement that has already been saved in the cache, Db2 will always prepare the non-SELECT statement for the DESCRIBE statement, and will prepare the statement again on EXECUTE only if Db2 determines that a new access path different from the one already saved in the cache can improve the performance.

    - If you execute DESCRIBE on a statement before you open a cursor for that statement, Db2 always prepares the statement on DESCRIBE. However, Db2 will not prepare the statement again on OPEN if the statement has already been saved in the cache and Db2 does not think that a new access path is needed at OPEN time. If you execute DESCRIBE on a statement after you open a
cursor for that statement, Db2 prepared the statement only once if it is not already saved in the cache. If the statement is already saved in the cache and you execute DESCRIBE after you open a cursor for that statement, Db2 does not prepare the statement, it used the statement that is saved in the cache.

- If you use predictive governing for applications that are bound with REOPT(AUTO), Db2 does not return a warning SQLCODE when dynamic SQL statements exceed the predictive governing warning threshold. Db2 does return an error SQLCODE when dynamic SQL statements exceed the predictive governing error threshold. Db2 returns the error SQLCODE for an EXECUTE or OPEN statement.

When you specify the bind option REOPT(ONCE), Db2 optimizes the access path only once, at the first EXECUTE or OPEN, for SQL statements that contain host variables, parameter markers, or special registers. The option REOPT(ONCE) has the following effects on dynamic SQL statements:

- When you specify the option REOPT(ONCE), Db2 automatically uses DEFER(PREPARE), which means that Db2 waits to prepare a statement until it encounters an OPEN or EXECUTE statement.

- When Db2 prepares a statement using REOPT(ONCE), it saves the access path in the dynamic statement cache. This access path is used each time the statement is run, until the statement that is in the cache is invalidated (or removed from the cache) and needs to be rebound.

- The DESCRIBE statement has the following effects on dynamic statements that are bound with REOPT(ONCE):

  - When you execute a DESCRIBE statement before an EXECUTE statement on a non-SELECT statement, Db2 prepares the statement twice if it is not already saved in the cache: Once for the DESCRIBE statement and once for the EXECUTE statement. Db2 uses the values of the input variables only during the second time the statement is prepared. It then saves the statement in the cache. If you execute a DESCRIBE statement before an EXECUTE statement on a non-SELECT statement that has already been saved in the cache, Db2 prepares the non-SELECT statement only for the DESCRIBE statement.

  - If you execute DESCRIBE on a statement before you open a cursor for that statement, Db2 always prepares the statement on DESCRIBE. However, Db2 will not prepare the statement again on OPEN if the statement has already been saved in the cache. If you execute DESCRIBE on a statement after you open a cursor for that statement, Db2 prepared the statement only once if it is not already saved in the cache. If the statement is already saved in the cache and you execute DESCRIBE after you open a cursor for that statement, Db2 does not prepare the statement, it used the statement that is saved in the cache.

To improve the performance of a program that is bound with REOPT(ONCE), execute the DESCRIBE statement after you open a cursor. To prevent an automatic DESCRIBE before a cursor is opened, do not use a PREPARE statement with the INTO clause.

- If you use predictive governing for applications that are bound with REOPT(ONCE), Db2 does not return a warning SQLCODE when dynamic SQL statements exceed the predictive governing warning threshold. Db2 does return an error SQLCODE when dynamic SQL statements exceed the predictive governing error threshold. Db2 returns the error SQLCODE for an EXECUTE or OPEN statement.

Related concepts

Assembler applications that issue SQL statements
You can code SQL statements in assembler programs wherever you can use executable statements.

C and C++ applications that issue SQL statements
You can code SQL statements in a C or C++ program wherever you can use executable statements.

COBOL applications that issue SQL statements
You can code SQL statements in certain COBOL program sections.

Fortran applications that issue SQL statements
You can code SQL statements in a Fortran program wherever you can place executable statements. If the SQL statement is within an IF statement, the precompiler generates any necessary THEN and END IF statements.

PL/I applications that issue SQL statements
You can code SQL statements in a PL/I program wherever you can use executable statements.

REXX applications that issue SQL statements
You can code SQL statements in a REXX programs wherever you can use REXX commands.

**Related reference**
DESCRIBE OUTPUT (Db2 SQL)
SQL descriptor area (SQLDA) (Db2 SQL)
SQLTYPE and SQLLEN (Db2 SQL)
The SQLDA Header (Db2 SQL)

Dynamically executing an SQL statement by using EXECUTE IMMEDIATE

In certain situations, you might want your program to prepare and dynamically execute a statement immediately after reading it.

**About this task**

Suppose that you design a program to read SQL DELETE statements, similar to these, from a terminal:

```sql
DELETE FROM DSN8C10.EMP WHERE EMPNO = '000190'
DELETE FROM DSN8C10.EMP WHERE EMPNO = '000220'
```

After reading a statement, the program is to run it immediately.

Recall that you must prepare (precompile and bind) static SQL statements before you can use them. You cannot prepare dynamic SQL statements in advance. The SQL statement EXECUTE IMMEDIATE causes an SQL statement to prepare and execute, dynamically, at run time.

Before you prepare and execute an SQL statement, you can read it into a host variable. If the maximum length of the SQL statement is 32 KB, declare the host variable as a character or graphic host variable according to the following rules for the host languages:

- In assembler, PL/I, COBOL and C, you must declare a string host variable as a varying-length string.
- In Fortran, it must be a fixed-length string variable.

If the length is greater than 32 KB, you must declare the host variable as a CLOB or DBCLOB, and the maximum is 2 MB.

**Examples**

**Example: Using a varying-length character host variable**

This excerpt is from a C program that reads a DELETE statement into the host variable `dstring` and executes the statement:

```c
EXEC SQL BEGIN DECLARE SECTION;
...
  struct VARCHAR {
    short len;
    char s[48];
  } dstring;
EXEC SQL END DECLARE SECTION;
/* Read a DELETE statement into the host variable dstring. */
gets(dstring);
EXEC SQL EXECUTE IMMEDIATE :dstring;
...
```

EXECUTE IMMEDIATE causes the DELETE statement to be prepared and executed immediately.

**Declaring a CLOB or DBCLOB host variable**

You declare CLOB and DBCLOB host variables according to certain rules.

The precompiler generates a structure that contains two elements, a 4-byte length field and a data field of the specified length. The names of these fields vary depending on the host language:
• In PL/I, assembler, and Fortran, the names are `variable_LENGTH` and `variable_DATA`.
• In COBOL, the names are `variable–LENGTH` and `variable–DATA`.
• In C, the names are `variable.LENGTH` and `variable.DATA`.

**Example: Using a CLOB host variable**
This excerpt is from a C program that copies an UPDATE statement into the host variable `string1` and executes the statement:

```c
EXEC SQL BEGIN DECLARE SECTION;
...
SQL TYPE IS CLOB(4k) string1;
EXEC SQL END DECLARE SECTION;
... */
strcpy(string1.data, "UPDATE DSN8610.EMP SET SALARY = SALARY * 1.1");
string1.length = 44;
EXEC SQL EXECUTE IMMEDIATE :string1;
... 
```

EXECUTE IMMEDIATE causes the UPDATE statement to be prepared and executed immediately.

**Related concepts**
LOB host variable, LOB locator, and LOB file reference variable declarations
When you write applications to manipulate LOB data, you need to declare host variables to hold the LOB data or LOB locator. Alternatively, you need to declare LOB file reference variables to point to the LOB data.

Assembler applications that issue SQL statements
You can code SQL statements in assembler programs wherever you can use executable statements.

C and C++ applications that issue SQL statements
You can code SQL statements in a C or C++ program wherever you can use executable statements.

COBOL applications that issue SQL statements
You can code SQL statements in certain COBOL program sections.

Fortran applications that issue SQL statements
You can code SQL statements in a Fortran program wherever you can place executable statements. If the SQL statement is within an IF statement, the precompiler generates any necessary THEN and END IF statements.

PL/I applications that issue SQL statements
You can code SQL statements in a PL/I program wherever you can use executable statements.

REXX applications that issue SQL statements
You can code SQL statements in a REXX program wherever you can use REXX commands.

**Dynamically executing an SQL statement by using PREPARE and EXECUTE**
As an alternative to executing an SQL statement immediately after it is read, you can prepare and execute the SQL statement in two steps. This two-step method is useful when you need to execute an SQL statement multiple times with different values.

**About this task**
Suppose that you want to execute DELETE statements repeatedly using a list of employee numbers. Consider how you would do it if you could write the DELETE statement as a static SQL statement:

```c
< Read a value for EMP from the list. >
DO UNTIL (EMP = 0);
EXEC SQL
   DELETE FROM DSN8C10.EMP WHERE EMPNO = :EMP ;
< Read a value for EMP from the list. >
END;
```

The loop repeats until it reads an EMP value of 0.
If you know in advance that you will use only the DELETE statement and only the table DSN8C10.EMP, you can use the more efficient static SQL. Suppose further that several different tables have rows that are identified by employee numbers, and that users enter a table name as well as a list of employee numbers to delete. Although variables can represent the employee numbers, they cannot represent the table name, so you must construct and execute the entire statement dynamically.

**Procedure**

To construct and execute statements dynamically your program must now do these things differently:

- **Use parameter markers instead of host variables.**
  Dynamic SQL statements cannot use host variables. Therefore, you cannot dynamically execute an SQL statement that contains host variables. Instead, substitute a parameter marker, indicated by a question mark (?), for each host variable in the statement.

  You can indicate to Db2 that a parameter marker represents a host variable of a certain data type by specifying the parameter marker as the argument of a CAST specification. When the statement executes, Db2 converts the host variable to the data type in the CAST specification. A parameter marker that you include in a CAST specification is called a *typed* parameter marker. A parameter marker without a CAST specification is called an *untyped* parameter marker.

  **Recommendation:** Because Db2 can evaluate an SQL statement with typed parameter markers more efficiently than a statement with untyped parameter markers, use typed parameter markers whenever possible. Under certain circumstances you must use typed parameter markers.

  For example, suppose that you want to prepare this statement:

  ```sql
  DELETE FROM DSN8C10.EMP WHERE EMPNO = :EMP;
  ```

  You need to prepare a string like this:

  ```sql
  DELETE FROM DSN8C10.EMP WHERE EMPNO = CAST(? AS CHAR(6))
  ```

  You associate host variable :EMP with the parameter marker when you execute the prepared statement. Suppose that S1 is the prepared statement. Then the EXECUTE statement looks like this:

  ```sql
  EXECUTE S1 USING :EMP;
  ```

- **Use the PREPARE statement.**
  Before you prepare an SQL statement, you can assign it to a host variable. If the length of the statement is greater than 32 KB, you must declare the host variable as a CLOB or DBCLOB.

  You can think of PREPARE and EXECUTE as an EXECUTE IMMEDIATE done in two steps. The first step, PREPARE, turns a character string into an SQL statement, and then assigns it a name of your choosing.

  For example, assume that the character host variable :DSTRING has the value “DELETE FROM DSN8C10.EMP WHERE EMPNO = ?”. To prepare an SQL statement from that string and assign it the name S1, write:

  ```sql
  EXEC SQL PREPARE S1 FROM :DSTRING;
  ```

  The prepared statement still contains a parameter marker, for which you must supply a value when the statement executes. After the statement is prepared, the table name is fixed, but the parameter marker enables you to execute the same statement many times with different values of the employee number.

- **Use EXECUTE instead of EXECUTE IMMEDIATE.**
  The EXECUTE statement executes a prepared SQL statement by naming a list of one or more host variables, one or more host-variable arrays, or a host structure. This list supplies values for all of the parameter markers.

  After you prepare a statement, you can execute it many times within the same unit of work. In most cases, COMMIT or ROLLBACK destroys statements prepared in a unit of work. Then, you must
prepare them again before you can execute them again. However, if you declare a cursor for a
dynamic statement and use the option WITH HOLD, a commit operation does not destroy the prepared
statement if the cursor is still open. You can execute the statement in the next unit of work without
preparing it again.

For example, to execute the prepared statement S1 just once, using a parameter value contained in
the host variable :EMP, write:

```
EXEC SQL EXECUTE S1 USING :EMP;
```

**Examples**

**Preparing and executing the example DELETE statement**

```
< Read a value for EMP from the list. >
DO UNTIL (EMP = 0);
   EXEC SQL
      DELETE FROM DSN8C10.EMP WHERE EMPNO = :EMP ;
   < Read a value for EMP from the list. >
END;
```

You can now write an equivalent example for a dynamic SQL statement:

```
< Read a statement containing parameter markers into DSTRING.>
EXEC SQL PREPARE S1 FROM :DSTRING;
< Read a value for EMP from the list. >
DO UNTIL (EMPNO = 0);
   EXEC SQL EXECUTE S1 USING :EMP;
   < Read a value for EMP from the list. >
END;
```

The PREPARE statement prepares the SQL statement and calls it S1. The EXECUTE statement
executes S1 repeatedly, using different values for EMP.

**Using more than one parameter marker**

The prepared statement (S1 in the example) can contain more than one parameter marker. If it does,
the USING clause of EXECUTE specifies a list of variables or a host structure. The variables must
contain values that match the number and data types of parameters in S1 in the proper order. You
must know the number and types of parameters in advance and declare the variables in your program,
or you can use an SQLDA (SQL descriptor area).

**Related concepts**

Assembler applications that issue SQL statements
You can code SQL statements in assembler programs wherever you can use executable statements.

C and C++ applications that issue SQL statements
You can code SQL statements in a C or C++ program wherever you can use executable statements.

COBOL applications that issue SQL statements
You can code SQL statements in certain COBOL program sections.

Fortran applications that issue SQL statements
You can code SQL statements in a Fortran program wherever you can place executable statements. If
the SQL statement is within an IF statement, the precompiler generates any necessary THEN and END IF
statements.

PL/I applications that issue SQL statements
You can code SQL statements in a PL/I program wherever you can use executable statements.

REXX applications that issue SQL statements
You can code SQL statements in a REXX program wherever you can use REXX commands.

**Related tasks**

Dynamically executing an SQL statement by using EXECUTE IMMEDIATE
In certain situations, you might want your program to prepare and dynamically execute a statement immediately after reading it.

Related reference
PREPARE (Db2 SQL)

Dynamically executing a data change statement

Dynamically executing data change statements with host-variable arrays is useful if you want to enter rows of data into different tables. It is also useful if you want to enter a different number of rows. The process is similar for both INSERT and MERGE statements.

About this task

For example, suppose that you want to repeatedly execute a multiple-row INSERT statement with a list of activity IDs, activity keywords, and activity descriptions that are provided by the user. You can use the following static SQL INSERT statement to insert multiple rows of data into the activity table:

```
EXEC SQL
  INSERT INTO DSN8C10.ACT
  VALUES (:hva_actno, :hva_actkwd, :hva_actdesc)
FOR :num_rows ROWS;
```

However, if you want to enter the rows of data into different tables or enter different numbers of rows, you can construct the INSERT statement dynamically.

Procedure

To execute a data change statement dynamically, use one of the following methods:

- Use host-variable arrays that contain the data to be inserted, by completing the following actions in your program:
  a) Assign the appropriate INSERT or MERGE statement to a host variable. If needed, use the CAST specification to explicitly assign types to parameter markers that represent host-variable arrays. For the activity table, the following string contains an INSERT statement that is to be prepared:

  ```
  INSERT INTO DSN8C10.ACT
  VALUES (CAST(? AS SMALLINT), CAST(? AS CHAR(6)), CAST(? AS VARCHAR(20)))
  FOR :num_rows ROWS;
  ```

  b) Assign any attributes for the SQL statement to a host variable.
  c) Include a PREPARE statement for the SQL statement.
  d) Include an EXECUTE statement with the FOR n ROWS clause.

    Each host variable in the USING clause of the EXECUTE statement represents an array of values for the corresponding column of the target of the SQL statement. You can vary the number of rows without needing to prepare the SQL statement again.

    For example, the following code prepares and executes an INSERT statement:

    ```
    /* Copy the INSERT string into the host variable sqlstmt */
    strcpy(sqlstmt, "INSERT INTO DSN8C10.ACT VALUES (CAST(? AS SMALLINT),
      CAST(? AS CHAR(6)), CAST(? AS VARCHAR(20))) ");
    
    /* Copy the INSERT attributes into the host variable attrvar */
    strcpy(attrvar, "FOR MULTIPLE ROWS ");
    
    /* Prepare and execute my_insert using the host-variable arrays */
    EXEC SQL PREPARE my_insert ATTRIBUTES :attrvar FROM :sqlstmt;
    EXEC SQL EXECUTE my_insert USING :hva1, :hva2, :hva3 FOR :num_rows ROWS;
    ```

- Use descriptor to describe the host-variable arrays that contain the data, by completing the following actions in your program:
  a) Set the following fields in the SQLDA structure to specify data types and other information about the host-variable arrays that contain the values to insert in your INSERT statement.
The SQLDA structure has the following fields:

- SQLDABC indicates the number of bytes of storage that are allocated for the SQLDA. The storage includes a 16-byte header and 44 bytes for each SQLVAR field. The value is SQLN x 44 + 16, or 192 for this example.
- SQLN is the number of SQLVAR occurrences, plus one for use by Db2 for the host variable that contains the number n in the FOR n ROWS clause.
- SQLD is the number of variables in the SQLDA that are used by Db2 when processing the INSERT statement.
- An SQLVAR occurrence specifies the attributes of an element of a host-variable array that corresponds to a value provided for a target column of the INSERT. Within each SQLVAR:
  - SQLTYPE indicates the data type of the elements of the host-variable array.
  - SQLLEN indicates the length of a single element of the host-variable array.
  - SQLDATA points to the corresponding host-variable array. Assume that your program allocates the dynamic variable arrays hva1, hva2, and hva3.
  - SQLNAME has two parts: the LENGTH and the DATA. The LENGTH is 8. The first two bytes of the DATA field is X'0000'. Bytes 5 and 6 of the DATA field are a flag indicating whether the variable is an array or a FOR n ROWS value. Bytes 7 and 8 are a two-byte binary integer representation of the dimension of the array.

b) Assign the appropriate INSERT or MERGE statement to a host variable. For example, the following string contains an INSERT statement that is to be prepared:
c) Assign any attributes for the SQL statement to a host variable.

d) Include a PREPARE statement for the SQL statement.

e) Include an EXECUTE statement with the FOR n ROWS clause. The host variable in the USING clause of the EXECUTE statement names the SQLDA that describes the parameter markers in the INSERT statement.

For example, the following code prepares and executes an INSERT statement:

```c
/* Copy the INSERT string into the host variable sqlstmt */
strcpy(sqlstmt, "INSERT INTO DSN8C10.ACT VALUES (?, ?, ?)";

/* Copy the INSERT attributes into the host variable attrvar */
strcpy(attrvar, "FOR MULTIPLE ROWS");

/* Prepare and execute my_insert using the descriptor */
EXEC SQL PREPARE my_insert ATTRIBUTES :attrvar FROM :sqlstmt;
EXEC SQL EXECUTE my_insert USING DESCRIPTOR :*sqldaptr FOR :num_rows ROWS;
```

**Related concepts**

Assembler applications that issue SQL statements
You can code SQL statements in assembler programs wherever you can use executable statements.

C and C++ applications that issue SQL statements
You can code SQL statements in a C or C++ program wherever you can use executable statements.

COBOL applications that issue SQL statements
You can code SQL statements in certain COBOL program sections.

Fortran applications that issue SQL statements
You can code SQL statements in a Fortran program wherever you can place executable statements. If the SQL statement is within an IF statement, the precompiler generates any necessary THEN and END IF statements.

PL/I applications that issue SQL statements
You can code SQL statements in a PL/I program wherever you can use executable statements.

**Related tasks**

Including dynamic SQL for varying-list SELECT statements in your program
A varying-list SELECT statement returns rows that contain an unknown number of values of unknown type. When you use this type of statement, you do not know in advance exactly what kinds of host variables you need to declare for storing the results.

**Related reference**

SQLTYPE and SQLLEN (Db2 SQL)

---

**Dynamically executing a statement with parameter markers by using the SQLDA**

Your program can get data type information about parameter markers by asking Db2 to set the fields in the SQLDA.

**Before you begin**

Before you dynamically execute a statement with parameter markers, allocate an SQLDA with enough instances of SQLVAR to represent all parameter markers in the SQL statement.

**Procedure**

To dynamically execute a statement with parameter markers by using the SQLDA:

1. Include in your program a DESCRIBE INPUT statement that specifies the prepared SQL statement and the name of an appropriate SQLDA.
Db2 puts the requested parameter marker information in the SQLDA.

2. Code the application in the same way as any other application in which you execute a prepared statement by using an SQLDA. First, obtain the addresses of the input host variables and their indicator variables and insert those addresses into the SQLDATA and SQLIND fields. Then, execute the prepared SQL statement.

**Example**

Suppose that you want to execute the following statement dynamically:

```
DELETE FROM DSN8C10.EMP WHERE EMPNO = ?
```

You can use the following code to set up an SQLDA, obtain parameter information by using the DESCRIBE INPUT statement, and execute the statement:

```sql
SQLDAID='SQLDA';          /* Get pointer to SQLDA         */
SQLDABC=LENGTH(INSQLDA);         /* Fill in SQLDA eye-catcher */
SQLN=1;                          /* Fill in number of SQLVARs */
SQLD=0;                          /* Initialize # of SQLVARs used */
DO IX=1 TO SQLN;                 /* Initialize the SQLVAR */
   SQLTYPE(IX)=0;                  /* Get input data address */
   SQLLEN(IX)=0;                   /* Get indicator address */
END;
SQLSTMT='DELETE FROM DSN8C10.EMP WHERE EMPNO = ?';
EXEC SQL PREPARE SQLOBJ FROM SQLSTMT;
EXEC SQL DESCRIBE INPUT SQLOBJ INTO :INSQLDA;
SQLDATA(1)=ADDR(HVEMP);          /* Get input data address */
SQLIND(1)=ADDR(HVEMPIND);        /* Get indicator address */
EXEC SQL EXECUTE SQLOBJ USING DESCRIPTOR :INSQLDA;
```

**Related concepts**

Assembler applications that issue SQL statements
You can code SQL statements in assembler programs wherever you can use executable statements.

C and C++ applications that issue SQL statements
You can code SQL statements in a C or C++ program wherever you can use executable statements.

COBOL applications that issue SQL statements
You can code SQL statements in certain COBOL program sections.

Fortran applications that issue SQL statements
You can code SQL statements in a Fortran program wherever you can place executable statements. If the SQL statement is within an IF statement, the precompiler generates any necessary THEN and END IF statements.

PL/I applications that issue SQL statements
You can code SQL statements in a PL/I program wherever you can use executable statements.

REXX applications that issue SQL statements
You can code SQL statements in a REXX program wherever you can use REXX commands.

**Related tasks**

Defining SQL descriptor areas (SQLDA)
If your program includes certain SQL statements, you must define at least one SQL descriptor area (SQLDA). Depending on the context in which it is used, the SQLDA stores information about prepared SQL statements or host variables. This information can then be read by either the application program or Db2.

**Related reference**

DESCRIBE INPUT (Db2 SQL)
Checking the execution of SQL statements

After executing an SQL statement, your program should check for any errors before you commit the data and handle the errors that they represent.

About this task

You can check the execution of SQL statements in one of the following ways:

- By displaying specific fields in the SQLCA.
- By testing SQLCODE or SQLSTATE for specific values.
- By using the WHENEVER statement in your application program.
- By testing indicator variables to detect numeric errors.
- By using the GET DIAGNOSTICS statement in your application program to return all the condition information that results from the execution of an SQL statement.
- By calling DSNTIAR to display the contents of the SQLCA.

Related concepts

Arithmetic and conversion errors
You can track arithmetic and conversion errors by using indicator variables. An indicator variable contains a small integer value that indicates some information about the associated host variable.

Related tasks

Defining the SQL communications area, SQLSTATE, and SQLCODE in assembler
Assembler programs that contain SQL statements can include an SQL communications area (SQLCA) to check whether an SQL statement executed successfully. Alternatively, these programs can declare individual SQLCODE and SQLSTATE host variables.

Defining the SQL communications area, SQLSTATE, and SQLCODE in C and C++
C and C++ programs that contain SQL statements can include an SQL communications area (SQLCA) to check whether an SQL statement executed successfully. Alternatively, these programs can declare individual SQLCODE and SQLSTATE host variables.

Defining the SQL communications area, SQLSTATE, and SQLCODE in COBOL
COBOL programs that contain SQL statements can include an SQL communications area (SQLCA) to check whether an SQL statement executed successfully. Alternatively, these programs can declare individual SQLCODE and SQLSTATE host variables.

Defining the SQL communications area, SQLSTATE, and SQLCODE in Fortran
Fortran programs that contain SQL statements can include an SQL communications area (SQLCA) to check whether an SQL statement executed successfully. Alternatively, these programs can declare individual SQLCODE and SQLSTATE host variables.

Defining the SQL communications area, SQLSTATE, and SQLCODE in PL/I
PL/I programs that contain SQL statements can include an SQL communications area (SQLCA) to check whether an SQL statement executed successfully. Alternatively, these programs can declare individual SQLCODE and SQLSTATE host variables.

Defining the SQL communications area, SQLSTATE, and SQLCODE in REXX
When Db2 prepares a REXX program that contains SQL statements, Db2 automatically includes an SQLCA in the program.

Displaying SQLCA fields by calling DSNTIAR
If you use the SQLCA to check whether an SQL statement executed successfully, your program needs to read the data in the appropriate SQLCA fields. One easy way to read these fields is to use the assembler subroutine DSNTIAR.

### Checking the execution of SQL statements by using the SQLCA

One way to check whether an SQL statement executed successfully is to use the SQL communication area (SQLCA). This area is set apart for communication with Db2.

### About this task

If you use the SQLCA, include the necessary instructions to display information that is contained in the SQLCA in your application program. Alternatively, you can use the GET DIAGNOSTICS statement, which is an SQL standard, to diagnose problems.

- When Db2 processes an SQL statement, it places return codes that indicate the success or failure of the statement execution in SQLCODE and SQLSTATE.
- When Db2 processes a FETCH statement, and the FETCH is successful, the contents of SQLERRD(3) in the SQLCA is set to the number of returned rows.
- When Db2 processes a multiple-row FETCH statement, the contents of SQLCODE is set to +100 if the last row in the table has been returned with the set of rows.
- When Db2 processes an UPDATE, INSERT, or DELETE statement, and the statement execution is successful, the contents of SQLERRD(3) in the SQLCA is set to the number of rows that are updated, inserted, or deleted.
- When Db2 processes a TRUNCATE statement and the statement execution is successful, SQLERRD(3) in the SQLCA is set to -1. The number of rows that are deleted is not returned.
- If SQLWARN0 contains W, Db2 has set at least one of the SQL warning flags (SQLWARN1 through SQLWARNA):
  - SQLWARN1 contains N for non-scrollable cursors and S for scrollable cursors after an OPEN CURSOR or ALLOCATE CURSOR statement.
  - SQLWARN4 contains I for insensitive scrollable cursors, S for sensitive static scrollable cursors, and D for sensitive dynamic scrollable cursors, after an OPEN CURSOR or ALLOCATE CURSOR statement, or blank if the cursor is not scrollable.
  - SQLWARN5 contains a character value of 1 (read only), 2 (read and delete), or 4 (read, delete, and update) to indicate the operation that is allowed on the result table of the cursor.

### Related tasks

**Accessing data by using a rowset-positioned cursor**

A rowset-positioned cursor is a cursor that can return one or more rows for a single fetch operation. The cursor is positioned on the set of rows that are to be fetched.

**Checking the execution of SQL statements by using SQLCODE and SQLSTATE**

Whenever an SQL statement executes, the SQLCODE and SQLSTATE fields of the SQLCA receive a return code.

**Defining the SQL communications area, SQLSTATE, and SQLCODE in assembler**

Assembler programs that contain SQL statements can include an SQL communications area (SQLCA) to check whether an SQL statement executed successfully. Alternatively, these programs can declare individual SQLCODE and SQLSTATE host variables.

**Defining the SQL communications area, SQLSTATE, and SQLCODE in C and C++**

C and C++ programs that contain SQL statements can include an SQL communications area (SQLCA) to check whether an SQL statement executed successfully. Alternatively, these programs can declare individual SQLCODE and SQLSTATE host variables.

**Defining the SQL communications area, SQLSTATE, and SQLCODE in COBOL**
COBOL programs that contain SQL statements can include an SQL communications area (SQLCA) to check whether an SQL statement executed successfully. Alternatively, these programs can declare individual SQLCODE and SQLSTATE host variables.

Defining the SQL communications area, SQLSTATE, and SQLCODE in Fortran

Fortran programs that contain SQL statements can include an SQL communications area (SQLCA) to check whether an SQL statement executed successfully. Alternatively, these programs can declare individual SQLCODE and SQLSTATE host variables.

Defining the SQL communications area, SQLSTATE, and SQLCODE in PL/I

PL/I programs that contain SQL statements can include an SQL communications area (SQLCA) to check whether an SQL statement executed successfully. Alternatively, these programs can declare individual SQLCODE and SQLSTATE host variables.

Defining the SQL communications area, SQLSTATE, and SQLCODE in REXX

When Db2 prepares a REXX program that contains SQL statements, Db2 automatically includes an SQLCA in the program.

Related reference

Description of SQLCA fields (Db2 SQL)

Displaying SQLCA fields by calling DSNTIAR

If you use the SQLCA to check whether an SQL statement executed successfully, your program needs to read the data in the appropriate SQLCA fields. One easy way to read these fields is to use the assembler subroutine DSNTIAR.

About this task

You should check for errors codes before you commit data, and handle the errors that they represent. The assembler subroutine DSNTIAR helps you to obtain a formatted form of the SQLCA and a text message based on the SQLCODE field of the SQLCA. You can retrieve this same message text by using the MESSAGE_TEXT condition item field of the GET DIAGNOSTICS statement. Programs that require long token message support should code the GET DIAGNOSTICS statement instead of DSNTIAR.

DSNTIAR takes data from the SQLCA, formats it into a message, and places the result in a message output area that you provide in your application program. Each time you use DSNTIAR, it overwrites any previous messages in the message output area. You should move or print the messages before using DSNTIAR again, and before the contents of the SQLCA change, to get an accurate view of the SQLCA.

DSNTIAR expects the SQLCA to be in a certain format. If your application modifies the SQLCA format before you call DSNTIAR, the results are unpredictable.

DSNTIAR

The assembler subroutine DSNTIAR helps you to obtain a formatted form of the SQLCA and a text message that is based on the SQLCODE field of the SQLCA.

DSNTIAR can run either above or below the 16-MB line of virtual storage. The DSNTIAR object module that comes with Db2 has the attributes AMODE(31) and RMODE(ANY). At installation time, DSNTIAR links as AMODE(31) and RMODE(ANY). DSNTIAR runs in 31-bit mode if any of the following conditions is true:

- DSNTIAR is linked with other modules that also have the attributes AMODE(31) and RMODE(ANY).
- DSNTIAR is linked into an application that specifies the attributes AMODE(31) and RMODE(ANY) in its link-edit JCL.
- An application loads DSNTIAR.

When loading DSNTIAR from another program, be careful how you branch to DSNTIAR. For example, if the calling program is in 24-bit addressing mode and DSNTIAR is loaded above the 16-MB line, you cannot use the assembler BALR instruction or CALL macro to call DSNTIAR, because they assume that DSNTIAR is in 24-bit mode. Instead, you must use an instruction that is capable of branching into 31-bit mode, such as BASSM.
You can dynamically link (load) and call DSNTIAR directly from a language that does not handle 31-bit addressing. To do this, link a second version of DSNTIAR with the attributes AMODE(24) and RMODE(24) into another load module library. Alternatively, you can write an intermediate assembler language program that calls DSNTIAR in 31-bit mode and then call that intermediate program in 24-bit mode from your application.

For more information on the allowed and default AMODE and RMODE settings for a particular language, see the application programming guide for that language. For details on how the attributes AMODE and RMODE of an application are determined, see the linkage editor and loader user's guide for the language in which you have written the application.

**Defining a message output area**

If a program calls DSNTIAR, the program must allocate enough storage in the message output area to hold all of the message text that DSNTIAR returns.

**About this task**

You will probably need no more than 10 lines, 80-bytes each, for your message output area. An application program can have only one message output area.

You must define the message output area in VARCHAR format. In this varying character format, a 2-byte length field precedes the data. The length field indicates to DSNTIAR how many total bytes are in the output message area; the minimum length of the output area is 240-bytes.

The following figure shows the format of the message output area, where `length` is the 2-byte total length field, and the length of each line matches the logical record length (`lrecl`) you specify to DSNTIAR.

![Figure 39. Format of the message output area](image)

When you call DSNTIAR, you must name an SQLCA and an output message area in the DSNTIAR parameters. You must also provide the logical record length (`lrecl`) as a value between 72 and 240 bytes. DSNTIAR assumes the message area contains fixed-length records of length `lrecl`.

DSNTIAR places up to 10 lines in the message area. If the text of a message is longer than the record length you specify on DSNTIAR, the output message splits into several records, on word boundaries if possible. The split records are indented. All records begin with a blank character for carriage control. If you have more lines than the message output area can contain, DSNTIAR issues a return code of 4. A completely blank record marks the end of the message output area.

**Possible return codes from DSNTIAR**

The assembler subroutine DSNTIAR helps your program read the information in the SQLCA. The subroutine also returns its own return code.

<table>
<thead>
<tr>
<th>Code</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>530</td>
<td>Db2 12 for z/OS: Application Programming and SQL Guide</td>
</tr>
</tbody>
</table>
Successful execution.

More data available than could fit into the provided message area.

Logical record length not between 72 and 240, inclusive.

Message area not large enough. The message length was 240 or greater.

Error in TSO message routine.

Module DSNTIA1 could not be loaded.

SQLCA data error.

A scenario for using DSNTIAR

You can use the assembler subroutine DSNTIAR to generate the error message text in the SQLCA.

Suppose you want your Db2 COBOL application to check for deadlocks and timeouts, and you want to make sure your cursors are closed before continuing. You use the statement WHENEVER SQLERROR to transfer control to an error routine when your application receives a negative SQLCODE.

In your error routine, you write a section that checks for SQLCODE -911 or -913. You can receive either of these SQLCODEs when a deadlock or timeout occurs. When one of these errors occurs, the error routine closes your cursors by issuing the statement:

```sql
EXEC SQL CLOSE cursor-name
```

An SQLCODE of 0 or -501 resulting from that statement indicates that the close was successful.

To use DSNTIAR to generate the error message text, first follow these steps:

1. Choose a logical record length (lrecl) of the output lines. For this example, assume lrecl is 72 (to fit on a terminal screen) and is stored in the variable named ERROR-TEXT-LEN.
2. Define a message area in your COBOL application. Assuming you want an area for up to 10 lines of length 72, you should define an area of 720 bytes, plus a 2-byte area that specifies the total length of the message output area.

```
01  ERROR-MESSAGE.
    02  ERROR-LEN   PIC S9(4)  COMP VALUE +720.
    02  ERROR-TEXT   PIC X(72)  OCCURS 10 TIMES
        77  ERROR-TEXT-LEN      PIC S9(9)  COMP VALUE +72.
```

For this example, the name of the message area is ERROR-MESSAGE.
3. Make sure you have an SQLCA. For this example, assume the name of the SQLCA is SQLCA.

To display the contents of the SQLCA when SQLCODE is 0 or -501, call DSNTIAR after the SQL statement that produces SQLCODE 0 or -501:

```sql
CALL 'DSNTIAR' USING SQLCA ERROR-MESSAGE ERROR-TEXT-LEN.
```

You can then print the message output area just as you would any other variable. Your message might look like this:

```
DSNT408I SQLCODE = -501, ERROR: THE CURSOR IDENTIFIED IN A FETCH OR CLOSE STATEMENT IS NOT OPEN
DSNT415I SQLERRORP = DSNXERT SQL PROCEDURE DETECTING ERROR
DSNT416I SQLERRD = -315 0 0 -1 0 0 SQL DIAGNOSTIC INFORMATION
DSNT416I SQLERRD = X'FFFFFEC5' X'00000000' X'00000000'
```
Checking the execution of SQL statements by using SQLCODE and SQLSTATE

Whenever an SQL statement executes, the SQLCODE and SQLSTATE fields of the SQLCA receive a return code.

**Procedure**

You can declare SQLCODE and SQLSTATE (SQLCOD and SQLSTA in Fortran) as stand-alone host variables. If you specify the STDSQL(YES) precompiler option, these host variables receive the return codes, and you should not include an SQLCA in your program.

Portable applications should use SQLSTATE instead of SQLCODE, although SQLCODE values can provide additional Db2-specific information about an SQL error or warning.

An advantage to using the SQLCODE field is that it can provide more specific information than the SQLSTATE. Many of the SQLCODEs have associated tokens in the SQLCA that indicate, for example, which object incurred an SQL error. However, an SQL standard application uses only SQLSTATE.

**SQLCODE**

Db2 returns the following codes in SQLCODE:

- If SQLCODE = 0, execution was successful.
- If SQLCODE > 0, execution was successful with a warning.
- If SQLCODE < 0, execution was not successful.

SQLCODE 100 indicates that no data was found.

The meaning of SQLCODEs other than 0 and 100 varies with the particular product implementing SQL.

**SQLSTATE**

SQLSTATE enables an application program to check for errors in the same way for different IBM database management systems.

Related tasks

- Defining the SQL communications area, SQLSTATE, and SQLCODE in assembler
- Defining the SQL communications area, SQLSTATE, and SQLCODE in C and C++
- Defining the SQL communications area, SQLSTATE, and SQLCODE in COBOL
- Defining the SQL communications area, SQLSTATE, and SQLCODE in Fortran
- Defining the SQL communications area, SQLSTATE, and SQLCODE in PL/I
- Defining the SQL communications area, SQLSTATE, and SQLCODE in REXX
When Db2 prepares a REXX program that contains SQL statements, Db2 automatically includes an SQLCA in the program.

**Related reference**

SQLSTATE values and common error codes (Db2 Codes)

### Checking the execution of SQL statements by using the WHENEVER statement

The WHENEVER statement causes Db2 to check the SQLCA and continue processing your program. If an error, exception, or warning occurs, Db2 branches to another area in your program. The condition handling area of your program can then examine the SQLCODE or SQLSTATE to react specifically to the error or exception.

**About this task**

The WHENEVER statement is not supported for REXX.

The WHENEVER statement enables you to specify what to do if a general condition is true. You can specify more than one WHENEVER statement in your program. When you do this, the first WHENEVER statement applies to all subsequent SQL statements in the source program until the next WHENEVER statement.

The WHENEVER statement looks like this:

```
EXEC SQL
WHENEVER condition action
END-EXEC
```

The *condition* of the WHENEVER statement is one of these three values:

- **SQLWARNING**
  Indicates what to do when SQLWARN0 = W or SQLCODE contains a positive value other than 100. Db2 can set SQLWARN0 for several reasons—for example, if a column value is truncated when moved into a host variable. Your program might not regard this as an error.

- **SQLERROR**
  Indicates what to do when Db2 returns an error code as the result of an SQL statement (SQLCODE < 0).

- **NOT FOUND**
  Indicates what to do when Db2 cannot find a row to satisfy your SQL statement or when there are no more rows to fetch (SQLCODE = 100).

The *action* of the WHENEVER statement is one of these two values:

- **CONTINUE**
  Specifies the next sequential statement of the source program.

- **GOTO** or **GO TO** *host-label*
  Specifies the statement identified by *host-label*. For *host-label*, substitute a single token, preceded by an optional colon. The form of the token depends on the host language. In COBOL, for example, it can be *section-name* or an unqualified *paragraph-name*.

The WHENEVER statement must precede the first SQL statement it is to affect. However, if your program checks SQLCODE directly, you must check SQLCODE after each SQL statement.

**Related concepts**

- REXX applications that issue SQL statements
- You can code SQL statements in a REXX programs wherever you can use REXX commands.

**Related reference**

- WHENEVER (Db2 SQL)
**Checking the execution of SQL statements by using the GET DIAGNOSTICS statement**

One way to check whether an SQL statement executed successfully is to ask Db2 to return the diagnostic information about the last executed SQL statement.

**Procedure**

You can use the GET DIAGNOSTICS statement to return diagnostic information about the last SQL statement that was executed.

You can request individual items of diagnostic information from the following groups of items:

- **Statement items**, which contain information about the SQL statement as a whole
- **Condition items**, which contain information about each error or warning that occurred during the execution of the SQL statement
- **Connection items**, which contain information about the SQL statement if it was a CONNECT statement

In addition to requesting individual items, you can request that GET DIAGNOSTICS return all diagnostic items that are set during the execution of the last SQL statement as a single string.

In SQL procedures, you can also retrieve diagnostic information by using handlers. Handlers tell the procedure what to do if a particular error occurs.

Use the GET DIAGNOSTICS statement to handle multiple SQL errors that might result from the execution of a single SQL statement. First, check SQLSTATE (or SQLCODE) to determine whether diagnostic information should be retrieved by using GET DIAGNOSTICS. This method is especially useful for diagnosing problems that result from a multiple-row INSERT that is specified as NOT ATOMIC CONTINUE ON SQLEXCEPTION and multiple row MERGE statements.

Even if you use only the GET DIAGNOSTICS statement in your application program to check for conditions, you must either include the instructions required to use the SQLCA or you must declare SQLSTATE (or SQLCODE) separately in your program.

When you use the GET DIAGNOSTICS statement, you assign the requested diagnostic information to host variables. Declare each target host variable with a data type that is compatible with the data type of the requested item.

To retrieve condition information, you must first retrieve the number of condition items (that is, the number of errors and warnings that Db2 detected during the execution of the last SQL statement). The number of condition items is at least one. If the last SQL statement returned SQLSTATE '00000' (or SQLCODE 0), the number of condition items is one.

**Example: Using GET DIAGNOSTICS with multiple-row INSERT**

You want to display diagnostic information for each condition that might occur during the execution of a multiple-row INSERT statement in your application program. You specify the INSERT statement as NOT ATOMIC CONTINUE ON SQLEXCEPTION, which means that execution continues regardless of the failure of any single-row insertion. Db2 does not insert the row that was processed at the time of the error.

In the following example, the first GET DIAGNOSTICS statement returns the number of rows inserted and the number of conditions returned. The second GET DIAGNOSTICS statement returns the following items for each condition: SQLCODE, SQLSTATE, and the number of the row (in the rowset that was being inserted) for which the condition occurred.

```
EXEC SQL BEGIN DECLARE SECTION;
  long row_count, num_condns, i;
  long ret_sqlcode, row_num;
  char ret_sqlstate[6];
  ...
EXEC SQL END DECLARE SECTION;
...
EXEC SQL
  INSERT INTO DSN8C10.ACT
```
In the activity table, the ACTNO column is defined as SMALLINT. Suppose that you declare the host-variable array hva1 as an array with data type long, and you populate the array so that the value for the fourth element is 32768.

If you check the SQLCA values after the INSERT statement, the value of SQLCODE is equal to 0, the value of SQLSTATE is '00000', and the value of SQLERRD(3) is 9 for the number of rows that were inserted. However, the INSERT statement specified that 10 rows were to be inserted.

The GET DIAGNOSTICS statement provides you with the information that you need to correct the data for the row that was not inserted. The printed output from your program looks like this:

```
Number of rows inserted = 9
SQLCODE = -302, SQLSTATE = 22003, ROW NUMBER = 4
```

The value 32768 for the input variable is too large for the target column ACTNO. You can print the MESSAGE_TEXT condition item.

**What to do next**

When you use the GET DIAGNOSTICS statement, you assign the requested diagnostic information to host variables. Declare each target host variable with a data type that is compatible with the data type of the requested item.

To retrieve condition information, you must first retrieve the number of condition items (that is, the number of errors and warnings that Db2 detected during the execution of the last SQL statement). The number of condition items is at least one. If the last SQL statement returned SQLSTATE '00000' (or SQLCODE 0), the number of condition items is one.

**Related concepts**

Handlers in an SQL procedure
If an error occurs when an SQL procedure executes, the procedure ends unless you include statements to tell the procedure to perform some other action. These statements are called handlers.

**Related reference**

Data types for GET DIAGNOSTICS items
You can use the GET DIAGNOSTICS statement to request information about the statement, condition, and connection for the last SQL statement that was executed. You must declare each target host variable with a data type that is compatible with the data type of the requested item.

GET DIAGNOSTICS (Db2 SQL)

**Related information**

-302 (Db2 Codes)
Data types for GET DIAGNOSTICS items

You can use the GET DIAGNOSTICS statement to request information about the statement, condition, and connection for the last SQL statement that was executed. You must declare each target host variable with a data type that is compatible with the data type of the requested item.

The following tables specify the data types for the statement, condition, and connection information items that you can request by using the GET DIAGNOSTICS statement.

**Table 93. Data types for GET DIAGNOSTICS items that return statement information**

<table>
<thead>
<tr>
<th>Item</th>
<th>Description</th>
<th>Data type</th>
</tr>
</thead>
<tbody>
<tr>
<td>DB2_GET_DIAGNOSTICS_DIAGNOSTICS</td>
<td>After a GET DIAGNOSTICS statement, if any error or warning occurred, this item contains all of the diagnostics as a single string.</td>
<td>VARCHAR(32672)</td>
</tr>
<tr>
<td>DB2_LAST_ROW</td>
<td>After a multiple-row FETCH statement, this item contains a value of +100 if the last row in the table is in the rowset that was returned.</td>
<td>INTEGER</td>
</tr>
<tr>
<td>DB2_NUMBER_PARAMETER_MARKERS</td>
<td>After a PREPARE statement, this item contains the number of parameter markers in the prepared statement.</td>
<td>INTEGER</td>
</tr>
<tr>
<td>DB2_NUMBER_RESULT_SETS</td>
<td>After a CALL statement that invokes a stored procedure, this item contains the number of result sets that are returned by the procedure.</td>
<td>INTEGER</td>
</tr>
<tr>
<td>DB2_NUMBER_ROWS</td>
<td>After an OPEN or FETCH statement for which the size of the result table is known, this item contains the number of rows in the result table. After a PREPARE statement, this item contains the estimated number of rows in the result table for the prepared statement. For SENSITIVE DYNAMIC cursors, this item contains the approximate number of rows. Otherwise, or if the server only returns an SQLCA, the value zero is returned.</td>
<td>DECIMAL(31,0)</td>
</tr>
<tr>
<td>DB2_RETURN_STATUS</td>
<td>After a CALL statement that invokes an SQL procedure, this item contains the return status if the procedure contains a RETURN statement.</td>
<td>INTEGER</td>
</tr>
<tr>
<td>DB2_SQL_ATTR_CURSOR_HOLD</td>
<td>After an ALLOCATE or OPEN statement, this item indicates whether the cursor can be held open across multiple units of work (Y or N).</td>
<td>CHAR(1)</td>
</tr>
<tr>
<td>DB2_SQL_ATTR_CURSOR_ROWSET</td>
<td>After an ALLOCATE or OPEN statement, this item indicates whether the cursor can use rowset positioning (Y or N).</td>
<td>CHAR(1)</td>
</tr>
<tr>
<td>Item</td>
<td>Description</td>
<td>Data type</td>
</tr>
<tr>
<td>----------------------------------</td>
<td>-----------------------------------------------------------------------------</td>
<td>-----------</td>
</tr>
<tr>
<td>DB2_SQL_ATTR_CURSOR_SCROLLABLE</td>
<td>After an ALLOCATE or OPEN statement, this item indicates whether the cursor is scrollable (Y or N).</td>
<td>CHAR(1)</td>
</tr>
<tr>
<td>DB2_SQL_ATTR_CURSOR_SENSITIVITY</td>
<td>After an ALLOCATE or OPEN statement, this item indicates whether the cursor shows updates made by other processes (sensitivity I or S).</td>
<td>CHAR(1)</td>
</tr>
<tr>
<td>DB2_SQL_ATTR_CURSOR_TYPE</td>
<td>After an ALLOCATE or OPEN statement, this item indicates whether the cursor is forward (F), declared static (S for INSENSITIVE or SENSITIVE STATIC, or dynamic (D for SENSITIVE DYNAMIC).</td>
<td>CHAR(1)</td>
</tr>
<tr>
<td>DB2_SQL_NESTING_LEVEL</td>
<td>After a CALL statement, this item identifies the current level of nesting or recursion in effect when the GET DIAGNOSTICS statement was executed. Each level of nesting corresponds to a nested or recursive invocation of a packaged SQL function, packaged SQL procedure, or trigger. If the GET DIAGNOSTICS statement is executed outside of a level of nesting, the value zero is returned. When an application connects to another server the value is reset to zero.</td>
<td>INTEGER</td>
</tr>
<tr>
<td>MORE</td>
<td>After any SQL statement, this item indicates whether some conditions items were discarded because of insufficient storage (Y or N).</td>
<td>CHAR(1)</td>
</tr>
<tr>
<td>NUMBER</td>
<td>After any SQL statement, this item contains the number of condition items. If no warning or error occurred, or if no previous SQL statement has been executed, the number that is returned is 1.</td>
<td>INTEGER</td>
</tr>
<tr>
<td>ROW_COUNT</td>
<td>After an insert, update, delete, or fetch, this item contains the number of rows that are deleted, inserted, updated, or fetched. After PREPARE, this item contains the estimated number of result rows in the prepared statement. After TRUNCATE, it contains -1.</td>
<td>DECIMAL(31,0)</td>
</tr>
<tr>
<td>Item</td>
<td>Description</td>
<td>Data type</td>
</tr>
<tr>
<td>---------------------------------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
<td>---------------</td>
</tr>
<tr>
<td>CATALOG_NAME</td>
<td>This item contains the server name of the table that owns a constraint that caused an error, or that caused an access rule or check violation.</td>
<td>VARCHAR(128)</td>
</tr>
<tr>
<td>CONDITION_NUMBER</td>
<td>This item contains the number of the condition.</td>
<td>INTEGER</td>
</tr>
<tr>
<td>CURSOR_NAME</td>
<td>This item contains the name of a cursor in an invalid cursor state.</td>
<td>VARCHAR(128)</td>
</tr>
<tr>
<td>DB2_ERROR_CODE1</td>
<td>This item contains an internal error code.</td>
<td>INTEGER</td>
</tr>
<tr>
<td>DB2_ERROR_CODE2</td>
<td>This item contains an internal error code.</td>
<td>INTEGER</td>
</tr>
<tr>
<td>DB2_ERROR_CODE3</td>
<td>This item contains an internal error code.</td>
<td>INTEGER</td>
</tr>
<tr>
<td>DB2_ERROR_CODE4</td>
<td>This item contains an internal error code.</td>
<td>INTEGER</td>
</tr>
<tr>
<td>DB2_INTERNAL_ERROR_POINTER</td>
<td>For some errors, this item contains a negative value that is an internal error pointer.</td>
<td>INTEGER</td>
</tr>
<tr>
<td>DB2_MESSAGE_ID</td>
<td>This item contains the message ID that corresponds to the message that is contained in the MESSAGE_TEXT diagnostic item.</td>
<td>CHAR(10)</td>
</tr>
<tr>
<td>DB2_MODULE_DETECTING_ERROR</td>
<td>After any SQL statement, this item indicates which module detected the error.</td>
<td>CHAR(8)</td>
</tr>
<tr>
<td>DB2_ORDINAL_TOKEN_n</td>
<td>After any SQL statement, this item contains the ( n )th token, where ( n ) is a value from 1 to 100.</td>
<td>VARCHAR(515)</td>
</tr>
<tr>
<td>DB2_REASON_CODE</td>
<td>After any SQL statement, this item contains the reason code for errors that have a reason code token in the message text.</td>
<td>INTEGER</td>
</tr>
<tr>
<td>DB2_RETURNED_SQLCODE</td>
<td>After any SQL statement, this item contains the SQLCODE for the condition.</td>
<td>INTEGER</td>
</tr>
<tr>
<td>DB2_ROW_NUMBER</td>
<td>After any SQL statement that involves multiple rows, this item contains the row number on which Db2 detected the condition.</td>
<td>DECIMAL(31,0)</td>
</tr>
<tr>
<td>DB2_TOKEN_COUNT</td>
<td>After any SQL statement, this item contains the number of tokens available for the condition.</td>
<td>INTEGER</td>
</tr>
<tr>
<td>MESSAGE_TEXT</td>
<td>After any SQL statement, this item contains the message text associated with the SQLCODE.</td>
<td>VARCHAR(32672)</td>
</tr>
<tr>
<td>RETURNED_SQLSTATE</td>
<td>After any SQL statement, this item contains the SQLSTATE for the condition.</td>
<td>CHAR(5)</td>
</tr>
</tbody>
</table>
### Table 94. Data types for GET DIAGNOSTICS items that return condition information (continued)

<table>
<thead>
<tr>
<th>Item</th>
<th>Description</th>
<th>Data type</th>
</tr>
</thead>
<tbody>
<tr>
<td>SERVER_NAME</td>
<td>After a CONNECT, DISCONNECT, or SET CONNECTION statement, this item contains the name of the server specified in the statement.</td>
<td>VARCHAR(128)</td>
</tr>
</tbody>
</table>

### Table 95. Data types for GET DIAGNOSTICS items that return connection information

<table>
<thead>
<tr>
<th>Item</th>
<th>Description</th>
<th>Data type</th>
</tr>
</thead>
<tbody>
<tr>
<td>DB2_AUTHENTICATION_TYPE</td>
<td>This item contains the authentication type (S, C, D, E, or blank).</td>
<td>CHAR(1)</td>
</tr>
<tr>
<td>DB2_AUTHORIZATION_ID</td>
<td>This item contains the authorization ID that is used by the connected server.</td>
<td>VARCHAR(128)</td>
</tr>
<tr>
<td>DB2_CONNECTION_STATE</td>
<td>This item indicates whether the connection is unconnected (-1), local (0), or remote (1).</td>
<td>INTEGER</td>
</tr>
<tr>
<td>DB2_CONNECTION_STATUS</td>
<td>This item indicates whether updates can be committed for the current unit of work (1 for Yes, 2 for No).</td>
<td>INTEGER</td>
</tr>
<tr>
<td>DB2_ENCRYPTION_TYPE</td>
<td>This item contains one of the following values that indicates the level of encryption for the connection:</td>
<td>CHAR(1)</td>
</tr>
<tr>
<td></td>
<td>A Only the authentication tokens (authid and password) are encrypted</td>
<td></td>
</tr>
<tr>
<td></td>
<td>D All of the data for the connection is encrypted</td>
<td></td>
</tr>
<tr>
<td>DB2_SERVER_CLASS_NAME</td>
<td>After a CONNECT or SET CONNECTION statement, this item contains the Db2 server class name.</td>
<td>VARCHAR(128)</td>
</tr>
<tr>
<td>DB2_PRODUCT_ID</td>
<td>This item contains the Db2 product signature.</td>
<td>VARCHAR(8)</td>
</tr>
</tbody>
</table>

### Related reference
GET DIAGNOSTICS (Db2 SQL)

### Handling SQL error codes

Application programs can request more information about SQL error codes from Db2.

#### Procedure

Take action based on the programming language that you use.

- “Handling SQL error codes in assembler applications” on page 570
- “Handling SQL error codes in C and C++ applications” on page 617
- “Handling SQL error codes in Cobol applications” on page 683
- “Handling SQL error codes in Fortran applications” on page 692
Displaying SQLCA fields by calling DSNTIAR
If you use the SQLCA to check whether an SQL statement executed successfully, your program needs to read the data in the appropriate SQLCA fields. One easy way to read these fields is to use the assembler subroutine DSNTIAR.

Checking the execution of SQL statements by using the GET DIAGNOSTICS statement
One way to check whether an SQL statement executed successfully is to ask Db2 to return the diagnostic information about the last executed SQL statement.

Related reference
GET DIAGNOSTICS (Db2 SQL)

Arithmetic and conversion errors
You can track arithmetic and conversion errors by using indicator variables. An indicator variable contains a small integer value that indicates some information about the associated host variable.

Numeric or character conversion errors or arithmetic expression errors can set an indicator variable to -2. For example, division by zero and arithmetic overflow do not necessarily halt the execution of a SELECT statement. If you use indicator variables and an error occurs in the SELECT list, the statement can continue to execute and return good data for rows in which the error does not occur.

For rows in which a conversion or arithmetic expression error does occur, the indicator variable indicates that one or more selected items have no meaningful value. The indicator variable flags this error with a -2 for the affected host variable and an SQLCODE of +802 (SQLSTATE '01519') in the SQLCA.

Writing applications that enable users to create and modify tables
You can write a Db2 application that enables users to create new tables, add columns to them, increase the length of columns, rearrange the columns, and drop columns.

Procedure
To create new tables:
- Use the CREATE TABLE statement.
To add columns or increase the length of columns:
- Use the ALTER TABLE statement with the ADD COLUMN clause or the ALTER COLUMN clause.
  Added columns initially contain either the null value or a default value. Both CREATE TABLE and ALTER TABLE, like any data definition statement, are relatively expensive to execute. Also consider the effects of locks.
To drop columns:
- Use the ALTER TABLE statement with the DROP COLUMN clause.
  Dropping a column from a table is a pending-definition change unless the table space is defined with the DEFINE NO option. The column is not removed from the table until the REORG utility is run on the table space. If you are planning on dropping a column from a table in addition to making other changes to the table, make all changes that take effect immediately, prior to issuing the ALTER TABLE statement with the DROP COLUMN clause.
To rearrange columns:
- Drop the table and create the table again, with the columns you want, in the order you want.
  Consider creating a view on the table, which includes only the columns that you want, in the order that you want, as an alternative to redefining the table.
Saving SQL statements that are translated from user requests

If your program translates requests from users into SQL statements and allows users to save their requests, your program can improve performance by saving those translated statements.

About this task
A program translates requests from users into SQL statements before executing them, and users can save a request.

Procedure
Save the corresponding SQL statements in a table with a column having a data type of VARCHAR(n), where n is the maximum length of any SQL statement.

You must save the source SQL statements, not the prepared versions. That means that you must retrieve and then prepare each statement before executing the version stored in the table. In essence, your program prepares an SQL statement from a character string and executes it dynamically.

Related tasks
Including dynamic SQL in your program
Dynamic SQL is prepared and executed while the program is running.

XML data in embedded SQL applications

Embedded SQL applications that are written in assembler language, C, C++, COBOL, or PL/I can update and retrieve data in XML columns.

In embedded SQL applications, you can:
• Store an entire XML document in an XML column using INSERT or UPDATE statements.
• Retrieve an entire XML document from an XML column using SELECT statements.
• Retrieve a sequence from a document in an XML column by using the SQL XMLQUERY function within a SELECT or FETCH statement, to retrieve the sequence into a textual XML string in the database, and then retrieve the data into an application variable.

Recommendation: Follow these guidelines when you write embedded SQL applications:
• Avoid using the XMLPARSE and XMSERIALIZE functions.
  Let Db2 do the conversions between the external and internal XML formats implicitly.
• Use XML host variables for input and output.
  Doing so allows Db2 to process values as XML data instead of character or binary string data. If the application cannot use XML host variables, it should use binary string host variables to minimize character conversion issues.
• Avoid character conversion by using UTF-8 host variables for input and output of XML values whenever possible.
Host variable data types for XML data in embedded SQL applications

Db2 provides XML host variable types for assembler, C, C++, COBOL, and PL/I.

Those types are:

- XML AS BLOB
- XML AS CLOB
- XML AS DBCLOB
- XML AS BLOB_FILE (C, C++, or PL/I) or XML AS BLOB-FILE (COBOL)
- XML AS CLOB_FILE (C, C++, or PL/I) or XML AS CLOB-FILE (COBOL)
- XML AS DBCLOB_FILE (C, C++, or PL/I) or XML AS DBCLOB-FILE (COBOL)

The XML host variable types are compatible only with the XML column data type.

You can use BLOB, CLOB, DBCLOB, CHAR, VARCHAR, GRAPHIC, VARGRAPHIC, BINARY, or VARBINARY host variables to update XML columns. You can convert the host variable data types to the XML type using the XMLPARSE function, or you can let the Db2 database server perform the conversion implicitly.

You can use BLOB, CLOB, DBCLOB, CHAR, VARCHAR, GRAPHIC, VARGRAPHIC, BINARY, or VARBINARY host variables to retrieve data from XML columns. You can convert the XML data to the host variable type using the XMLSERIALIZE function, or you can let the Db2 database server perform the conversion implicitly.

The following examples show you how to declare XML host variables in each supported language. In each table, the left column contains the declaration that you code in your application program. The right column contains the declaration that Db2 generates.

Declarations of XML host variables in assembler

The following table shows assembler language declarations for some typical XML types.

<table>
<thead>
<tr>
<th>You declare this variable</th>
<th>Db2 generates this variable</th>
</tr>
</thead>
</table>
| **BLOB_XML** SQL TYPE IS XML AS BLOB 1M | BLOB_XML DS 0FL4  
BLOB_XML_LENGTH DS FL4  
BLOB_XML_DATA DS CL65535"1" on page 543  
ORG **(983041)" |
| **CLOB_XML** SQL TYPE IS XML AS CLOB 40000K | CLOB_XML DS 0FL4  
CLOB_XML_LENGTH DS FL4  
CLOB_XML_DATA DS CL65535"1" on page 543  
ORG **(40894465)" |
| **DBCLOB_XML** SQL TYPE IS XML AS DBCLOB 4000K | DBCLOB_XML DS 0FL4  
DBCLOB_XML_LENGTH DS FL4  
DBCLOB_XML_DATA DS GL65534"2" on page 543  
ORG **(4930466)" |
| **BLOB_XML_FILE** SQL TYPE IS XML AS BLOB_FILE | BLOB_XML_FILE DS 0FL4  
BLOB_XML_FILE_NAME_LENGTH DS FL4  
BLOB_XML_FILE_DATA_LENGTH DS FL4  
BLOB_XML_FILE_OPTIONS DS FL4  
BLOB_XML_FILE_NAME DS CL255 |
Table 96. Example of assembler XML variable declarations (continued)

<table>
<thead>
<tr>
<th>You declare this variable</th>
<th>Db2 generates this variable</th>
</tr>
</thead>
</table>
| **CLOB.XML.FILE** SQL TYPE IS XML AS CLOB_FILE | CLOB.XML.FILE DS 0FL4  
  CLOB.XML.FILE_NAME_LENGTH DS FL4  
  CLOB.XML.FILE_DATA_LENGTH DS FL4  
  CLOB.XML.FILE_FILE_OPTIONS DS FL4  
  CLOB.XML.FILE_NAME DS CL255 |
| **DBCLOB.XML.FILE** SQL TYPE IS XML AS DBCLOB_FILE | DBCLOB.XML.FILE DS 0FL4  
  DBCLOB.XML.FILE_NAME_LENGTH DS FL4  
  DBCLOB.XML.FILE_DATA_LENGTH DS FL4  
  DBCLOB.XML.FILE_FILE_OPTIONS DS FL4  
  DBCLOB.XML.FILE_NAME DS CL255 |

Notes:
1. Because assembler language allows character declarations of no more than 65535 bytes, Db2 separates the host language declarations for XML AS BLOB and XML AS CLOB host variables that are longer than 65535 bytes into two parts.
2. Because assembler language allows graphic declarations of no more than 65534 bytes, Db2 separates the host language declarations for XML AS DBCLOB host variables that are longer than 65534 bytes into two parts.

Declarations of XML host variables in C and C++

The following table shows C and C++ language declarations that are generated by the Db2 precompiler for some typical XML types. The declarations that the Db2 coprocessor generates might be different.

Table 97. Examples of C language variable declarations

<table>
<thead>
<tr>
<th>You declare this variable</th>
<th>Db2 generates this variable</th>
</tr>
</thead>
</table>
| SQL TYPE IS XML AS BLOB (1M) blob_xml; | struct  
  { unsigned long length;  
  char data [(1048576)];  
} blob_xml; |
| SQL TYPE IS XML AS CLOB(40000K) clob_xml; | struct  
  { unsigned long length;  
  char data [(40960000)];  
} clob_xml; |
| SQL TYPE IS XML AS DBCLOB (4000K) dbclob_xml; | struct  
  { unsigned long length;  
  unsigned short data [(4096000)];  
} dbclob_xml; |
| SQL TYPE IS XML AS BLOB_FILE blob_xml_file; | struct  
  { unsigned long name_length;  
  unsigned long data_length;  
  unsigned long file_options;  
  char name [(255)];  
} blob_xml_file; |
Table 97. Examples of C language variable declarations (continued)

<table>
<thead>
<tr>
<th>You declare this variable</th>
<th>Db2 generates this variable</th>
</tr>
</thead>
</table>
| SQL TYPE IS XML AS CLOB_FILE clob_xml_file;                                                | struct {
|                                                                                           |    unsigned long name_length;
|                                                                                           |    unsigned long data_length;
|                                                                                           |    unsigned long file_options;
|                                                                                           |    char name??(255??);
|                                                                                           | } clob_xml_file;                                                                             |
| SQL TYPE IS XML AS DBCLOB_FILE dbclob_xml_file;                                           | struct {
|                                                                                           |    unsigned long name_length;
|                                                                                           |    unsigned long data_length;
|                                                                                           |    unsigned long file_options;
|                                                                                           |    char name??(255??);
|                                                                                           | } dbclob_xml_file;                                                                           |

Declarations of XML host variables in COBOL

The declarations that are generated for COBOL differ, depending on whether you use the Db2 precompiler or the Db2 coprocessor.

The following table shows COBOL declarations that the Db2 precompiler generates for some typical XML types.

Table 98. Examples of COBOL variable declarations by the Db2 precompiler

<table>
<thead>
<tr>
<th>You declare this variable</th>
<th>Db2 precompiler generates this variable</th>
</tr>
</thead>
<tbody>
<tr>
<td>01  BLOB-XML USAGE IS SQL TYPE IS XML AS BLOB(1M).</td>
<td>01  BLOB-XML.</td>
</tr>
<tr>
<td></td>
<td>02  BLOB-XML-LENGTH PIC 9(9) COMP.</td>
</tr>
<tr>
<td></td>
<td>02  BLOB-XML-DATA.</td>
</tr>
<tr>
<td></td>
<td>49  FILLER PIC X(32767). “1” on page 545</td>
</tr>
<tr>
<td></td>
<td>49  FILLER PIC X(32767). Repeat 30 times</td>
</tr>
<tr>
<td></td>
<td>:</td>
</tr>
<tr>
<td></td>
<td>49  FILLER PIC X(1048576-32*32767).</td>
</tr>
<tr>
<td>01  CLOB-XML USAGE IS SQL TYPE IS XML AS CLOB(40000K).</td>
<td>01  CLOB-XML.</td>
</tr>
<tr>
<td></td>
<td>02  CLOB-XML-LENGTH PIC 9(9) COMP.</td>
</tr>
<tr>
<td></td>
<td>02  CLOB-XML-DATA.</td>
</tr>
<tr>
<td></td>
<td>49  FILLER PIC X(32767). “1” on page 545</td>
</tr>
<tr>
<td></td>
<td>49  FILLER PIC X(32767). Repeat 1248 times</td>
</tr>
<tr>
<td></td>
<td>:</td>
</tr>
<tr>
<td></td>
<td>49  FILLER PIC X(40960000-1250*32767).</td>
</tr>
<tr>
<td>01  DBCLOB-XML USAGE IS SQL TYPE IS XML AS DBCLOB(4000K).</td>
<td>01  DBCLOB-XML.</td>
</tr>
<tr>
<td></td>
<td>02  DBCLOB-XML-LENGTH PIC 9(9) COMP.</td>
</tr>
<tr>
<td></td>
<td>02  DBCLOB-XML-DATA.</td>
</tr>
<tr>
<td></td>
<td>49  FILLER PIC G(32767). “2” on page 545</td>
</tr>
<tr>
<td></td>
<td>49  FILLER PIC G(32767). USAGE DISPLAY-1.</td>
</tr>
<tr>
<td></td>
<td>:</td>
</tr>
<tr>
<td></td>
<td>49  FILLER PIC G(40960000-125*32767) USAGE DISPLAY-1.</td>
</tr>
</tbody>
</table>
Table 98. Examples of COBOL variable declarations by the Db2 precompiler (continued)

<table>
<thead>
<tr>
<th>You declare this variable</th>
<th>Db2 precompiler generates this variable</th>
</tr>
</thead>
<tbody>
<tr>
<td>01  BLOB-XML-FILE USAGE IS SQL</td>
<td>01  BLOB-XML-FILE.</td>
</tr>
<tr>
<td>TYPE IS XML AS BLOB-FILE.</td>
<td>49  BLOB-XML-FILE-NAME-LENGTH PIC S9(9) COMP-5 SYNC.</td>
</tr>
<tr>
<td></td>
<td>49  BLOB-XML-FILE-DATA-LENGTH PIC S9(9) COMP-5.</td>
</tr>
<tr>
<td></td>
<td>49  BLOB-XML-FILE-FILE-OPTION PIC S9(9) COMP-5.</td>
</tr>
<tr>
<td></td>
<td>49  BLOB-XML-FILE-NAME PIC (255).</td>
</tr>
<tr>
<td>01  CLOB-XML-FILE USAGE IS SQL</td>
<td>01  CLOB-XML-FILE.</td>
</tr>
<tr>
<td>TYPE IS XML AS CLOB-FILE.</td>
<td>49  CLOB-XML-FILE-NAME-LENGTH PIC S9(9) COMP-5 SYNC.</td>
</tr>
<tr>
<td></td>
<td>49  CLOB-XML-FILE-DATA-LENGTH PIC S9(9) COMP-5.</td>
</tr>
<tr>
<td></td>
<td>49  CLOB-XML-FILE-FILE-OPTION PIC S9(9) COMP-5.</td>
</tr>
<tr>
<td></td>
<td>49  CLOB-XML-FILE-NAME PIC (255).</td>
</tr>
<tr>
<td>01  DBCLOB-XML-FILE USAGE IS SQL</td>
<td>01  DBCLOB-XML-FILE.</td>
</tr>
<tr>
<td>TYPE IS XML AS DBCLOB-FILE.</td>
<td>49  DBCLOB-XML-FILE-NAME-LENGTH PIC S9(9) COMP-5 SYNC.</td>
</tr>
<tr>
<td></td>
<td>49  DBCLOB-XML-FILE-DATA-LENGTH PIC S9(9) COMP-5.</td>
</tr>
<tr>
<td></td>
<td>49  DBCLOB-XML-FILE-FILE-OPTION PIC S9(9) COMP-5.</td>
</tr>
<tr>
<td></td>
<td>49  DBCLOB-XML-FILE-NAME PIC (255).</td>
</tr>
</tbody>
</table>

Notes:

1. For XML AS BLOB or XML AS CLOB host variables that are greater than 32767 bytes in length, Db2 creates multiple host language declarations of 32767 or fewer bytes.
2. For XML AS DBCLOB host variables that are greater than 32767 double-byte characters in length, Db2 creates multiple host language declarations of 32767 or fewer double-byte characters.

Declarations of XML host variables in PL/I

The declarations that are generated for PL/I differ, depending on whether you use the Db2 precompiler or the Db2 coprocessor.

The following table shows PL/I declarations that the Db2 precompiler generates for some typical XML types.

Table 99. Examples of PL/I variable declarations

<table>
<thead>
<tr>
<th>You declare this variable</th>
<th>Db2 precompiler generates this variable</th>
</tr>
</thead>
<tbody>
<tr>
<td>DCL BLOB.XML</td>
<td>DCL 1 BLOB.XML,</td>
</tr>
<tr>
<td>SQL TYPE IS XML AS BLOB</td>
<td>2 BLOB.XML_LENGTH BIN FIXED(31),</td>
</tr>
<tr>
<td>(1M);</td>
<td>2 BLOB.XML_DATA, &quot;1&quot; on page 547</td>
</tr>
<tr>
<td></td>
<td>3 BLOB.XML_DATA1 (32) CHAR(32767),</td>
</tr>
<tr>
<td></td>
<td>3 BLOB.XML_DATA2 CHAR(32);</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>DCL CLOB.XML</td>
<td>DCL 1 CLOB.XML,</td>
</tr>
<tr>
<td>SQL TYPE IS XML AS CLOB</td>
<td>2 CLOB.XML_LENGTH BIN FIXED(31),</td>
</tr>
<tr>
<td>(40000K);</td>
<td>2 CLOB.XML_DATA, &quot;1&quot; on page 547</td>
</tr>
<tr>
<td></td>
<td>3 CLOB.XML_DATA1 (1250) CHAR(32767),</td>
</tr>
<tr>
<td></td>
<td>3 CLOB.XML_DATA2 CHAR(1250);</td>
</tr>
<tr>
<td>You declare this variable</td>
<td>Db2 precompiler generates this variable</td>
</tr>
<tr>
<td>---------------------------------------------------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td><code>DCL DBCLOB.XML SQL TYPE IS XML AS DBCLOB (4000K);</code></td>
<td><code>DCL</code> <code>DBCLOB.XML</code>, <code>DBCLOB.XML_LENGTH BIN FIXED(31)</code>, <code>DBCLOB.XML_DATA</code></td>
</tr>
<tr>
<td></td>
<td>3 <code>DBCLOB.XML_DATA1 (250) GRAPHIC(16383)</code>, <code>DBCLOB.XML_DATA2 GRAPHIC(250)</code>;</td>
</tr>
<tr>
<td><code>DCL BLOB.XML_FILE SQL TYPE IS XML AS BLOB_FILE;</code></td>
<td><code>DCL</code> <code>BLOB.XML_FILE</code>, <code>BLOB.XML_FILE_NAME_LENGTH BIN FIXED(31) ALIGNED</code>, <code>BLOB.XML_FILE_DATA_LENGTH BIN FIXED(31)</code>, <code>BLOB.XML_FILE_FILE_OPTIONS BIN FIXED(31)</code>, <code>BLOB.XML_FILE_NAME CHAR(255)</code>;</td>
</tr>
<tr>
<td><code>DCL CLOB.XML_FILE SQL TYPE IS XML AS CLOB_FILE;</code></td>
<td><code>DCL</code> <code>CLOB.XML_FILE</code>, <code>CLOB.XML_FILE_NAME_LENGTH BIN FIXED(31) ALIGNED</code>, <code>CLOB.XML_FILE_DATA_LENGTH BIN FIXED(31)</code>, <code>CLOB.XML_FILE_FILE_OPTIONS BIN FIXED(31)</code>, <code>CLOB.XML_FILE_NAME CHAR(255)</code>;</td>
</tr>
<tr>
<td><code>DCL DBCLOB.XML_FILE SQL TYPE IS XML AS DBCLOB_FILE;</code></td>
<td><code>DCL</code> <code>DBCLOB.XML_FILE</code>, <code>DBCLOB.XML_FILE_NAME_LENGTH BIN FIXED(31) ALIGNED</code>, <code>DBCLOB.XML_FILE_DATA_LENGTH BIN FIXED(31)</code>, <code>DBCLOB.XML_FILE_FILE_OPTIONS BIN FIXED(31)</code>, <code>DBCLOB.XML_FILE_NAME CHAR(255)</code>;</td>
</tr>
</tbody>
</table>
Table 99. Examples of PL/I variable declarations (continued)

<table>
<thead>
<tr>
<th>You declare this variable</th>
<th>Db2 precompiler generates this variable</th>
</tr>
</thead>
</table>

Notes:

1. For XML AS BLOB or XML AS CLOB host variables that are greater than 32767 bytes in length, Db2 creates host language declarations in the following way:
   - If the length of the XML is greater than 32767 bytes and evenly divisible by 32767, Db2 creates an array of 32767-byte strings. The dimension of the array is length/32767.
   - If the length of the XML is greater than 32767 bytes but not evenly divisible by 32767, Db2 creates two declarations: The first is an array of 32767 byte strings, where the dimension of the array, n, is length/32767. The second is a character string of length length-n*32767.

2. For XML AS DBCLOB host variables that are greater than 16383 double-byte characters in length, Db2 creates host language declarations in the following way:
   - If the length of the XML is greater than 16383 characters and evenly divisible by 16383, Db2 creates an array of 16383-character strings. The dimension of the array is length/16383.
   - If the length of the XML is greater than 16383 characters but not evenly divisible by 16383, Db2 creates two declarations: The first is an array of 16383 byte strings, where the dimension of the array, m, is length/16383. The second is a character string of length length-m*16383.

Related concepts

Insertion of rows with XML column values (Db2 Programming for XML)
Retrieving XML data (Db2 Programming for XML)
Updates of XML columns (Db2 Programming for XML)

XML column updates in embedded SQL applications

When you update or insert data into XML columns of a Db2 table, the input data must be in the textual XML format.

The encoding of XML data can be derived from the data itself, which is known as internally encoded data, or from external sources, which is known as externally encoded data. XML data that is sent to the database server as binary data is treated as internally encoded data. XML data that is sent to the database server as character data is treated as externally encoded data.

Externally encoded data can have internal encoding. That is, the data might be sent to the database server as character data, but the data contains encoding information. Db2 does not enforce consistency of the internal and external encoding. When the internal and external encoding information differs, the external encoding takes precedence. However, if there is a difference between the external and internal encoding, intervening character conversion might have occurred on the data, and there might be data loss.

Character data in XML columns is stored in UTF-8 encoding. The database server handles conversion of the data from its internal or external encoding to UTF-8.

The following examples demonstrate how to update XML columns in assembler, C, COBOL, and PL/I applications. The examples use a table named MYCUSTOMER, which is a copy of the sample CUSTOMER table.

Example

The following example shows an assembler program that inserts data from XML AS BLOB, XML AS CLOB, and CLOB host variables into an XML column. The XML AS BLOB data is inserted as binary data, so the database server honors the internal encoding. The XML AS CLOB and CLOB data is inserted as character data, so the database server honors the external encoding.
Example

The following example shows a C language program that inserts data from XML AS BLOB, XML AS CLOB, and CLOB host variables into an XML column. The XML AS BLOB data is inserted as binary data, so the database server honors the internal encoding. The XML AS CLOB and CLOB data is inserted as character data, so the database server honors the external encoding.

```c
EXEC SQL
BEGIN DECLARE SECTION;
   SQL TYPE IS XML AS CLOB(10K) xmlBuf;
   SQL TYPE IS XML AS BLOB(10K) xmlblob;
   SQL TYPE IS CLOB(10K) clobBuf;
END DECLARE SECTION;
EXEC SQL UPDATE MYCUSTOMER SET INFO = :xmlBuf where CID = 1000;
EXEC SQL UPDATE MYCUSTOMER SET INFO = :xmlblob where CID = 1000;
EXEC SQL UPDATE MYCUSTOMER SET INFO = XMLPARSE(DOCUMENT :clobBuf) where CID = 1000;
```

Example

The following example shows a COBOL program that inserts data from XML AS BLOB, XML AS CLOB, and CLOB host variables into an XML column. The XML AS BLOB data is inserted as binary data, so the database server honors the internal encoding. The XML AS CLOB and CLOB data is inserted as character data, so the database server honors the external encoding.

```cobol
EXEC SQL UPDATE MYCUSTOMER SET INFO = :XMLBUF where CID = 1000.
EXEC SQL UPDATE MYCUSTOMER SET INFO = :XMLBLOB where CID = 1000.
EXEC SQL UPDATE MYCUSTOMER SET INFO = XMLPARSE(DOCUMENT :CLOBBUF) where CID = 1000.
```
Example
The following example shows a PL/I program that inserts data from XML AS BLOB, XML AS CLOB, and CLOB host variables into an XML column. The XML AS BLOB data is inserted as binary data, so the database server honors the internal encoding. The XML AS CLOB and CLOB data is inserted as character data, so the database server honors the external encoding.

```pli
держива
//***************
/* Host variable declarations */
//******************************/
DCL
XMLBUF SQL TYPE IS XML AS CLOB(10K),
XMLBLOB SQL TYPE IS XML AS BLOB(10K),
CLOBBUF SQL TYPE IS CLOB(10K);
//***************
/* Update an XML column with data in an XML AS CLOB host variable */
EXEC SQL UPDATE MYCUSTOMER SET INFO = :XMLBUF where CID = 1000;
//***************
/* Update an XML column with data in a CLOB host variable. Use */
/* the XMLPARSE function to convert the data to the XML type. */
EXEC SQL UPDATE MYCUSTOMER SET INFO = XMLPARSE(DOCUMENT :CLOBBUF) where CID = 1000;
```

Related concepts
- Insertion of rows with XML column values (Db2 Programming for XML)
- Updates of XML columns (Db2 Programming for XML)

XML data retrieval in embedded SQL applications
In an embedded SQL application, if you retrieve the data into a character host variable, Db2 converts the data from the UTF-8 encoding scheme to the application encoding scheme. If you retrieve the data into binary host variable, Db2 does not convert the data to another encoding scheme.

The output data is in the textual XML format.

Db2 might add an XML encoding specification to the retrieved data, depending on whether you call the XMLSERIALIZE function when you retrieve the data. If you do not call the XMLSERIALIZE function, Db2 adds the correct XML encoding specification to the retrieved data. If you call the XMLSERIALIZE function, Db2 adds an internal XML encoding declaration for UTF-8 encoding if you specify INCLUDING XMLDECLARATION in the function call. When you use INCLUDING XMLDECLARATION, you need to ensure that the retrieved data is not converted from UTF-8 encoding to another encoding.

The following examples demonstrate how to retrieve data from XML columns in assembler, C, COBOL, and PL/I applications. The examples use a table named MYCUSTOMER, which is a copy of the sample CUSTOMER table.

Example: The following example shows an assembler program that retrieves data from an XML column into XML AS BLOB, XML AS CLOB, and CLOB host variables. The data that is retrieved into an XML AS BLOB host variable is retrieved as binary data, so the database server generates an XML declaration with UTF-8 encoding. The data that is retrieved into an XML AS CLOB host variable is retrieved as character data, so the database server generates an XML declaration with an internal encoding declaration that is consistent with the external encoding. The data that is retrieved into a CLOB host variable is retrieved as character data, so the database server generates an XML declaration with a consistent internal encoding declaration. That declaration might not be consistent with the external encoding.

```asm
retreive xml column data into an xml as clob host variable
```

Chapter 4. Embedded SQL programming 549
**Example:** The following example shows a C language program that retrieves data from an XML column into XML AS BLOB, XML AS CLOB, and CLOB host variables. The data that is retrieved into an XML AS BLOB host variable is retrieved as binary data, so the database server generates an XML declaration with UTF-8 encoding. The data that is retrieved into an XML AS CLOB host variable is retrieved as character data, so the database server generates an XML declaration with an internal encoding declaration that is consistent with the external encoding. The data that is retrieved into a CLOB host variable is retrieved as character data, so the database server generates an XML declaration with an internal encoding declaration. That declaration might not be consistent with the external encoding.

```c
EXEC SQL
SELECT INFO INTO :xmlBuf
FROM MYCUSTOMER
WHERE CID = 1000

EXEC SQL
SELECT INFO INTO :xmlBlob
FROM MYCUSTOMER
WHERE CID = 1000

EXEC SQL
SELECT XMLSERIALIZE(INFO AS CLOB(10K))
INTO :clobBuf
FROM MYCUSTOMER
WHERE CID = 1000
```

Example: The following example shows a COBOL program that retrieves data from an XML column into XML AS BLOB, XML AS CLOB, and CLOB host variables. The data that is retrieved into an XML AS BLOB host variable is retrieved as binary data, so the database server generates an XML declaration with UTF-8 encoding. The data that is retrieved into an XML AS CLOB host variable is retrieved as character data, so the database server generates an XML declaration with an internal encoding declaration that is consistent with the external encoding. The data that is retrieved into a CLOB host variable is retrieved as character data, so the database server generates an XML declaration with an internal encoding declaration. That declaration might not be consistent with the external encoding.

```cobol
EXEC SQL BEGIN DECLARE SECTION;
SQL TYPE IS XML AS CLOB(10K) xmlBuf;
SQL TYPE IS XML AS BLOB(10K) xmlBlob;
EXEC SQL END DECLARE SECTION;
EXEC SQL SELECT INFO INTO :xmlBuf from myTable where CID = 1000;
EXEC SQL SELECT INFO INTO :xmlBlob from myTable where CID = 1000;
EXEC SQL SELECT XMLSERIALIZE(INFO AS CLOB(10K)) INTO :clobBuf from myTable where CID = 1000;
```
data, so the database server generates an XML declaration with an internal encoding declaration. That declaration might not be consistent with the external encoding.

---

```
**************************************************************
* Host variable declarations *
**************************************************************
01 XMLBUF  USAGE IS SQL TYPE IS XML AS CLOB(10K).
01 XMLBLOB USAGE IS SQL TYPE IS XML AS BLOB(10K).
01 CLOBBUF USAGE IS SQL TYPE IS CLOB(10K).
**************************************************************
* Retrieve data from an XML column into an XML AS CLOB host variable *
EXEC SQL SELECT INFO
  INTO :XMLBUF
  FROM MYTABLE
  WHERE CID = 1000
END-EXEC.
**************************************************************
* Retrieve data from an XML column into an XML AS BLOB host variable *
EXEC SQL SELECT INFO
  INTO :XMLBLOB
  FROM MYTABLE
  WHERE CID = 1000
END-EXEC.
**************************************************************
* RETRIEVE DATA FROM AN XML COLUMN INTO A CLOB HOST VARIABLE.        *
* BEFORE SENDING THE DATA TO THE APPLICATION, INVOKE THE             *
* XMLSERIALIZE FUNCTION TO CONVERT THE DATA FROM THE XML            *
* TYPE TO THE CLOB TYPE.                                             *
**************************************************************
EXEC SQL SELECT XMLSERIALIZE(INFO AS CLOB(10K))
  INTO :CLOBBUF
  FROM MYTABLE
  WHERE CID = 1000
END-EXEC.
```

---

**Example:** The following example shows a PL/I program that retrieves data from an XML column into XML AS BLOB, XML AS CLOB, and CLOB host variables. The data that is retrieved into an XML AS BLOB host variable is retrieved as binary data, so the database server generates an XML declaration with UTF-8 encoding. The data that is retrieved into an XML AS CLOB host variable is retrieved as character data, so the database server generates an XML declaration with an internal encoding declaration that is consistent with the external encoding. The data that is retrieved into a CLOB host variable is retrieved as character data, so the database server generates an XML declaration with an internal encoding declaration. That declaration might not be consistent with the external encoding.

---

```
/* Host variable declarations */
/* *****************************************/
DCL XMLBUF SQL TYPE IS XML AS CLOB(10K),
      XMLBLOB SQL TYPE IS XML AS BLOB(10K),
      CLOBBUF SQL TYPE IS CLOB(10K);
/* Retrieve data from an XML column into an XML AS CLOB host variable */
EXEC SQL SELECT INFO INTO :XMLBUF FROM MYTABLE WHERE CID = 1000;
/* Retrieve data from an XML column into an XML AS BLOB host variable */
EXEC SQL SELECT INFO INTO :XMLBLOB FROM MYTABLE WHERE CID = 1000;
/* Retrieve data from an XML column into a CLOB host variable */
EXEC SQL SELECT XMLSERIALIZE(INFO AS CLOB(10K))
  INTO :CLOBBUF
  FROM MYTABLE
  WHERE CID = 1000;
```

---

Retrieving XML data (Db2 Programming for XML)
Example programs that call stored procedures

Examples can be used as models when you write applications that call stored procedures. In addition, prefix.DSNNSAMP contains sample jobs DSNTEJ6P and DSNTEJ6S and programs DSN8EP1 and DSN8EP2, which you can run.

Related concepts

Sample applications supplied with Db2 for z/OS

Db2 provides sample applications to help you with Db2 programming techniques and coding practices within each of the four environments: batch, TSO, IMS, and CICS. The sample applications contain various applications that might apply to managing a company.

Assembler applications that issue SQL statements

You can code SQL statements in assembler programs wherever you can use executable statements.

Each SQL statement in an assembler program must begin with EXEC SQL. The EXEC and SQL keywords must appear on one line, but the remainder of the statement can appear on subsequent lines.

You might code an UPDATE statement in an assembler program as follows:

```
EXEC SQL UPDATE DSNBC10.DEPT  X
  SET MGRNO = :MGRNUM     X
  WHERE DEPTNO = :INTDEPT
```

Comments

You cannot include assembler comments in SQL statements. However, you can include SQL comments in any embedded SQL statement. For more information, see SQL comments (Db2 SQL).

Continuation for SQL statements

The line continuation rules for SQL statements are the same as those for assembler statements, except that you must specify EXEC SQL within one line. Any part of the statement that does not fit on one line can appear on subsequent lines, beginning at the continuation margin (column 16, the default). Every line of the statement, except the last, must have a continuation character (a non-blank character) immediately after the right margin in column 72.

Delimiters for SQL statements

Delimit an SQL statement in your assembler program with the beginning keyword EXEC SQL and an end of line or end of last continued line.

Declaring tables and views

Your assembler program should include a DECLARE statement to describe each table and view the program accesses.

Including code

To include SQL statements or assembler host variable declaration statements from a member of a partitioned data set, place the following SQL statement in the source code where you want to include the statements:

```
EXEC SQL INCLUDE member-name
```

You cannot nest SQL INCLUDE statements.

Margins

Use the precompiler option MARGINS to set a left margin, a right margin, and a continuation margin. The default values for these margins are columns 1, 71, and 16, respectively. If EXEC SQL starts before the specified left margin, the Db2 precompiler does not recognize the SQL statement. If you use the default margins, you can place an SQL statement anywhere between columns 2 and 71.

Multiple-row FETCH statements

You can use only the FETCH ... USING DESCRIPTOR form of the multiple-row FETCH statement in an assembler program. The Db2 precompiler does not recognize declarations of host-variable arrays for an assembler program.
Names
You can use any valid assembler name for a host variable. However, do not use external entry names or access plan names that begin with 'DSN' or host variable names that begin with 'SQL'. These names are reserved for Db2.

The first character of a host variable that is used in embedded SQL cannot be an underscore. However, you can use an underscore as the first character in a symbol that is not used in embedded SQL.

Statement labels
You can prefix an SQL statement with a label. The first line of an SQL statement can use a label beginning in the left margin (column 1). If you do not use a label, leave column 1 blank.

WHENEVER statement
The target for the GOTO clause in an SQL WHENEVER statement must be a label in the assembler source code and must be within the scope of the SQL statements that WHENEVER affects.

Special assembler considerations
The following considerations apply to programs written in assembler:

• To allow for reentrant programs, the precompiler puts all the variables and structures it generates within a DSECT called SQLDSECT, and it generates an assembler symbol called SQLDLEN. SQLDLEN contains the length of the DSECT. Your program must allocate an area of the size indicated by SQLDLEN, initialize it, and provide addressability to it as the DSECT SQLDSECT. The precompiler does not generate code to allocate the storage for SQLDSECT; the application program must allocate the storage.

CICS: An example of code to support reentrant programs, running under CICS, follows:

```assembly
DFHEISTG DSECT
DFHEISTG
EXEC SQL INCLUDE SQLCA
*
DS    0F
SQDNSREG EQU   R7
SQDNSTOR DS    (SQLDLEN)C  RESERVE STORAGE TO BE USED FOR SQLDSECT
:
XXPROGRM DFHEIENT CODEREG=R12,EIBREG=R11,DATAREG=R13
*
*
* SQL WORKING STORAGE
LA    SQDNSREG,SQDNSTOR     GET ADDRESS OF SQLDSECT
USING SQLDSECT,SQDNSREG     AND TELL ASSEMBLER ABOUT IT
*
```

In this example, the actual storage allocation is done by the DFHEIENT macro.

TSO: The sample program in prefix.SDSNSAMP(DSNTIAD) contains an example of how to acquire storage for the SQDNSECT in a program that runs in a TSO environment. The following example code contains pieces from prefix.SDSNSAMP(DSNTIAD) with explanations in the comments.

```assembly
DSNTIAD CSECT          CONTROL SECTION NAME
SAVE  (14,12)       ANY SAVE SEQUENCE
LR    R12,R15       CODE ADDRESSABILITY
USING DSNTIAD,R12    TELL THE ASSEMBLER
LR    R7,R1         SAVE THE PARM POINTER
*
* Allocate storage of size PRGSIZ1+SQLDSIZ, where:
* - PRGSIZ1 is the size of the DSNTIAD program area
* - SQLDSIZ is the size of the SQDNSECT, and declared
* when the DB2 precompiler includes the SQDNSECT
*
L     R6,PRGSIZ1     GET SPACE FOR USER PROGRAM
A     R6,SQLDSIZ     GET SPACE FOR SQDNSECT
GETMAIN R,LV=(6)     GET STORAGE FOR PROGRAM VARIABLES
LR    R10,R1        POINT TO IT
*
*
* Initialize the storage
*
LR    R2,R10        POINT TO THE FIELD
LR    R3,R6         GET ITS LENGTH
```
• Db2 does not process set symbols in SQL statements.
• Generated code can include more than two continuations per comment.
• Generated code uses literal constants (for example, =F'-84' ), so an LTORG statement might be necessary.
• Generated code uses registers 0, 1, 14, and 15. Register 13 points to a save area that the called program uses. Register 15 does not contain a return code after a call that is generated by an SQL statement.

**CICS:** A CICS application program uses the DFHEIENT macro to generate the entry point code. When using this macro, consider the following:

− If you use the default DATAREG in the DFHEIENT macro, register 13 points to the save area.
− If you use any other DATAREG in the DFHEIENT macro, you must provide addressability to a save area.

For example, to use SAVED, you can code instructions to save, load, and restore register 13 around each SQL statement as in the following example.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Register</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ST 13,SAVER13</td>
<td>13</td>
<td>SAVE REGISTER 13</td>
</tr>
<tr>
<td>LA 13,SAVED</td>
<td>13</td>
<td>POINT TO SAVE AREA</td>
</tr>
<tr>
<td>EXEC SQL . . .</td>
<td></td>
<td></td>
</tr>
<tr>
<td>L 13,SAVER13</td>
<td>13</td>
<td>RESTORE REGISTER 13</td>
</tr>
</tbody>
</table>

• If you have an addressability error in precompiler-generated code because of input or output host variables in an SQL statement, check to make sure that you have enough base registers.
• Do not put CICS translator options in the assembly source code. Instead, pass the options to the translator by using the PARM field.

**Handling SQL error codes**

Assembler applications can request more information about SQL errors from Db2. For more information, see “Handling SQL error codes in assembler applications” on page 570.

**Related tasks**

Overview of programming applications that access Db2 for z/OS data
Applications that interact with Db2 must first connect to Db2. They can then read, add, or modify data or manipulate Db2 objects.

Including dynamic SQL in your program
Dynamic SQL is prepared and executed while the program is running.

Setting limits for system resource usage by using the resource limit facility (Db2 Performance)

Assembler programming examples
You can write Db2 programs in assembler. These programs can access a local or remote Db2 subsystem and can execute static or dynamic SQL statements. This information contains several such programming examples.

To prepare and run these applications, use the JCL in prefix.SDSNSAMP as a model for your JCL.

Related reference
Application languages and environments for the sample applications
The sample applications demonstrate how to run Db2 applications in the TSO, IMS, or CICS environments.

Defining the SQL communications area, SQLSTATE, and SQLCODE in assembler
Assembler programs that contain SQL statements can include an SQL communications area (SQLCA) to check whether an SQL statement executed successfully. Alternatively, these programs can declare individual SQLCODE and SQLSTATE host variables.

About this task
If you specify the SQL processing option STDSQL(YES), do not define an SQLCA. If you do, Db2 ignores your SQLCA, and your SQLCA definition causes compile-time errors. If you specify the SQL processing option STDSQL(NO), include an SQLCA explicitly.

If your application contains SQL statements and does not include an SQL communications area (SQLCA), you must declare individual SQLCODE and SQLSTATE host variables. Your program can use these variables to check whether an SQL statement executed successfully.

Procedure
Choose one of the following actions:

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
</table>
| To define the SQL communications area: | a. Code the SQLCA directly in the program or use the following SQL INCLUDE statement to request a standard SQLCA declaration:  

```
EXEC SQL INCLUDE SQLCA
```

If your program is reentrant, you must include the SQLCA within a unique data area that is acquired for your task (a DSECT). For example, at the beginning of your program, specify the following code:

```
PROGAREA DSECT
EXEC SQL INCLUDE SQLCA
```

As an alternative, you can create a separate storage area for the SQLCA and provide addressability to that area.

Db2 sets the SQLCODE and SQLSTATE values in the SQLCA after each SQL statement executes. Your application should check these values to determine whether the last SQL statement was successful.
### Related tasks

**Checking the execution of SQL statements**

After executing an SQL statement, your program should check for any errors before you commit the data and handle the errors that they represent.

**Checking the execution of SQL statements by using the SQLCA**

One way to check whether an SQL statement executed successfully is to use the SQL communication area (SQLCA). This area is set apart for communication with Db2.

**Checking the execution of SQL statements by using SQLCODE and SQLSTATE**

Whenever an SQL statement executes, the SQLCODE and SQLSTATE fields of the SQLCA receive a return code.

**Defining the items that your program can use to check whether an SQL statement executed successfully**

If your program contains SQL statements, the program should define some infrastructure so that it can check whether the statements executed successfully. You can either include an SQL communications area (SQLCA), which contains SQLCODE and SQLSTATE variables, or declare individual SQLCODE and SQLSTATE host variables.

---

### Defining SQL descriptor areas (SQLDA) in assembler

If your program includes certain SQL statements, you must define at least one SQL descriptor area (SQLDA). Depending on the context in which it is used, the SQLDA stores information about prepared SQL statements or host variables. This information can then be read by either the application program or Db2.

#### Procedure

Code the SQLDA directly in the program, or use the following SQL INCLUDE statement to request a standard SQLDA declaration:

```sql
EXEC SQL INCLUDE SQLDA
```

**Restriction:** You must place SQLDA declarations before the first SQL statement that references the data descriptor, unless you use the TWOPASS SQL processing option.

#### Example

You can use host-variable arrays for certain multi-row operations in other host languages, such as C, C++, COBOL, and PL/I. However, the Db2 precompiler does not recognize declarations of host-variable arrays for assembler. However, you can SQLDA declarations to achieve similar results in assembler programs, as shown in the following examples:

- Assembler support for multiple-row FETCH is limited to the FETCH statement with the INTO DESCRIPTOR clause. For example:
• Assembler support for multiple-row INSERT is limited to the following cases:
  – Static multiple-row INSERT statement with scalar values (scalar host variables or scalar expressions) in the VALUES clause. For example:
    
    ```sql
    EXEC SQL INSERT INTO T1 VALUES (1, CURRENT DATE, 'TEST') FOR 10 ROWS
    ```
  
  – Dynamic multiple-row INSERT executed with the USING DESCRIPTOR clause on the EXECUTE statement. For example:
    
    ```assembly
    ATR       DS    CL20                         ATTRIBUTES FOR PREPARE
S1        DS    H,CL30                       VARCHAR STATEMENT STRING
MVC   ATR(20),=C'FOR MULTIPLE ROWS '
MVC   S1(2),=H'25'
MVC   S1+2(30),=C'INSERT INTO T1 VALUES (?) '
EXEC  SQL PREPARE STMT ATTRIBUTES :ATR FROM :S1
EXEC  SQL EXECUTE STMT USING DESCRIPTOR :SQLDA FOR 10 ROWS
    ```
    
    where the descriptor is set up correctly in advance according to the specifications for dynamic execution of a multiple-row INSERT statement with a descriptor

  – Assembler does not support multiple-row MERGE. You cannot specify MERGE statements that reference host-variable arrays.

Related tasks
Defining SQL descriptor areas (SQLDA)
If your program includes certain SQL statements, you must define at least one SQL descriptor area (SQLDA). Depending on the context in which it is used, the SQLDA stores information about prepared SQL statements or host variables. This information can then be read by either the application program or Db2.

Related reference
SQL descriptor area (SQLDA) (Db2 SQL)

Declaring host variables and indicator variables in assembler
You can use host variables, host-variable arrays, and host structures in SQL statements in your program to pass data between Db2 and your application.

Procedure
To declare host variables, host-variable arrays, and host structures:
1. Declare the variables according to the following rules and guidelines:
   • You can declare host variables in normal assembler style (DC or DS), depending on the data type and the limitations on that data type. You can specify a value on DC or DS declarations (for example, `DC H'5'`). The Db2 precompiler examines only packed decimal declarations.
   • If you specify the ONEPASS SQL processing option, you must explicitly declare each host variable and each host-variable array before using them in an SQL statement. If you specify the TWOPASS precompiler option, you must declare each host variable before using it in the DECLARE CURSOR statement.
   • If you specify the STDSQL(YES) SQL processing option, you must precede the host language statements that define the host variables and host-variable arrays with the BEGIN DECLARE SECTION statement and follow the host language statements with the END DECLARE SECTION statement. Otherwise, these statements are optional.
   • Ensure that any SQL statement that uses a host variable or host-variable array is within the scope of the statement that declares that variable or array.
   • If you are using the Db2 precompiler, ensure that the names of host variables and host-variable arrays are unique within the program, even if the variables and variable arrays are in different blocks,
classes, procedures, functions, or subroutines. You can qualify the names with a structure name to make them unique.

2. Optional: Define any associated indicator variables, arrays, and structures.

**Related tasks**
Declaring host variables and indicator variables
You can use host variables and indicator variables in SQL statements in your program to pass data between Db2 and your application.

**Host variables in assembler**
In assembler programs, you can specify numeric, character, graphic, binary, LOB, XML, and ROWID host variables. You can also specify result set, table, and LOB locators and LOB and XML file reference variables.

**Restrictions:**
- Only some of the valid assembler declarations are valid host variable declarations. If the declaration for a host variable is not valid, any SQL statement that references the variable might result in the message UNDECLARED HOST VARIABLE.
- The locator data types are assembler language data types and SQL data types. You cannot use locators as column types.

**Recommendations:**
- Be careful of overflow. For example, suppose that you retrieve an INTEGER column value into a DS H host variable, and the column value is larger than 32767. You get an overflow warning or an error, depending on whether you provide an indicator variable.
- Be careful of truncation. For example, if you retrieve an 80-character CHAR column value into a host variable that is declared as DS CL70, the rightmost ten characters of the retrieved string are truncated. If you retrieve a floating-point or decimal column value into a host variable declared as DS F, any fractional part of the value is removed.

**Numeric host variables**
The following diagram shows the syntax for declaring numeric host variables.
Notes:

1 value is a numeric value that specifies the scale of the packed decimal variable. If value does not include a decimal point, the scale is 0.

For floating-point data types (E, EH, EB, D, DH, and DB), use the FLOAT SQL processing option to specify whether the host variable is in IEEE binary floating-point or z/Architecture® hexadecimal floating-point format. If you specify FLOAT(S390), you need to define your floating-point host variables as E, EH, D, or DH. If you specify FLOAT(IEEE), you need to define your floating-point host variables as EB or DB. Db2 does not check if the host variable declarations or format of the host variable contents match the format that you specified with the FLOAT SQL processing option. Therefore, you need to ensure that your floating-point host variable types and contents match the format that you specified with the FLOAT SQL processing option. Db2 converts all floating-point input data to z/Architecture hexadecimal floating-point format before storing it.

Restriction: The FLOAT SQL processing options do not apply to the decimal floating-point host variable types ED, DD, or LD.

For the decimal floating-point host variable types ED, DD, and LD, you can specify the following special values: MIN, MAX, NAN, SNAN, and INFINITY.
**Character host variables**

You can specify the following forms of character host variables:

- Fixed-length strings
- Varying-length strings
- CLOBs

The following diagrams show the syntax for forms other than CLOBs.

The following diagram shows the syntax for declaring fixed-length character strings.

![Fixed-length character strings diagram]

Notes:

1. If you declare a character string host variable without a length (for example, `DC C ‘ABCD’`) Db2 interprets the length as 1. To get the correct length, specify a length attribute (for example, `DC CL 4 ‘ABCD’`).

The following diagram shows the syntax for declaring varying-length character strings.

![Varying-length character strings diagram]

**Graphic host variables**

You can specify the following forms of graphic host variables:

- Fixed-length strings
- Varying-length strings
- DBCLOBs

The following diagrams show the syntax for forms other than DBCLOBs. In the syntax diagrams, `value` denotes one or more DBCS characters, and the symbols `<` and `>` represent the shift-out and shift-in characters.

The following diagram shows the syntax for declaring fixed-length graphic strings.

![Fixed-length graphic strings diagram]

The following diagram shows the syntax for declaring varying-length graphic strings.

![Varying-length graphic strings diagram]

**Binary host variables**

The following diagram shows the syntax for declaring binary host variables.
**Varbinary host variables**

The following diagram shows the syntax for declaring varbinary host variables.

```plaintext
variable-name  —  DS  —  X  —  Ln  
```

Notes:

1. $1 \leq n \leq 255$

**Result set locators**

The following diagram shows the syntax for declaring result set locators.

```plaintext
variable-name  —  DS  —  H  —  L2  —  ,  —  X  —  Ln  
```

Notes:

1. $1 \leq n \leq 32704$

**Table Locators**

The following diagram shows the syntax for declaring of table locators.

```plaintext
variable-name  —  SQL TYPE IS  —  TABLE LIKE  —  table-name  —  AS LOCATOR  
```

**LOB variables, locators, and file reference variables**

The following diagram shows the syntax for declaring BLOB, CLOB, and DBCLOB host variables, locators, and file reference variables.
XML data host and file reference variables

The following diagram shows the syntax for declaring BLOB, CLOB, and DBCLOB host variables and file reference variables for XML data types.

Notes:

1 If you specify the length of the LOB in terms of KB, MB, or GB, do not leave spaces between the length and K, M, or G.

ROWIDs

The following diagram shows the syntax for declaring ROWID host variables.

Related concepts

Host variables
Use host variables to pass a single data item between Db2 and your application.

Using host variables in SQL statements
Use scalar host variables in embedded SQL statements to represent a single value. Host variables are useful for storing retrieved data or for passing values that are to be assigned or used for comparisons.

Related tasks
Determining whether a retrieved value in a host variable is null or truncated
Before your application manipulates the data that was retrieved from Db2 into a host variable, determine if the value is null. Also determine if it was truncated when assigned to the variable. You can use indicator variables to obtain this information.

Inserting a single row by using a host variable
Use host variables in your INSERT statement when you don’t know at least some of the values to insert until the program runs.

Inserting null values into columns by using indicator variables or arrays
If you need to insert null values into a column, using an indicator variable or array is an easy way to do so. An indicator variable or array is associated with a particular host variable or host-variable array.

Storing LOB data in a table
Db2 handles LOB data differently than it handles other kinds of data. As a result, in some cases, you need to take additional actions when you define LOB columns and insert the LOB data.

Retrieving a single row of data into host variables
If you know that your query returns only one row, you can specify one or more host variables to contain the column values of the retrieved row.

Updating data by using host variables
When you want to update a value in a Db2 table, but you do not know the exact value until the program runs, use host variables. Db2 can change a table value to match the current value of the host variable.

Related reference
Descriptions of SQL processing options
You can specify any SQL processing options regardless of whether you use the Db2 precompiler or the Db2 coprocessor. However, the Db2 coprocessor might ignore certain options because host language compiler options exist that provide the same information.

High Level Assembler (HLASM) and Toolkit Feature Library

Indicator variables in assembler
An indicator variable is a 2-byte integer (DS HL2). You declare indicator variables in the same way as host variables. You can mix the declarations of the two types of variables.

The following diagram shows the syntax for declaring an indicator variable in assembler.

```
variable-name
DC
DS
1
H
L2
```

Example
The following example shows a FETCH statement with the declarations of the host variables that are needed for the FETCH statement and their associated indicator variables.

```
EXEC SQL FETCH CLS_CURSOR INTO :CLSCD, X
    :DAY :DAYIND, X
    :BGN :BGNIND, X
    :END :ENDIND
```

You can declare these variables as follows:

```
CLSCD    DS CL7
DAY      DS HL2
```
Definition of indicator variables

Indicator variables are associated with particular host variables. Each indicator variable contains a small integer value that indicates some information about the associated host variable. Indicator arrays and structures serve the same purpose for host-variable arrays and structures.

Related tasks

Inserting null values into columns by using indicator variables or arrays

If you need to insert null values into a column, using an indicator variable or array is an easy way to do so. An indicator variable or array is associated with a particular host variable or host-variable array.

Equivalent SQL and assembler data types

When you declare host variables in your assembler programs, the precompiler uses equivalent SQL data types. When you retrieve data of a particular SQL data type into a host variable, ensure that the host variable is of an equivalent data type.

The following table describes the SQL data type and the base SQLTYPE and SQLLEN values that the precompiler uses for host variables in SQL statements.

<table>
<thead>
<tr>
<th>Assembler host variable data type</th>
<th>SQLTYPE of host variable</th>
<th>SQLLEN of host variable</th>
<th>SQL data type</th>
</tr>
</thead>
<tbody>
<tr>
<td>DS HL2</td>
<td>500</td>
<td>2</td>
<td>SMALLINT</td>
</tr>
<tr>
<td>DS FL4</td>
<td>496</td>
<td>4</td>
<td>INTEGER</td>
</tr>
<tr>
<td>DS 'value' or DS Pn'value' or DS Pln 1&lt;=n&lt;=16</td>
<td>484</td>
<td>p in byte 1, s in byte 2</td>
<td>DECIMAL(p,s)</td>
</tr>
<tr>
<td>short decimal FLOAT:</td>
<td>996</td>
<td>4</td>
<td>DECFLOAT</td>
</tr>
<tr>
<td>SDFP DC ED</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SDFP DC EDL4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SDFP DC EDL4'11.11'</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>long decimal FLOAT:</td>
<td>996</td>
<td>8</td>
<td>DECFLOAT</td>
</tr>
<tr>
<td>LDFP DC DD</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LDFP DC DDL8</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LDFP DC DDL8'22.22'</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>extended decimal FLOAT:</td>
<td>996</td>
<td>16</td>
<td>DECFLOAT</td>
</tr>
<tr>
<td>EDFP DC LD</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EDFP DC LDL16</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EDFP DC LDL16'33.33'</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DS EL4</td>
<td>480</td>
<td>4</td>
<td>REAL or FLOAT (n)</td>
</tr>
<tr>
<td>DS EHL4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DS EBL4</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Related concepts

Indicator variables, arrays, and structures

An indicator variable is associated with a particular host variable. Each indicator variable contains a small integer value that indicates some information about the associated host variable. Indicator arrays and structures serve the same purpose for host-variable arrays and structures.
Table 100. SQL data types, SQLLEN values, and SQLTYPE values that the precompiler uses for host variables in assembler programs (continued)

<table>
<thead>
<tr>
<th>Assembler host variable data type</th>
<th>SQLTYPE of host variable</th>
<th>SQLLEN of host variable</th>
<th>SQL data type</th>
</tr>
</thead>
<tbody>
<tr>
<td>DS DL8</td>
<td>480</td>
<td>8</td>
<td>DOUBLE PRECISION, or FLOAT (n) 22&lt;=n&lt;=53</td>
</tr>
<tr>
<td>DS DHL8</td>
<td>492</td>
<td>8</td>
<td>BIGINT</td>
</tr>
<tr>
<td>DS DBL8</td>
<td>492</td>
<td>8</td>
<td>BIGINT</td>
</tr>
<tr>
<td>DS FDL8</td>
<td>492</td>
<td>8</td>
<td>BIGINT (n)</td>
</tr>
<tr>
<td>SQL TYPE IS BINARY(n) 1&lt;=n&lt;=255</td>
<td>912</td>
<td>n</td>
<td>BINARY(n)</td>
</tr>
<tr>
<td>SQL TYPE IS VARBINARY(n) or SQL TYPE IS BINARY(n) VARYING 1&lt;=n&lt;=32764</td>
<td>908</td>
<td>n</td>
<td>VARBINARY(n)</td>
</tr>
<tr>
<td>DS CLn 1&lt;=n&lt;=255</td>
<td>452</td>
<td>n</td>
<td>CHAR(n)</td>
</tr>
<tr>
<td>DS HL2,CLn 1&lt;=n&lt;=255</td>
<td>448</td>
<td>n</td>
<td>VARCHAR(n)</td>
</tr>
<tr>
<td>DS HL2,CLn n&gt;255</td>
<td>456</td>
<td>n</td>
<td>VARCHAR(n)</td>
</tr>
<tr>
<td>DS GLm 2&lt;=m&lt;=254</td>
<td>468</td>
<td>n</td>
<td>GRAPHIC(n)</td>
</tr>
<tr>
<td>DS HL2,GLm 2&lt;=m&lt;=254</td>
<td>464</td>
<td>n</td>
<td>VARGRAPHIC(n)</td>
</tr>
<tr>
<td>DS HL2,GLm m&gt;254</td>
<td>472</td>
<td>n</td>
<td>VARGRAPHIC(n)</td>
</tr>
<tr>
<td>SQL TYPE IS RESULT_SET_LOCATOR</td>
<td>972</td>
<td>4</td>
<td>Result set locator4,5</td>
</tr>
<tr>
<td>SQL TYPE IS TABLE LIKE table-name AS LOCATOR</td>
<td>976</td>
<td>4</td>
<td>Table locator4</td>
</tr>
<tr>
<td>SQL TYPE IS BLOB_LOCATOR</td>
<td>960</td>
<td>4</td>
<td>BLOB locator4</td>
</tr>
<tr>
<td>SQL TYPE IS CLOB_LOCATOR</td>
<td>964</td>
<td>4</td>
<td>CLOB locator4</td>
</tr>
</tbody>
</table>
Table 100. SQL data types, SQLLEN values, and SQLTYPE values that the precompiler uses for host variables in assembler programs (continued)

<table>
<thead>
<tr>
<th>Assembler host variable data type</th>
<th>SQLTYPE of host variable&lt;sup&gt;1&lt;/sup&gt;</th>
<th>SQLLEN of host variable</th>
<th>SQL data type</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQL TYPE IS DBCLOB_LOCATOR</td>
<td>968</td>
<td>4</td>
<td>DBCLOB locator</td>
</tr>
<tr>
<td>SQL TYPE IS BLOB(n) 1s1ns2147483647</td>
<td>404</td>
<td>n</td>
<td>BLOB(n)</td>
</tr>
<tr>
<td>SQL TYPE IS CLOB(n) 1s1ns2147483647</td>
<td>408</td>
<td>n</td>
<td>CLOB(n)</td>
</tr>
<tr>
<td>SQL TYPE IS DBCLOB(n) 1s1ns1073741823</td>
<td>412</td>
<td>n</td>
<td>DBCLOB(n)</td>
</tr>
<tr>
<td>SQL TYPE IS XML AS BLOB(n)</td>
<td>404</td>
<td>0</td>
<td>XML</td>
</tr>
<tr>
<td>SQL TYPE IS XML AS CLOB(n)</td>
<td>408</td>
<td>0</td>
<td>XML</td>
</tr>
<tr>
<td>SQL TYPE IS XML AS DBCLOB(n)</td>
<td>412</td>
<td>0</td>
<td>XML</td>
</tr>
<tr>
<td>SQL TYPE IS BLOB_FILE</td>
<td>916/917</td>
<td>267</td>
<td>BLOB file reference</td>
</tr>
<tr>
<td>SQL TYPE IS CLOB_FILE</td>
<td>920/921</td>
<td>267</td>
<td>CLOB file reference</td>
</tr>
<tr>
<td>SQL TYPE IS DBCLOB_FILE</td>
<td>924/925</td>
<td>267</td>
<td>DBCLOB file reference</td>
</tr>
<tr>
<td>SQL TYPE IS XML AS BLOB_FILE</td>
<td>916/917</td>
<td>267</td>
<td>XML BLOB file reference</td>
</tr>
<tr>
<td>SQL TYPE IS XML AS CLOB_FILE</td>
<td>920/921</td>
<td>267</td>
<td>XML CLOB file reference</td>
</tr>
<tr>
<td>SQL TYPE IS XML AS DBCLOB_FILE</td>
<td>924/925</td>
<td>267</td>
<td>XML DBCLOB file reference</td>
</tr>
<tr>
<td>SQL TYPE IS ROWID</td>
<td>904</td>
<td>40</td>
<td>ROWID</td>
</tr>
</tbody>
</table>

Notes:
1. If a host variable includes an indicator variable, the SQLTYPE value is the base SQLTYPE value plus 1.
2. \( m \) is the number of bytes.
3. \( n \) is the number of double-byte characters.
4. This data type cannot be used as a column type.
5. To be compatible with previous releases, result set locator host variables may be declared as fullword integers (FL4), but the method shown is the preferred syntax.
The following table shows equivalent assembler host variables for each SQL data type. Use this table to determine the assembler data type for host variables that you define to receive output from the database. For example, if you retrieve TIMESTAMP data, you can define variable DS CLn.

This table shows direct conversions between SQL data types and assembler data types. However, a number of SQL data types are compatible. When you do assignments or comparisons of data that have compatible data types, Db2 converts those compatible data types.

<table>
<thead>
<tr>
<th>SQL data type</th>
<th>Assembler host variable equivalent</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>SMALLINT</td>
<td>DS HL2</td>
<td></td>
</tr>
<tr>
<td>INTEGER</td>
<td>DS F</td>
<td></td>
</tr>
<tr>
<td>BIGINT</td>
<td>DS FD OR DS FDL8</td>
<td>DS FDL8 requires High Level Assembler (HLASM), Release 4 or later.</td>
</tr>
<tr>
<td>DECIMAL(p,s) or NUMERIC(p,s)</td>
<td>DS P'value' DS PLn'value' DS PLn</td>
<td>p is precision; s is scale. 1&lt;=p&lt;=31 and 0&lt;=s&lt;=p. 1&lt;=n&lt;=16. value is a literal value that includes a decimal point. You must use Ln, value, or both. Using only value is recommended. Precision: If you use Ln, it is 2n-1; otherwise, it is the number of digits in value. Scale: If you use value, it is the number of digits to the right of the decimal point; otherwise, it is 0. For efficient use of indexes: Use value. If p is even, do not use Ln and be sure the precision of value is p and the scale of value is s. If p is odd, you can use Ln (although it is not advised), but you must choose n so that 2n-1=p, and value so that the scale is s. Include a decimal point in value, even when the scale of value is 0.</td>
</tr>
<tr>
<td>REAL or FLOAT(n)</td>
<td>DS EL4 DS EHL4 DS EBL4(^1)</td>
<td>1&lt;=n&lt;=21</td>
</tr>
<tr>
<td>DOUBLE PRECISION, DOUBLE, or FLOAT(n)</td>
<td>DS DL8 DS DHL8 DS DBL8(^1)</td>
<td>22&lt;=n&lt;=53</td>
</tr>
<tr>
<td>DECFLOAT</td>
<td>DC EDL4 DC DDL8 DC LDL16</td>
<td></td>
</tr>
<tr>
<td>CHAR(n)</td>
<td>DS CLn</td>
<td>1&lt;=n&lt;=255</td>
</tr>
<tr>
<td>VARCHAR(n)</td>
<td>DS HL2,CLn</td>
<td></td>
</tr>
<tr>
<td>GRAPHIC(n)</td>
<td>DS GLm</td>
<td>m is expressed in bytes. n is the number of double-byte characters. 1&lt;=n&lt;=127</td>
</tr>
<tr>
<td>VARGRAPHIC(n)</td>
<td>DS HL2,GLx DS HL2'm',GLx'value'</td>
<td>x and m are expressed in bytes. n is the number of double-byte characters. &lt; and &gt; represent shift-out and shift-in characters.</td>
</tr>
<tr>
<td>BINARY(n)</td>
<td><strong>Format 1:</strong> variable-name--DS--X--Ln</td>
<td>1&lt;=n&lt;=255</td>
</tr>
<tr>
<td></td>
<td><strong>Format 2:</strong> SQL TYPE IS BINARY(n)</td>
<td></td>
</tr>
<tr>
<td>SQL data type</td>
<td>Assembler host variable equivalent</td>
<td>Notes</td>
</tr>
<tr>
<td>------------------------</td>
<td>-------------------------------------</td>
<td>----------------------------------------------------------------------</td>
</tr>
<tr>
<td>VARBINARY(n)</td>
<td><strong>Format 1:</strong> variable-name--</td>
<td>1&lt;=n&lt;=32704</td>
</tr>
<tr>
<td></td>
<td>DS--H--L2--,--X--Ln</td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>Format 2:</strong> SQL TYPE IS</td>
<td></td>
</tr>
<tr>
<td></td>
<td>VARBINARY(n) or SQL TYPE IS BINARY(n)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>VARYING</td>
<td></td>
</tr>
<tr>
<td>DATE</td>
<td>DS CLn</td>
<td>If you are using a date exit routine, n is determined by that routine; otherwise, n must be at least 10.</td>
</tr>
<tr>
<td>TIME</td>
<td>DS CLn</td>
<td>If you are using a time exit routine, n is determined by that routine. Otherwise, n must be at least 6; to include seconds, n must be at least 8.</td>
</tr>
<tr>
<td>TIMESTAMP</td>
<td>DS CLn</td>
<td>n must be at least 19. To include microseconds, n must be 26; if n is less than 26, truncation occurs on the microseconds part.</td>
</tr>
<tr>
<td>TIMESTAMP(0)</td>
<td>DS CLn</td>
<td>n must be at least 19.</td>
</tr>
<tr>
<td>TIMESTAMP(p) p &gt; 0</td>
<td>DS CLn</td>
<td>n must be at least 19. To include fractional seconds, n must be 20+x where x is the number of fractional seconds to include; if x is less than p, truncation occurs on the fractional seconds part.</td>
</tr>
<tr>
<td>TIMESTAMP(0) WITH TIME ZONE</td>
<td>DS HL2,CLn</td>
<td>n must be at least 25.</td>
</tr>
<tr>
<td>TIMESTAMP(p) WITH TIME ZONE p &gt; 0</td>
<td>DS HL2,CLn</td>
<td>n must be at least 26+p.</td>
</tr>
<tr>
<td>Result set locator</td>
<td>DS F</td>
<td>Use this data type only to receive result sets. Do not use this data type as a column type.</td>
</tr>
<tr>
<td>Table locator</td>
<td>SQL TYPE IS TABLE LIKE</td>
<td>Use this data type only in a user-defined function or stored procedure to receive rows of a transition table. Do not use this data type as a column type.</td>
</tr>
<tr>
<td></td>
<td>table-name AS LOCATOR</td>
<td></td>
</tr>
<tr>
<td>BLOB locator</td>
<td>SQL TYPE IS BLOB_LOCATOR</td>
<td>Use this data type only to manipulate data in BLOB columns. Do not use this data type as a column type.</td>
</tr>
<tr>
<td>CLOB locator</td>
<td>SQL TYPE IS CLOB_LOCATOR</td>
<td>Use this data type only to manipulate data in CLOB columns. Do not use this data type as a column type.</td>
</tr>
<tr>
<td>DBCLOB locator</td>
<td>SQL TYPE IS DBCLOB_LOCATOR</td>
<td>Use this data type only to manipulate data in DBCLOB columns. Do not use this data type as a column type.</td>
</tr>
<tr>
<td>SQL data type</td>
<td>Assembler host variable equivalent</td>
<td>Notes</td>
</tr>
<tr>
<td>------------------</td>
<td>-------------------------------------</td>
<td>------------------------------------------------</td>
</tr>
<tr>
<td>BLOB($n$)</td>
<td>SQL TYPE IS BLOB($n$)</td>
<td>$1\leq n \leq 2147483647$</td>
</tr>
<tr>
<td>CLOB($n$)</td>
<td>SQL TYPE IS CLOB($n$)</td>
<td>$1\leq n \leq 2147483647$</td>
</tr>
<tr>
<td>DBCLOB($n$)</td>
<td>SQL TYPE IS DBCLOB($n$)</td>
<td>$n$ is the number of double-byte characters. $1\leq n \leq 1073741823$</td>
</tr>
<tr>
<td>XML</td>
<td>SQL TYPE IS XML AS BLOB($n$)</td>
<td>$1\leq n \leq 2147483647$</td>
</tr>
<tr>
<td>XML</td>
<td>SQL TYPE IS XML AS CLOB($n$)</td>
<td>$1\leq n \leq 2147483647$</td>
</tr>
<tr>
<td>XML</td>
<td>SQL TYPE IS XML AS DBCLOB($n$)</td>
<td>$n$ is the number of double-byte characters. $1\leq n \leq 1073741823$</td>
</tr>
<tr>
<td>BLOB file reference</td>
<td>SQL TYPE IS BLOB_FILE</td>
<td>Use this data type only to manipulate data in BLOB columns. Do not use this data type as a column type.</td>
</tr>
<tr>
<td>CLOB file reference</td>
<td>SQL TYPE IS CLOB_FILE</td>
<td>Use this data type only to manipulate data in CLOB columns. Do not use this data type as a column type.</td>
</tr>
<tr>
<td>DBCLOB file reference</td>
<td>SQL TYPE IS DBCLOB_FILE</td>
<td>Use this data type only to manipulate data in DBCLOB columns. Do not use this data type as a column type.</td>
</tr>
<tr>
<td>XML BLOB file reference</td>
<td>SQL TYPE IS XML AS BLOB_FILE</td>
<td>Use this data type only to manipulate XML data as BLOB files. Do not use this data type as a column type.</td>
</tr>
<tr>
<td>XML CLOB file reference</td>
<td>SQL TYPE IS XML AS CLOB_FILE</td>
<td>Use this data type only to manipulate XML data as CLOB files. Do not use this data type as a column type.</td>
</tr>
<tr>
<td>XML DBCLOB file reference</td>
<td>SQL TYPE IS XML AS DBCLOB_FILE</td>
<td>Use this data type only to manipulate XML data as DBCLOB files. Do not use this data type as a column type.</td>
</tr>
<tr>
<td>ROWID</td>
<td>SQL TYPE IS ROWID</td>
<td></td>
</tr>
</tbody>
</table>

**Notes:**
1. Although stored procedures and user-defined functions can use IEEE floating-point host variables, you cannot declare a user-defined function or stored procedure parameter as IEEE.

**Related concepts**
Compatibility of SQL and language data types
The host variable data types that are used in SQL statements must be compatible with the data types of the columns with which you intend to use them.

LOB host variable, LOB locator, and LOB file reference variable declarations
When you write applications to manipulate LOB data, you need to declare host variables to hold the LOB data or LOB locator. Alternatively, you need to declare LOB file reference variables to point to the LOB data.

Host variable data types for XML data in embedded SQL applications (Db2 Programming for XML)
Macros for assembler applications

Data set DSN1210.SDSNMACS contains all Db2 macros that are available for use.

Handling SQL error codes in assembler applications

Assembler applications can request more information about SQL error codes by using the DSNTIAR subroutine or issuing a GET DIAGNOSTICS statement.

Procedure

To request information about SQL errors in assembler programs, use the following approaches:

- You can use the subroutine DSNTIAR to convert an SQL return code into a text message.

  DSNTIAR takes data from the SQLCA, formats it into a message, and places the result in a message output area that you provide in your application program. For concepts and more information about the behavior of DSNTIAR, see “Displaying SQLCA fields by calling DSNTIAR” on page 529.

  **DSNTIAR syntax**

  DSNTIAR has the following syntax:

  ```
  CALL DSNTIAR, (sqlca, message, lrecl), MF=(E, PARM)
  ```

  **DSNTIAR parameters**

  The DSNTIAR parameters have the following meanings:

  - **sqlca**
    
    An SQL communication area.

  - **message**
    
    An output area, defined as a varying-length string, in which DSNTIAR places the message text. The first halfword contains the length of the remaining area; its minimum value is 240.

    The output lines of text, each line being the length specified in **lrecl**, are put into this area. For example, you could specify the format of the output area as:

    ```
    LINES    EQU   10
    LRECL    EQU   132
    ⋮
    MSGLRECL DC     AL4(LRECL)
    MESSAGE  DS    H, CL(LINES*LRECL)
             ORG MESSAGE
    MESSAGE1 DS    CL(LRECL)      text line 1
    MESSAGE2 DS    CL(LRECL)      text line 2
    ⋮
    MESSAGEn DS    CL(LRECL)      text line n
    ⋮
    CALL DSNTIAR, (SQLCA, MESSAGE, MSGLRECL), MF=(E, PARM)
    ```

    where MESSAGE is the name of the message output area, LINES is the number of lines in the message output area, and LRECL is the length of each line.

  - **lrecl**
    
    A fullword containing the logical record length of output messages, between 72 and 240.

    The expression **MF=(E, PARM)** is an z/OS macro parameter that indicates dynamic execution. **PARM** is the name of a data area that contains a list of pointers to the call parameters of DSNTIAR.

    See “Sample applications supplied with Db2 for z/OS” on page 1024 for instructions on how to access and print the source code for the sample program.

- If your CICS application requires CICS storage handling, you must use the subroutine DSNTIAC instead of DSNTIAR.

  **DSNTIAC syntax**

  DSNTIAC has the following syntax:
DSNTIAC parameters

DSNTIAC has extra parameters, which you must use for calls to routines that use CICS commands.

**eib**
EXEC interface block

**commarea**
communication area

For more information on these parameters, see the appropriate application programming guide for CICS. The remaining parameter descriptions are the same as those for DSNTIAR. Both DSNTIAC and DSNTIAR format the SQLCA in the same way.

You must define DSNTIA1 in the CSD. If you load DSNTIAR or DSNTIAC, you must also define them in the CSD. For an example of CSD entry generation statements for use with DSNTIAC, see member DSN8FRDO in the data set `prefix.SDSNSAMP`.

The assembler source code for DSNTIAC and job DSNTEJ5A, which assembles and link-edits DSNTIAC, are also in the data set `prefix.SDSNSAMP`.

- You can also use the MESSAGE_TEXT condition item field of the GET DIAGNOSTICS statement to convert an SQL return code into a text message.

Programs that require long token message support should code the GET DIAGNOSTICS statement instead of DSNTIAR. For more information about GET DIAGNOSTICS, see “Checking the execution of SQL statements by using the GET DIAGNOSTICS statement” on page 534.

Related tasks
- Handling SQL error codes
  Application programs can request more information about SQL error codes from Db2.

Related reference
- GET DIAGNOSTICS (Db2 SQL)

C and C++ applications that issue SQL statements

You can code SQL statements in a C or C++ program wherever you can use executable statements.

Each SQL statement in a C or C++ program must begin with EXEC SQL and end with a semicolon (;). The EXEC and SQL keywords must appear on one line, but the remainder of the statement can appear on subsequent lines.

In general, because C is case sensitive, use uppercase letters to enter all SQL keywords. However, if you use the FOLD precompiler suboption, Db2 folds lowercase letters in SBCS SQL ordinary identifiers to uppercase. For information about host language precompiler options, see Table 139 on page 852.

You must keep the case of host variable names consistent throughout the program. For example, if a host variable name is lowercase in its declaration, it must be lowercase in all SQL statements. You might code an UPDATE statement in a C program as follows:

```sql
EXEC SQL
UPDATE DSN8C10.DEPT
SET MGRNO = :mgr_num
WHERE DEPTNO = :int_dept;
```

Comments
- You can include C comments (/* ... */) within SQL statements wherever you can use a blank, except between the keywords EXEC and SQL. You can use single-line comments (starting with //) in C language statements, but not in embedded SQL. You can use SQL comments within embedded SQL statements. For more information, see SQL comments (Db2 SQL).

You can nest comments.
To include EBCDIC DBCS characters in comments, you must delimit the characters by a shift-out and shift-in control character; the first shift-in character in the DBCS string signals the end of the DBCS string.

**Continuation for SQL statements**
You can use a backslash to continue a character-string constant or delimited identifier on the following line. However, EBCDIC DBCS string constants cannot be continued on a second line.

**Delimiters**
Delimit an SQL statement in your C program with the beginning keyword EXEC SQL and a Semicolon (;).

**Declaring tables and views**
Your C program should use the DECLARE TABLE statement to describe each table and view the program accesses. You can use the Db2 declarations generator (DCLGEN) to generate the DECLARE TABLE statements. For more information, see “DCLGEN (declarations generator)” on page 464.

**Including SQL statements and variable declarations in source code that is to be processed by the Db2 precompiler**
To include SQL statements or C host variable declarations from a member of a partitioned data set, add the following SQL statement to the source code where you want to include the statements:

```sql
EXEC SQL INCLUDE member-name;
```

You cannot nest SQL INCLUDE statements. Do not use C #include statements to include SQL statements or C host variable declarations.

**Margins**
Code SQL statements in columns 1 through 72, unless you specify other margins to the Db2 precompiler. If EXEC SQL is not within the specified margins, the Db2 precompiler does not recognize the SQL statement. The margin rules do not apply to the Db2 coprocessor. The Db2 coprocessor allows variable length source input.

**Names**
You can use any valid C name for a host variable, subject to the following restrictions:

- Do not use DBCS characters.
- Do not use external entry names or access plan names that begin with 'DSN', and do not use host variable names or macro names that begin with 'SQL' (in any combination of uppercase or lowercase letters). These names are reserved for Db2.

An SQL identifier that starts with the pound character ("#") can be interpreted as a C macro statement.

**Nulls and NULs**
C and SQL differ in the way they use the word null. The C language has a null character (NUL), a null pointer (NULL), and a null statement (just a semicolon). The C NUL is a single character that compares equal to 0. The C NULL is a special reserved pointer value that does not point to any valid data object. The SQL null value is a special value that is distinct from all non-null values and denotes the absence of a (nonnull) value. NUL (or NUL-terminator) is the null character in C and C++, and NULL is the SQL null value.

**Sequence numbers**
The Db2 precompiler generates statements without sequence numbers. (The Db2 coprocessor does not perform this action, because the source is read and modified by the compiler.)

**Statement labels**
You can precede SQL statements with a label.

**Trigraph characters**
Some characters from the C character set are not available on all keyboards. You can enter these characters into a C source program using a sequence of three characters called a trigraph. The trigraph characters that Db2 supports are the same as those that the C compiler supports.
WHENEVER statement
The target for the GOTO clause in an SQL WHENEVER statement must be within the scope of any SQL statements that the statement WHENEVER affects.

Special C/C++ considerations
• Using the C/370 multi-tasking facility, in which multiple tasks execute SQL statements, causes unpredictable results.
• Except for the Db2 coprocessor, you must run the Db2 precompiler before running the C preprocessor.
• Except for the Db2 coprocessor, Db2 precompiler does not support C preprocessor directives.
• If you use conditional compiler directives that contain C code, either place them after the first C token in your application program, or include them in the C program using the #include preprocessor directive.

Refer to the appropriate C documentation for more information about C preprocessor directives.

To use the decimal floating-point host data type, you must do the following:
• Use z/OS 1.10 or above (z/OS V1R10 XL C/C++).
• Compile with the C/C++ compiler option, DFP.
• Specify the SQL compiler option to enable the Db2 coprocessor.
• Specify C/C++ compiler option, ARCH(7). It is required by the DFP compiler option if the DFP type is used in the source.
• Specify 'DEFINE(__STDC_WANT_DEC_FP__)' compiler option.

Handling SQL error codes
C and C++ applications can request more information about SQL errors from Db2. For more information, see "Handling SQL error codes in C and C++ applications" on page 617.

Related concepts
Using host-variable arrays in SQL statements
Use host-variable arrays in embedded SQL statements to represent values that the program does not know until the query is executed. Host-variable arrays are useful for storing a set of retrieved values or for passing a set of values that are to be inserted into a table.

Related tasks
Overview of programming applications that access Db2 for z/OS data
Applications that interact with Db2 must first connect to Db2. They can then read, add, or modify data or manipulate Db2 objects.

Including dynamic SQL in your program
Dynamic SQL is prepared and executed while the program is running.

Handling SQL error codes
Application programs can request more information about SQL error codes from Db2.

Setting limits for system resource usage by using the resource limit facility (Db2 Performance)

C and C++ programming examples
You can write Db2 programs in C and C++. These programs can access a local or remote Db2 subsystem and can execute static or dynamic SQL statements. This information contains several such programming examples.

To prepare and run these applications, start with the JCL in member DSNTEJ2D of data set prefix.SDSSAMP as a model for your JCL.

Related concepts
Job DSNTEJ2D (Db2 Installation and Migration)
Related reference
Assembler, C, C++, COBOL, PL/I, and REXX programming examples (Db2 Programming samples)
Sample dynamic and static SQL in a C program

Programs that access Db2 can contain static SQL, dynamic SQL, or both.

This example shows a C program that contains both static and dynamic SQL.

The following figure illustrates dynamic SQL and static SQL embedded in a C program. Each section of the program is identified with a comment. Section 1 of the program shows static SQL; sections 2, 3, and 4 show dynamic SQL. The function of each section is explained in detail in the prologue to the program.

```c
#include "stdio.h"
#include "stdefs.h"
EXEC SQL INCLUDE SQLCA;
EXEC SQL INCLUDE SQLDA;
EXEC SQL BEGIN DECLARE SECTION;
short edlevel;
struct { short len;  
  char x1[56]; 
} stmtbf1, stmtbf2, inpstr;
EXEC SQL END DECLARE SECTION;

/* Descriptive name = Dynamic SQL sample using C language */
/* Function = To show examples of the use of dynamic and static SQL. */
/* Notes = This example assumes that the EMP and DEPT tables are defined. They need not be the same as the DB2 Sample tables. */
/* Module type = C program */
/* Processor = DB2 precompiler, C compiler */
/* Module size = see link edit */
/* Attributes = not reentrant or reusable */
/* Input = symbolic label/name = DEPT 
  description = arbitrary table */
/* symbolic label/name = EMP 
  description = arbitrary table */
/* Output = symbolic label/name = SYSPRINT 
  description = print results via printf */
/* Exit-normal = return code 0 normal completion */
/* Exit-error = */
/* Return code = SQLCA */
/* Abend codes = none */
/* External references = none */
/* Control-blocks = SQLCA - sql communication area */
/* Logic specification: */
/* There are four SQL sections. */
/* 1) STATIC SQL 1: using static cursor with a SELECT statement. */
/* Two output host variables. */
/* 2) Dynamic SQL 2: Fixed-list SELECT, using same SELECT statement used in SQL 1 to show the difference. The prepared string :iptstr can be assigned with other dynamic-able SQL statements. */
/* 3) Dynamic SQL 3: Insert with parameter markers. */
/* Using four parameter markers which represent four input host variables within a host structure. */
/* 4) Dynamic SQL 4: EXECUTE IMMEDIATE */
/* A GRANT statement is executed immediately by passing it to DB2 via a varying string host variable. The example shows how to set up the host variable before passing it. */

#include "stdio.h"
#include "stdefs.h"
EXEC SQL INCLUDE SQLCA;
EXEC SQL INCLUDE SQLDA;
EXEC SQL BEGIN DECLARE SECTION;
short edlevel;
struct { short len;  
  char x1??(56??);
} stmtbf1, stmtbf2, inpstr;
EXEC SQL END DECLARE SECTION;
```
char *x1??(15??);
} lname;
short hv1;
struct { char  deptno??(4??);
struct { short len;
char  x??(36??);
} deptname;
char  mgno??(7??);
char  admrdept??(4??);
char  location??(17??);
} hv2;
short ind??(4??);
EXEC SQL END DECLARE SECTION;
EXEC SQL DECLARE EMP TABLE
(EMPNO           CHAR(6)             ,
FIRSTNAME       VARCHAR(12)         ,
MIDINIT         CHAR(1)             ,
LASTNAME        VARCHAR(15)         ,
WORKDEPT        CHAR(3)             ,
PHONENO         CHAR(4)             ,
HIREDATE        DECIMAL(6)          ,
JOBCODE         DECIMAL(3)          ,
EDLEVEL         SMALLINT            ,
SEX             CHAR(1)             ,
BIRTHDATE       DECIMAL(6)          ,
SALARY          DECIMAL(8,2)        ,
FORFNAME        VARGRAPHIC(12)      ,
FORMNAME        GRAPHIC(1)          ,
FORLNAME        VARGRAPHIC(15)      ,
FORADDR         VARGRAPHIC(256) )   ;
EXEC SQL DECLARE DEPT TABLE
( DEPTNO          CHAR(3)             ,
DEPTNAME        VARCHAR(36)         ,
MGRNO           CHAR(6)             ,
ADMRDEPT        CHAR(3)             ,
LOCATION        CHAR(16));
main ()
{
printf("\n***      begin of program                        ***");
EXEC SQL WHENEVER SQLERROR GO TO HANDLEERR;
EXEC SQL WHENEVER SQLWARNING GO TO HANDWARN;
EXEC SQL WHENEVER NOT FOUND GO TO NOTFOUND;
/******************************************************************/
/* Assign values to host variables which will be input to DB2      */
/******************************************************************/
strcpy(hv2.deptno,"M92");
strcpy(hv2.deptname.x,"DDL");
hv2.deptname.len = strlen(hv2.deptname.x);
strcpy(hv2.mgrno,"000010");
strcpy(hv2.admrdept,"A00");
/******************************************************************/
/* Static  SQL 1:  DECLARE CURSOR, OPEN, FETCH, CLOSE             */
/******************************************************************/
printf("\n***      begin declare                           ***");
EXEC SQL DECLARE C1 CURSOR FOR SELECT EDLEVEL, LASTNAME FROM EMP
WHERE EMPNO = '000010';
printf("\nn***      begin open                              ***");
EXEC SQL OPEN C1;
printf("\nn***      begin fetch                             ***");
EXEC SQL FETCH C1 INTO :edlevel, :lname;
printf("\nn***      begin close                             ***");
EXEC SQL CLOSE C1;
EXEC SQL DECLARE C1 CURSOR FOR SELECT EDLEVEL, LASTNAME FROM EMP
WHERE EMPNO = '000010';
printf("\nn***      begin prepare                           ***");
EXEC SQL PREPARE STAT1 FROM :inpstr;
EXEC SQL DECLARE C2 CURSOR FOR STAT1;
EXEC SQL OPEN C2;
Example C program that calls a stored procedure

You can call the C language version of the GETPRML stored procedure that uses the GENERAL WITH NULLS linkage convention.

Because the stored procedure returns result sets, this program checks for result sets and retrieves the contents of the result sets. The following figure contains the example C program that calls the GETPRML stored procedure.

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

main()
{
  EXEC SQL INCLUDE SQLCA;
  EXEC SQL INCLUDE SQLDA;
  short int i;                /* Loop counter                 */
  EXEC SQL FETCH C2 INTO :edlevel, :lname;
  printf("??/n\***      begin fetch                             ***");
  printf("??/n\*** returned values                              ***");
  printf("??/nedlevel = %d",edlevel);
  printf("??/n\***      begin close                             ***");
  EXEC SQL CLOSE C2;
  EXEC SQL PREPARE s1 FROM :stmtbf1;
  EXEC SQL EXECUTE IMMEDIATE :stmtbf2;
  printf("??/n\***      begin execute                           ***");
  EXEC SQL EXECUTE IMMEDIATE :stmtbf3;
  printf("??/n\***      begin prepare                           ***");
  EXEC SQL COMMIT;
  EXEC SQL EXECUTE IMMEDIATE :stmtbf4;
  printf("??/n\***      end of program                          ***");
}
```
/ * - Parameters used to call stored procedure GETPRML */ /* - An SQLDA for DESCRIBE PROCEDURE */ /* - An SQLDA for DESCRIBE CURSOR */ /* - Result set variable locators for up to three result */ /* - sets */ /****************************************************************************/ EXEC SQL BEGIN DECLARE SECTION; char procnm[19]; /* INPUT parm -- PROCEDURE name */ char schema[9]; /* INPUT parm -- User's schema */ long int out_code; /* OUTPUT -- SQLCODE from the */ /* SELECT operation. */ char parmlst[255]; /* OUTPUT -- RUNOPTS values */ /* for the matching row in */ /* catalog table SYSROUTINES */ struct indicators { short int procnm_ind; /* Indicator variable structure */ short int schema_ind; short int out_code_ind; short int parmlst_ind; } parmind; struct sqlda *proc_da; /* SQLDA for DESCRIBE PROCEDURE */ struct sqlda *res_da; /* SQLDA for DESCRIBE CURSOR */ static volatile SQL TYPE IS RESULT_SET_LOCATOR *loc1, *loc2, *loc3; /* Locator variables */ EXEC SQL END DECLARE SECTION; /****************************************************************************/ /* Allocate the SQLDAs to be used for DESCRIBE */ /* PROCEDURE and DESCRIBE CURSOR. Assume that at most */ /* three cursors are returned and that each result set */ /* has no more than five columns. */ /****************************************************************************/ proc_da = (struct sqlda *)malloc(SQLDASIZE(3)); res_da = (struct sqlda *)malloc(SQLDASIZE(5)); /****************************************************************************/ /* Call the GETPRML stored procedure to retrieve the */ /* RUNOPTS values for the stored procedure. In this */ /* example, we request the PARMLIST definition for the */ /* stored procedure named DSN8EP2. */ /* */ /* The call should complete with SQLCODE +466 because */ /* GETPRML returns result sets. */ /****************************************************************************/ strcpy(procnm,”dsn8ep2 “); /* Input parameter -- PROCEDURE to be found */ strcpy(schema,” “); /* Input parameter -- Schema name for proc */ parmind.procnm_ind=0; parmind.schema_ind=0; parmind.out_code_ind=0; /* Indicate that none of the input parameters */ /* have null values */ parmind.parmlst_ind=1; /* The parmlst parameter is an output parm. */ /* Mark PARMLIST parameter as null, so the DB2 */ /* requester does not have to send the entire */ /* PARMLIST variable to the server. This */ /* helps reduce network I/O time, because */ /* PARMLIST is fairly large. */ EXEC SQL CALL GETPRML(procnm INDICATOR :parmind.procnm_ind, :schema INDICATOR :parmind.schema_ind, :out_code INDICATOR :parmind.out_code_ind, :parmlst INDICATOR :parmind.parmlst_ind); if(SQLCODE!=+466) /* If SQL CALL failed, */ { printf("SQL CALL failed due to SQLCODE = %d\n", sqlca.sqlcode); printf("sqlca.sqlerrmc = "); for(i=0;i<sqlca.sqlerrml;i++) printf("%c",sqlca.sqlerrmc[i]); */ /* print the SQLCODE and any */ /* message tokens */ */ /* If SQL CALL failed, */ }}
Example C stored procedure with a GENERAL linkage convention

You can call a stored procedure that uses the GENERAL linkage convention from a C program.

This example stored procedure does the following:
• Searches the Db2 catalog table SYSROUTINES for a row that matches the input parameters from the client program. The two input parameters contain values for NAME and SCHEMA.
• Searches the Db2 catalog table SYSTABLES for all tables in which the value of CREATOR matches the value of input parameter SCHEMA. The stored procedure uses a cursor to return the table names.

The linkage convention used for this stored procedure is GENERAL.

The output parameters from this stored procedure contain the SQLCODE from the SELECT statement and the value of the RUNOPTS column from SYSROUTINES.

The CREATE PROCEDURE statement for this stored procedure might look like this:

```
CREATE PROCEDURE GETPRML(PROCNM CHAR(18) IN, SCHEMA CHAR(8) IN,
OUTCODE INTEGER OUT, PARMLST VARCHAR(254) OUT)
LANGUAGE C
DETERMINISTIC
READS SQL DATA
EXTERNAL NAME "GETPRML"
COLLID GETPRML
ASUTIME NO LIMIT
PARAMETER STYLE GENERAL
STAY RESIDENT NO
RUN OPTIONS "MSGFILE(OUTFILE),RPTSTG(ON),RPTOPTS(ON)"
WLM ENVIRONMENT SAMPPROG
PROGRAM TYPE MAIN
SECURITY DB2
RESULT SETS 2
COMMIT ON RETURN NO;
```

The following example is a C stored procedure with linkage convention GENERAL

```c
#include <stdlib.h>
EXEC SQL INCLUDE SQLCA;

/**********************************************************/
/* Declare C variables for SQL operations on the parameters. */
/* These are local variables to the C program, which you must */
/* copy to and from the parameter list provided to the stored */
/* procedure.                                               */
/**********************************************************/
EXEC SQL BEGIN DECLARE SECTION;
char PROCNM[19];
char SCHEMA[9];
char PARMLST[255];
EXEC SQL END DECLARE SECTION;

/**********************************************************/
/* Declare cursors for returning result sets to the caller. */
/**********************************************************/
EXEC SQL DECLARE C1 CURSOR WITH RETURN FOR
SELECT NAME
FROM SYSIBM.SYSTABLES
WHERE CREATOR=:SCHEMA;

main(argc,argv)
int argc;
char *argv[];
{
	/**********************************************************/
/* Copy the input parameters into the area reserved in */
/* the program for SQL processing. */
/**********************************************************/
strcpy(PROCNM, argv[1]);
strcpy(SCHEMA, argv[2]);
	/**********************************************************/
/* Issue the SQL SELECT against the SYSROUTINES */
/* DB2 catalog table. */
/**********************************************************/
strcpy(PARMLST, ""); /* Clear PARMLST */
EXEC SQL
SELECT RUNOPTS INTO :PARMLST
FROM SYSIBM.ROUTINES
```
Example C stored procedure with a GENERAL WITH NULLS linkage convention

You can call a stored procedure that uses the GENERAL WITH NULLS linkage convention from a C program.

This example stored procedure does the following:

- Searches the Db2 catalog table SYSROUTINES for a row that matches the input parameters from the client program. The two input parameters contain values for NAME and SCHEMA.
- Searches the Db2 catalog table SYSTABLES for all tables in which the value of CREATOR matches the value of input parameter SCHEMA. The stored procedure uses a cursor to return the table names.

The linkage convention for this stored procedure is GENERAL WITH NULLS.

The output parameters from this stored procedure contain the SQLCODE from the SELECT operation, and the value of the RUNOPTS column retrieved from the SYSROUTINES table.

The CREATE PROCEDURE statement for this stored procedure might look like this:

```sql
CREATE PROCEDURE GETPRML(PROCNM CHAR(18) IN, SCHEMA CHAR(8) IN, OUTCODE INTEGER OUT, PARMLST VARCHAR(254) OUT)
LANGUAGE C
DETERMINISTIC
READS SQL DATA
EXTERNAL NAME "GETPRML"
COLLID GETPRML
ASUTIME NO LIMIT
PARAMETER STYLE GENERAL WITH NULLS
STAY RESIDENT NO
RUN OPTIONS "MSGFILE(OUTFILE),RPTSTG(ON),RPTOPTS(ON)"
WLM ENVIRONMENT SAMPPROG
PROGRAM TYPE MAIN
SECURITY DB2
RESULT SETS 2
COMMIT ON RETURN NO;
```

The following example is a C stored procedure with linkage convention GENERAL WITH NULLS.

```c
#pragma runopts(plist(os))
#include <stdlib.h>
EXEC SQL INCLUDE SQLCA;

/******************************
/* Declare C variables used for SQL operations on the */
/* parameters. These are local variables to the C program, */
/* which you must copy to and from the parameter list provided */
/* to the stored procedure. */
/******************************
EXEC SQL BEGIN DECLARE SECTION;
char PROCNM[19];
```
char SCHEMA[9];
char PARMLST[255];
struct INDICATORS {
    short int PROCNM_IND;
    short int SCHEMA_IND;
    short int OUT_CODE_IND;
    short int PARMLST_IND;
} PARM_IND;
EXEC SQL END DECLARE SECTION;

EXEC SQL DECLARE C1 CURSOR WITH RETURN FOR
SELECT NAME
FROM SYSIBM.SYSTABLES
WHERE CREATOR=:SCHEMA;

main(argc,argv)
int argc;
char *argv[];
{
    /********************************************************/
    /* Copy the input parameters into the area reserved in */
    /* the local program for SQL processing.                */
    /********************************************************/
    strcpy(PROCNM, argv[1]);
    strcpy(SCHEMA, argv[2]);

    /********************************************************/
    /* Copy null indicator values for the parameter list.   */
    /********************************************************/
    memcpy(&PARM_IND,(struct INDICATORS *) argv[5],
    sizeof(PARM_IND));

    /********************************************************/
    /* If any input parameter is NULL, return an error     */
    /* return code and assign a NULL value to PARMLST.      */
    /********************************************************/
    if (PARM_IND.PROCNM_IND<0 ||
        PARM_IND.SCHEMA_IND<0 ||
        PARM_IND.OUT_CODE_IND<0)
        { /* set output return code */
        PARM_IND.OUT_CODE_IND = 0; /* value is not NULL */
        PARM_IND.PARMLST_IND = -1; /* PARMLST is NULL */
        }
    else {
        /********************************************************/
        /* If the input parameters are not NULL, issue the SQL */
        /* SELECT against the SYSIBM.SYSROUTINES catalog       */
        /* table.                                                */
        /********************************************************/
        strcpy(PARMLST, ""); /* Clear PARMLST */
        EXEC SQL
        SELECT RUNOPTS INTO :PARMLST
        FROM SYSIBM.SYSROUTINES
        WHERE NAME=:PROCNM AND
        SCHEMA=:SCHEMA;
    /********************************************************/
    /* Copy SQLCODE to the output parameter list.          */
    /********************************************************/
        *(int *) argv[3] = SQLCODE;
        PARM_IND.OUT_CODE_IND = 0; /* OUT_CODE is not NULL */
    }

    /********************************************************/
    /* Copy the RUNOPTS value back to the output parameter */
    /* area.                                                 */
    /********************************************************/
    strcpy(argv[4], PARMLST);

    /********************************************************/
    /* Copy the null indicators back to the output parameter*/
    /* area.                                                 */
    /********************************************************/
    memcpy((struct INDICATORS *) argv[5],&PARM_IND,
    sizeof(PARM_IND));

    /********************************************************/
    /* Open cursor C1 to cause DB2 to return a result set   */
    /* ***/
**Defining the SQL communications area, SQLSTATE, and SQLCODE in C and C++**

C and C++ programs that contain SQL statements can include an SQL communications area (SQLCA) to check whether an SQL statement executed successfully. Alternatively, these programs can declare individual SQLCODE and SQLSTATE host variables.

**About this task**

If you specify the SQL processing option STDSQL(YES), do not define an SQLCA. If you do, Db2 ignores your SQLCA, and your SQLCA definition causes compile-time errors. If you specify the SQL processing option STDSQL(NO), include an SQLCA explicitly.

If your application contains SQL statements and does not include an SQL communications area (SQLCA), you must declare individual SQLCODE and SQLSTATE host variables. Your program can use these variables to check whether an SQL statement executed successfully.

**Procedure**

Choose one of the following actions:

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>To define the SQL communications area:</strong></td>
<td>a. Code the SQLCA directly in the program or use the following SQL INCLUDE statement to request a standard SQLCA declaration:</td>
</tr>
<tr>
<td></td>
<td>EXEC SQL INCLUDE SQLCA</td>
</tr>
<tr>
<td></td>
<td>The standard declaration includes both a structure definition and a static data area named 'sqlca'.</td>
</tr>
<tr>
<td></td>
<td>Db2 sets the SQLCODE and SQLSTATE values in the SQLCA after each SQL statement executes. Your application should check these values to determine whether the last SQL statement was successful.</td>
</tr>
<tr>
<td><strong>To declare SQLCODE and SQLSTATE host variables:</strong></td>
<td>a. Declare the SQLCODE variable within a BEGIN DECLARE SECTION statement and an END DECLARE SECTION statement in your program declarations as a long integer:</td>
</tr>
<tr>
<td></td>
<td>long SQLCODE;</td>
</tr>
<tr>
<td></td>
<td>b. Declare the SQLSTATE variable within a BEGIN DECLARE SECTION statement and an END DECLARE SECTION statement in your program declarations as a character array of length 6:</td>
</tr>
<tr>
<td></td>
<td>char SQLSTATE[6];</td>
</tr>
<tr>
<td><strong>Restriction:</strong> Do not declare an SQLSTATE variable as an element of a structure.</td>
<td></td>
</tr>
<tr>
<td><strong>Requirement:</strong> After you declare the SQLCODE and SQLSTATE variables, ensure that all SQL statements in the program are within the scope of the declaration of these variables.</td>
<td></td>
</tr>
</tbody>
</table>

**Related tasks**

Checking the execution of SQL statements
After executing an SQL statement, your program should check for any errors before you commit the data and handle the errors that they represent.

Checking the execution of SQL statements by using the SQLCA
One way to check whether an SQL statement executed successfully is to use the SQL communication area (SQLCA). This area is set apart for communication with Db2.

Checking the execution of SQL statements by using SQLCODE and SQLSTATE
Whenever an SQL statement executes, the SQLCODE and SQLSTATE fields of the SQLCA receive a return code.

Defining the items that your program can use to check whether an SQL statement executed successfully
If your program contains SQL statements, the program should define some infrastructure so that it can check whether the statements executed successfully. You can either include an SQL communications area (SQLCA), which contains SQLCODE and SQLSTATE variables, or declare individual SQLCODE and SQLSTATE host variables.

Defining SQL descriptor areas (SQLDA) in C and C++
If your program includes certain SQL statements, you must define at least one SQL descriptor area (SQLDA). Depending on the context in which it is used, the SQLDA stores information about prepared SQL statements or host variables. This information can then be read by either the application program or Db2.

Procedure
Code the SQLDA directly in the program, or use the following SQL INCLUDE statement to request a standard SQLDA declaration:

```sql
EXEC SQL INCLUDE SQLDA
```

You can place an SQLDA declaration wherever C allows a structure definition. Normal C scoping rules apply. The standard declaration includes only a structure definition with the name sqlda.

Restriction: You must place SQLDA declarations before the first SQL statement that references the data descriptor, unless you use the TWOPASS SQL processing option.

Related tasks
Defining SQL descriptor areas (SQLDA)
If your program includes certain SQL statements, you must define at least one SQL descriptor area (SQLDA). Depending on the context in which it is used, the SQLDA stores information about prepared SQL statements or host variables. This information can then be read by either the application program or Db2.

Related reference
SQL descriptor area (SQLDA) (Db2 SQL)

Declaring host variables and indicator variables in C and C++
You can use host variables, host-variable arrays, and host structures in SQL statements in your program to pass data between Db2 and your application.

Procedure
To declare host variables, host-variable arrays, and host structures:

1. Declare the variables according to the following rules and guidelines:
   - You can have more than one host variable declaration section in your program.
   - You can use class members as host variables. Class members that are used as host variables are accessible to any SQL statement within the class. However, you cannot use class objects as host variables.
   - If you specify the ONEPASS SQL processing option, you must explicitly declare each host variable and each host-variable array before using them in an SQL statement. If you specify the TWOPASS
precompiler option, you must declare each host variable before using it in the DECLARE CURSOR statement.

**Restriction:** The Db2 coprocessor for C/C++ supports only the ONEPASS option.

- If you specify the STDSQL(YES) SQL processing option, you must precede the host language statements that define the host variables and host-variable arrays with the BEGIN DECLARE SECTION statement and follow the host language statements with the END DECLARE SECTION statement. Otherwise, these statements are optional.
- Ensure that any SQL statement that uses a host variable or host-variable array is within the scope of the statement that declares that variable or array.
- If you are using the Db2 precompiler, ensure that the names of host variables and host-variable arrays are unique within the program, even if the variables and variable arrays are in different blocks, classes, procedures, functions, or subroutines. You can qualify the names with a structure name to make them unique.

2. Optional: Define any associated indicator variables, arrays, and structures.

**Related tasks**

Declaring host variables and indicator variables

You can use host variables and indicator variables in SQL statements in your program to pass data between Db2 and your application.

**Host variables in C and C++**

In C and C++ programs, you can specify numeric, character, graphic, binary, LOB, XML, and ROWID host variables. You can also specify result set, table, and LOB locators and LOB and XML file reference variables.

**Restrictions:**

- Only some of the valid C declarations are valid host variable declarations. If the declaration for a variable is not valid, any SQL statement that references the variable might result in the message UNDECLARED HOST VARIABLE.
- C supports some data types and storage classes with no SQL equivalents, such as register storage class, typedef, and long long.
- The following locator data types are special SQL data types that do not have C equivalents:
  - Result set locator
  - Table locator
  - LOB locators
  You cannot use them to define column types.
- Although Db2 allows you to use properly formed L-literals in C application programs, Db2 does not check for all the restrictions that the C compiler imposes on the L-literal. \n- Do not use L-literals in SQL statements. Use Db2 graphic string constants in SQL statements to work with the L-literal.

**Recommendations:**

- Be careful of overflow. For example, suppose that you retrieve an INTEGER column value into a short integer host variable, and the column value is larger than 32767. You get an overflow warning or an error, depending on whether you provide an indicator variable.
- Be careful of truncation. Ensure that the host variable that you declare can contain the data and a NUL terminator, if needed. Retrieving a floating-point or decimal column value into a long integer host variable removes any fractional part of the value.

**Numeric host variables**

The following diagram shows the syntax for declaring numeric host variables.
Notes:

1 If you use the pointer notation of the host variable, you must use the Db2 coprocessor.

Restrictions:

- If your C compiler does not have a decimal data type, no exact equivalent exists for the SQL data type DECIMAL. In this case, you can use one of the following variables or techniques to handle decimal values:
  - An integer or floating-point variable, which converts the value. If you use an integer variable, you lose the fractional part of the number. If the decimal number can exceed the maximum value for an integer or if you want to preserve a fractional value, use floating-point variables. Floating-point numbers are approximations of real numbers. Therefore, when you assign a decimal number to a floating-point variable, the result might be different from the original number.
  - A character-string host variable. Use the CHAR function to get a string representation of a decimal number.
  - The DECIMAL function to explicitly convert a value to a decimal data type, as shown in the following example:

```
long duration=10100; /* 1 year and 1 month */
char result_dt[11];
EXEC SQL SELECT START_DATE + DECIMAL(:duration,8,0) INTO :result_dt FROM TABLE1;
```

- z/OS 1.10 or above (z/OS V1R10 XL C/C++ ) is required to use the decimal floating-point host data type.
- The special C only 'complex floating-point' host data type is not a supported type for host variable.
- The FLOAT precompiler option does not apply to the decimal floating-point host variable types.
- To use decimal floating-point host variable, you must use the Db2 coprocessor.

For floating-point data types, use the FLOAT SQL processing option to specify whether the host variable is in IEEE binary floating-point or z/Architecture hexadecimal floating-point format. Db2 does not check
if the format of the host variable contents match the format that you specified with the FLOAT SQL processing option. Therefore, you need to ensure that your floating-point host variable contents match the format that you specified with the FLOAT SQL processing option. Db2 converts all floating-point input data to z/Architecture hexadecimal floating-point format before storing it.

**Character host variables**

You can specify the following forms of character host variables:

- Single-character form
- NUL-terminated character form
- VARCHAR structured form
- CLOBs

The following diagrams show the syntax for forms other than CLOBs.

The following diagram shows the syntax for declaring single-character host variables.

```
auto extern static
const volatile unsigned
char
, variable-name
=expression

Notes:
1 If you use the pointer notation of the host variable, you must use the Db2 coprocessor.
```

The following diagram shows the syntax for declaring NUL-terminated character host variables.

```
auto extern static
const volatile unsigned
char
, variable-name [length]
=expression

Notes:
1 If you use the pointer notation of the host variable, you must use the Db2 coprocessor.
2 Any string that is assigned to this variable must be NUL-terminated. Any string that is retrieved from this variable is NUL-terminated.
3 A NUL-terminated character host variable maps to a varying-length character string (except for the NUL).
```

The following diagram shows the syntax for declaring varying-length character host variables that use the VARCHAR structured form.
Example

The following example code shows valid and invalid declarations of the VARCHAR structured form:

```sql
EXEC SQL BEGIN DECLARE SECTION;
/* valid declaration of host variable VARCHAR vstring */
struct VARCHAR {
  short len;
  char s[10];
} vstring;
/* invalid declaration of host variable VARCHAR wstring */
struct VARCHAR wstring;
```

For NUL-terminated string host variables, use the SQL processing options PADNTSTR and NOPADNTSTR to specify whether the variable should be padded with blanks. The option that you specify determines where the NUL-terminator is placed.

If you assign a string of length \( n \) to a NUL-terminated string host variable, the variable has one of the values that is shown in the following table.

<table>
<thead>
<tr>
<th>Length of the NUL-terminated string host variable</th>
<th>Value of the variable</th>
</tr>
</thead>
<tbody>
<tr>
<td>Less than or equal to ( n )</td>
<td>The source string up to a length of ( n-1 ) and a NUL at the end of the string. (^1) \nDb2 sets SQLWARN[1] to W and any indicator variable that you provide to the original length of the source string.</td>
</tr>
<tr>
<td>Equal to ( n+1 )</td>
<td>The source string and a NUL at the end of the string. (^1)</td>
</tr>
</tbody>
</table>

\(^1\) You can use the struct tag to define other variables, but you cannot use them as host variables in SQL.
\(^2\) You cannot use `var-1` and `var-2` as host variables in an SQL statement.
\(^3\) If you use the pointer notation of the host variable, you must use the Db2 coprocessor.
<table>
<thead>
<tr>
<th>Length of the NUL-terminated string host variable</th>
<th>Value of the variable</th>
</tr>
</thead>
</table>
| Greater than \( n+1 \) and the source is a fixed-length string | If **PADNTSTR** is in effect  
The source string, blanks to pad the value, and a NUL at the end of the string.  
If **NOPADNTSTR** is in effect  
The source string and a NUL at the end of the string. |
| Greater than \( n+1 \) and the source is a varying-length string | The source string and a NUL at the end of the string.  
1 |

**Note:**  
1. In these cases, whether NOPADNTSTR or PADNTSTR is in effect is irrelevant.

**Restriction:** If you use the Db2 precompiler, you cannot use a host variable that is of the NUL-terminated form in either a PREPARE or DESCRIBE statement. However, if you use the Db2 coprocessor, you can use host variables of the NUL-terminated form in PREPARE, DESCRIBE, and EXECUTE IMMEDIATE statements.

**Graphic host variables**

You can specify the following forms of graphic host variables:

- Single-graphic form
- NUL-terminated graphic form
- VARGRAPHIC structured form.
- DBCLOBs

**Recommendation:** Instead of using the C data type wchar_t to define graphic and vargraphic host variables, use one of the following techniques:

- Define the sqldbchar data type by using the following typedef statement:

```
typedef unsigned short sqldbchar;
```

- Use the sqldbchar data type that is defined in the typedef statement in one of the following files or libraries:
  - SQL library, sql.h
  - Db2 CLI library, sqlcli.h
  - SQLUDF file in data set DSN1210.SDSNC.H

- Use the C data type unsigned short.

Using sqldbchar or unsigned short enables you to manipulate DBCS and Unicode UTF-16 data in the same format in which it is stored in Db2. Using sqldbchar also makes applications easier to port to other platforms.

The following diagrams show the syntax for forms other than DBCLOBs.

The following diagram shows the syntax for declaring single-graphic host variables.
The following diagram shows the syntax for declaring NUL-terminated graphic host variables.

Notes:
1. You cannot use array notation in `variable-name`.
2. The single-graphic form declares a fixed-length graphic string of length 1.

The following diagram shows the syntax for declaring graphic host variables that use the VARGRAPHIC structured form.

Notes:
1. If you use the pointer notation of the host variable, you must use the Db2 coprocessor.
2. `length` must be a decimal integer constant greater than 1 and not greater than 16352.
3. Any string that is assigned to this variable must be NUL-terminated. Any string that is retrieved from this variable is NUL-terminated.
4. The NUL-terminated graphic form does not accept single-byte characters for the variable.
Notes:

1 You can use the struct tag to define other variables, but you cannot use them as host variables in SQL.
2 \texttt{var-1} must be less than or equal to \textit{length}.
3 You cannot use \texttt{var-1} or \texttt{var-2} as host variables in an SQL statement.
4 \textit{length} must be a decimal integer constant greater than 1 and not greater than 16352.
5 If you use the pointer notation of the host variable, you must use the Db2 coprocessor.

Example

The following example shows valid and invalid declarations of graphic host variables that use the VARGRAPHIC structured form:

```
EXEC SQL BEGIN DECLARE SECTION;
/* valid declaration of host variable structured vgraph */
struct VARGRAPH {
    short len;
    sqldbchar d[10];
} vgraph;

/* invalid declaration of host variable structured wgraph */
struct VARGRAPH wgraph;
```

Binary host variables

You can specify the following forms of binary host variables:

- Fixed-length strings
- Varying-length strings
- BLOBs

The following diagrams show the syntax for forms other than BLOBs.

The following diagram shows the syntax for declaring binary host variables.

```
+--------------------------+
| auto | extern | static |
+--------------------------+
| const | volatile |
+--------------------------+
| variable-name ; |
+--------------------------+
```

Notes:

1 The length must be a value in the range 1 - 255.
Notes:

1 For VARBINARY host variables, the length must be in the range 1 - 32704.

The C language does not have variables that correspond to the SQL binary data types BINARY and VARBINARY. To create host variables that can be used with these data types, use the SQL TYPE IS clause. The SQL precompiler replaces this declaration with the C language structure in the output source member.

When you reference a BINARY or VARBINARY host variable in an SQL statement, you must use the variable that you specify in the SQL TYPE declaration. When you reference the host variable in a host language statement, you must use the variable that Db2 generates.

**Examples of binary variable declarations**

The following table shows examples of variables that Db2 generates when you declare binary host variables.

<table>
<thead>
<tr>
<th>Variable declaration that you include in your C program</th>
<th>Corresponding variable that Db2 generates in the output source member</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQL TYPE IS BINARY(10) bin_var;</td>
<td>char bin_var[10]</td>
</tr>
<tr>
<td>SQL TYPE IS VARBINARY(10) vbin_var;</td>
<td>struct { short length; char data[10]; } vbin_var;</td>
</tr>
</tbody>
</table>

**Recommendation:** Be careful when you use binary host variables with C and C++. The SQL TYPE declaration for BINARY and VARBINARY does not account for the NUL-terminator that C expects, because binary strings are not NUL-terminated strings. Also, the binary host variable might contain zeroes at any point in the string.

**Result set locators**

The following diagram shows the syntax for declaring result set locators.
Table locators
The following diagram shows the syntax for declaring table locators.

LOB variables, locators, and file reference variables
The following diagram shows the syntax for declaring BLOB, CLOB, and DBCLOB host variables, locators, and file reference variables.
Notes:

1 Specify the initial value as a series of expressions. For example, specify = \{expression, expression\}. For BLOB_FILE, CLOB_FILE, and DBCLOB_FILE, specify = \{name_length, data_length, file_option_map, file_name\}.

**XML data host and file reference variables**

The following diagram shows the syntax for declaring BLOB, CLOB, and DBCLOB host variables and file reference variables for XML data types.
**Notes:**

1. Specify the initial value as a series of expressions. For example, specify `{expression, expression}`. For BLOB_FILE, CLOB_FILE, and DBCLOB_FILE, specify `{name_length, data_length, file_option_map, file_name}`.

---

**ROWID host variables**

The following diagram shows the syntax for declaring ROWID host variables.

---

**Constants**

The syntax for constants in C and C++ programs differs from the syntax for constants in SQL statements in the following ways:

- **C/C++** uses various forms for numeric literals (possible suffixes are: ll, LL, u, U, f,F,l,L,df,DF, dd, DD, dl, DL,d, D). For example, in C/C++:
  - 4850976 is a decimal literal
  - 0x4bD is a hexadecimal integer literal
  - 03245 is an octal integer literal
  - 3.2E+4 is a double floating-point literal
  - 3.2E+4f is a float floating-point literal
  - 3.2E+4l is a long double floating-point literal
  - 0x4bDP+4 is a double hexadecimal floating-point literal
  - 22.2df is a _Decimal32 decimal floating-point literal
  - 0.00D is a fixed-point decimal literal (z/OS only when LANG_LVL(EXTENDED) is specified)

- Use C/C++ literal form only outside of SQL statements. Within SQL statements, use numeric constants.

- In C, character constants and string constants can use escape sequences. You cannot use the escape sequences in SQL statements.

- Apostrophes and quotation marks have different meanings in C and SQL. In C, you can use double quotation marks to delimit string constants, and apostrophes to delimit character constants.

**Example: Use of quotation marks in C**

```c
printf( "%d lines read. \n", num_lines);
```

**Example: Use of apostrophes in C**

```c
#define NUL '\0'
```

In SQL, you can use double quotation marks to delimit identifiers and apostrophes to delimit string constants.

**Example: quotation marks in SQL**

```sql
SELECT "COL#1" FROM TBL1;
```
Example: apostrophes in SQL

SELECT COL1 FROM TBL1 WHERE COL2 = 'BELL';

- Character data in SQL is distinct from integer data. Character data in C is a subtype of integer data.

**Related concepts**

Host variables
Use host variables to pass a single data item between Db2 and your application.

Using host variables in SQL statements
Use scalar host variables in embedded SQL statements to represent a single value. Host variables are useful for storing retrieved data or for passing values that are to be assigned or used for comparisons.

**Related tasks**

Determining whether a retrieved value in a host variable is null or truncated
Before your application manipulates the data that was retrieved from Db2 into a host variable, determine if the value is null. Also determine if it was truncated when assigned to the variable. You can use indicator variables to obtain this information.

Inserting a single row by using a host variable
Use host variables in your INSERT statement when you don't know at least some of the values to insert until the program runs.

Inserting null values into columns by using indicator variables or arrays
If you need to insert null values into a column, using an indicator variable or array is an easy way to do so. An indicator variable or array is associated with a particular host variable or host-variable array.

Storing LOB data in a table
Db2 handles LOB data differently than it handles other kinds of data. As a result, in some cases, you need to take additional actions when you define LOB columns and insert the LOB data.

Retrieving a single row of data into host variables
If you know that your query returns only one row, you can specify one or more host variables to contain the column values of the retrieved row.

Retrieving a single row of data into a host structure
If you know that your query returns multiple column values for only one row, you can specify a host structure to contain the column values.

Updating data by using host variables
When you want to update a value in a Db2 table, but you do not know the exact value until the program runs, use host variables. Db2 can change a table value to match the current value of the host variable.

**Related reference**

Descriptions of SQL processing options
You can specify any SQL processing options regardless of whether you use the Db2 precompiler or the Db2 coprocessor. However, the Db2 coprocessor might ignore certain options because host language compiler options exist that provide the same information.

**Host-variable arrays in C and C++**

In C and C++ programs, you can specify numeric, character, graphic, binary, LOB, XML, and ROWID host-variable arrays. You can also specify LOB locators and LOB and XML file reference variables.

Host-variable arrays can be referenced only as a simple reference in the following contexts. In syntax diagrams, host-variable-array designates a reference to a host-variable array.

- In a FETCH statement for a multiple-row fetch. See FETCH (Db2 SQL).
- In an INSERT statement with multiple rows of source data. This is also referred to as multiple-row insert. See INSERT (Db2 SQL).
- In a MERGE statement with multiple rows of source data. See MERGE (Db2 SQL).
- In an EXECUTE statement to provide a value for a parameter marker in a dynamic multi-row INSERT or MERGE. See EXECUTE (Db2 SQL).
Restrictions:

- Only some of the valid C declarations are valid host-variable array declarations. If the declaration for a variable array is not valid, any SQL statement that references the variable array might result in the message UNDECLARED HOST VARIABLE ARRAY.
- For both C and C++, you cannot specify the _packed attribute on the structure declarations for the following arrays that are used in multiple-row INSERT, FETCH, and MERGE statements:
  - varying-length character arrays
  - varying-length graphic arrays
  - LOB arrays

In addition, the #pragma pack(1) directive cannot be in effect if you plan to use these arrays in multiple-row statements.

Numeric host-variable arrays

The following diagram shows the syntax for declaring numeric host-variable arrays.

Notes:

1. dimension must be an integer constant between 1 and 32767.
Example

The following example shows a declaration of a numeric host-variable array:

```sql
EXEC SQL BEGIN DECLARE SECTION;
/* declaration of numeric host-variable array */
long serial_num[10];
EXEC SQL END DECLARE SECTION;
```

Character host-variable arrays

You can specify the following forms of character host-variable arrays:

- NUL-terminated character form
- VARCHAR structured form
- CLOBs

The following diagrams show the syntax for forms other than CLOBs.

The following diagram shows the syntax for declaring NUL-terminated character host-variable arrays.

```
variable-name [dimension] [length] = { expression }
```

Notes:

1. `dimension` must be an integer constant between 1 and 32767.
2. Any string that is assigned to this variable must be NUL-terminated. Any string that is retrieved from this variable is NUL-terminated.
3. The strings in a NUL-terminated character host-variable array map to varying-length character strings (except for the NUL).

The following diagram shows the syntax for declaring varying-length character host-variable arrays that use the VARCHAR structured form.
Notes:

1 You can use the struct tag to define other variables, but you cannot use them as host-variable arrays in SQL.
2 var-1 must be a scalar numeric variable.
3 var-2 must be a scalar CHAR array variable.
4 dimension must be an integer constant between 1 and 32767.

Example
The following example shows valid and invalid declarations of VARCHAR host-variable arrays.

```sql
EXEC SQL BEGIN DECLARE SECTION;
/* valid declaration of VARCHAR host-variable array */
struct VARCHAR {
  short len;
  char s[10];
} name[10];

/* invalid declaration of VARCHAR host-variable array */
struct VARCHAR name[10];
```

Binary host-variable arrays
The following diagram shows the syntax for declaring binary host-variable arrays.

Notes:
**Graphic host-variable arrays**

You can specify the following forms of graphic host-variable arrays:

- NUL-terminated graphic form
- VARGRAPHIC structured form.

**Recommendation:** Instead of using the C data type wchar_t to define graphic and vargraphic host-variable arrays, use one of the following techniques:

- Define the sqldbchar data type by using the following typedef statement:

  ```
  typedef unsigned short sqldbchar;
  ```

- Use the sqldbchar data type that is defined in the typedef statement in the header files that are supplied by Db2.

- Use the C data type unsigned short.

The following diagram shows the syntax for declaring NUL-terminated graphic host-variable arrays.

The following diagram shows the syntax for declaring graphic host-variable arrays that use the VARGRAPHIC structured form.

Notes:

1. *dimension* must be an integer constant between 1 and 32767.
2. *length* must be a decimal integer constant greater than 1 and not greater than 16352.
3. Any string that is assigned to this variable must be NUL-terminated. Any string that is retrieved from this variable is NUL-terminated.
4. Do not assign single-byte characters into a NUL-terminated graphic host-variable array.
Example

The following example shows valid and invalid declarations of graphic host-variable arrays that use the VARGRAPHIC structured form.

```sql
EXEC SQL BEGIN DECLARE SECTION;
/* valid declaration of host-variable array vgraph */
struct VARGRAPH {
    short len;
    sqldbchar d[10];
} vgraph[20];

/* invalid declaration of host-variable array vgraph */
struct VARGRAPH vgraph[20];
```

LOB, locator, and file reference variable arrays

The following diagram shows the syntax for declaring BLOB, CLOB, and DBCLOB host-variable arrays, locators, and file reference variables.
Notes:

1. *dimension* must be an integer constant between 1 and 32767.

**XML host and file reference variable arrays**

The following diagram shows the syntax for declaring BLOB, CLOB, and DBCLOB host-variable arrays and file reference variable arrays for XML data types.
ROWID variable arrays

The following diagram shows the syntax for declaring ROWID variable arrays.

Notes:
1 \textit{dimension} must be an integer constant between 1 and 32767.

Related concepts
Using host-variable arrays in SQL statements
Use host-variable arrays in embedded SQL statements to represent values that the program does not know until the query is executed. Host-variable arrays are useful for storing a set of retrieved values or for passing a set of values that are to be inserted into a table.

Host-variable arrays
You can use host-variable arrays to pass a data array between Db2 and your application. A host-variable array is a data array that is declared in the host language to be used within an SQL statement.

Host-variable arrays in PL/I, C, C++, and COBOL (Db2 SQL)

Related tasks
Inserting multiple rows of data from host-variable arrays
Use host-variable arrays in your INSERT statement when you do not know at least some of the values to insert until the program runs.

Storing LOB data in a table
Db2 handles LOB data differently than it handles other kinds of data. As a result, in some cases, you need to take additional actions when you define LOB columns and insert the LOB data.

Retrieving multiple rows of data into host-variable arrays
If you know that your query returns multiple rows, you can specify host-variable arrays to store the retrieved column values.

Host structures in C and C++
A C host structure contains an ordered group of data fields.

Host structures
The following diagram shows the syntax for declaring host structures.
VARCHAR structures

The following diagram shows the syntax for VARCHAR structures that are used within declarations of host structures.
**VARGRAPHIC structures**

The following diagram shows the syntax for VARGRAPHIC structures that are used within declarations of host structures.

```plaintext
struct
tag
    { signed short int var-3 ;
    char var-4 [ length ] ; }
```

**Binary structures**

The following diagram shows the syntax for binary structures that are used within declarations of host structures.

```plaintext
SQL TYPE IS
  BINARY
  VARBINARY (length)
  BINARY VARYING
```

**LOB data types**

The following diagram shows the syntax for LOB data types that are used within declarations of host structures.

```plaintext
SQL TYPE IS
  BINARY LARGE OBJECT
  BLOB
  CHAR LARGE OBJECT
  CHAR LARGE OBJECT
  CLOB
  DBCLOB
  BLOB_LOCATOR
  CLOB_LOCATOR
  DBCLOB_LOCATOR
  ( length
    K M G )
```
LOB data types for XML data

The following diagram shows the syntax for LOB data types that are used within declarations of host structures for XML data.

![Diagram of LOB data types for XML data]

Example

In the following example, the host structure is named target, and it contains the fields c1, c2, and c3. c1 and c3 are character arrays, and c2 is a host variable that is equivalent to the SQL VARCHAR data type. The target host structure can be part of another host structure but must be the deepest level of the nested structure.

```c
struct {char  c1[3];
    struct {short len;
        char  data[5];
    }c2;
    char  c3[2];
} target;
```

Related concepts

- Host structures
- Use host structures to pass a group of host variables between Db2 and your application.

Indicator variables, indicator arrays, and host structure indicator arrays in C and C++

An indicator variable is a 2-byte integer (short int). An indicator variable array is an array of 2-byte integers (short int). You declare indicator variables in the same way as host variables. You can mix the declarations of the two types of variables.

The following diagram shows the syntax for declaring an indicator variable in C and C++.
The following diagram shows the syntax for declaring an indicator array or a host structure indicator array in C and C++.

Notes:
1 dimension must be an integer constant between 1 and 32767.

Example
The following example shows a FETCH statement with the declarations of the host variables that are needed for the FETCH statement and their associated indicator variables.

EXEC SQL FETCH CLS_CURSOR INTO :ClsCd,
    :Day :DayInd,
    :Bgn :BgnInd,
    :End :EndInd;

You can declare these variables as follows:

EXEC SQL BEGIN DECLARE SECTION;
char ClsCd[8];
char Bgn[9];
char End[9];
short Day, DayInd, BgnInd, EndInd;
EXEC SQL END DECLARE SECTION;

Related concepts
Indicator variables, arrays, and structures
An indicator variable is associated with a particular host variable. Each indicator variable contains a small integer value that indicates some information about the associated host variable. Indicator arrays and structures serve the same purpose for host-variable arrays and structures.

Related tasks
Inserting null values into columns by using indicator variables or arrays
If you need to insert null values into a column, using an indicator variable or array is an easy way to do so. An indicator variable or array is associated with a particular host variable or host-variable array.

**Referencing pointer host variables in C programs**

If you use the Db2 coprocessor, you can reference any declared pointer host variables in your SQL statements.

**Procedure**

Specify the pointer host variable exactly as it was declared.

The only exception is when you reference pointers to nul-terminated character arrays. In this case, you do not have to include the parentheses that were part of the declaration.

**Examples**

**Examples: Scalar pointer host variable references**

<table>
<thead>
<tr>
<th>Declaration</th>
<th>Description</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>short *hvshortp;</td>
<td>hvshortp is a pointer host variable that points to two bytes of storage.</td>
<td>EXEC SQL set:*hvshortp=123;</td>
</tr>
<tr>
<td>double *hvdoubp;</td>
<td>hvdoubp is a pointer host variable that points to eight bytes of storage.</td>
<td>EXEC SQL set:*hvdoubp=456;</td>
</tr>
<tr>
<td>char (*hvcharpn) [20];</td>
<td>hvcharpn is a pointer host variable that points to a nul-terminated character array of up to 20 bytes.</td>
<td>EXEC SQL set:*hvcharpn='nul_terminated';</td>
</tr>
</tbody>
</table>

**Example: Bounded character pointer host variable reference**

Suppose that your program declares the following bounded character pointer host variable:

```c
struct {
  unsigned long len;
  char * data;
} hvbcharp;
```

The following example references this bounded character pointer host variable:

```c
hvcharp.len = dynlen;  // a
hvcharp.data = (char *) malloc (hvcharp.len);  // b
EXEC SQL set :hvcharp = 'data buffer with length';  // c
```

**Note:**

a
dynlen can be either a compile time constant or a variable with a value that is assigned at run time.

b
Storage is dynamically allocated for hvcharp.data.

c
The SQL statement references the name of the structure, not an element within the structure.
Examples: Array pointer host variable references

Table 105. Example references to array pointer host variables

<table>
<thead>
<tr>
<th>Declaration</th>
<th>Description</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>short * hvarrpl[6]</td>
<td>hvarrp1 is an array of 6 pointers that point to two bytes of storage each.</td>
<td>EXEC SQL set:*hvarrpl[n]=123;</td>
</tr>
<tr>
<td>double * hvarrp2[3]</td>
<td>hvarrp2 is an array of 3 pointers that point to 8 bytes of storage each.</td>
<td>EXEC SQL set:*hvarrp2[n]=456;</td>
</tr>
</tbody>
</table>
| struct {
  unsigned long len;
  char * data;
} hvarrp3[5]; | hvarrp3 is an array of 5 bounded character pointers. | EXEC SQL set :hvarrp3[n] = 'data buffer with length' |

Example: Structure array host variable reference

Suppose that your program declares the following pointer to the structure tbl_struct:

```c
struct tbl_struct *ptr_tbl_struct =
    (struct tbl_struct *) malloc (sizeof (struct tbl_struct) * n);
```

To reference this data in SQL statements, use the pointer as shown in the following example. Assume that tbl_sel_cur is a declared cursor.

```sql
for (L_col_cnt = 0; L_col_cnt < n; L_con_cnt++)
{
    EXEC SQL FETCH tbl_sel_cur INTO :ptr_tbl_struct [L_col_cnt]
    ...
}
```

Related tasks

Declaring pointer host variables in C programs

If you use the Db2 coprocessor, you can use pointer host variables with statically or dynamically allocated storage. These pointer host variables can point to numeric data, non-numeric data, or a structure.

Declaring pointer host variables in C programs

If you use the Db2 coprocessor, you can use pointer host variables with statically or dynamically allocated storage. These pointer host variables can point to numeric data, non-numeric data, or a structure.

About this task

You can declare the following types of pointer host variables:

**Scalar pointer host variable**

A host variable that points to numeric or non-numeric scalar data.

**Array pointer host variable**

A host variable that is an array of pointers.

**Structure array host variable**

A host variable that points to a structure.

Procedure

Include an asterisk (*) in each variable declaration to indicate that the variable is a pointer.

Restrictions:

- You cannot use pointer host variables that point to character data of an unknown length. For example, do not specify the following declaration: char * hvcharpu. Instead, specify the length of the data.
by using a bounded character pointer host variable. A bounded character pointer host variable is a host variable that is declared as a structure with the following elements:

- A 4-byte field that contains the length of the storage area.
- A pointer to the non-numeric dynamic storage area.

• You cannot use untyped pointers. For example, do not specify the following declaration: `void * untypedprt`.

### Examples

#### Example: Scalar pointer host variable declarations

<table>
<thead>
<tr>
<th>Declaration</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>short *hvshortp;</td>
<td>hvshortp is a pointer host variable that points to two bytes of storage.</td>
</tr>
<tr>
<td>double *hvdoubp;</td>
<td>hvdoubp is a pointer host variable that points to eight bytes of storage.</td>
</tr>
<tr>
<td>char (*hvcharpn)[20];</td>
<td>hvcharpn is a pointer host variable that points to a nul-terminated character array of up to 20 bytes.</td>
</tr>
</tbody>
</table>

#### Example: Bounded character pointer host variable declaration

The following example code declares a bounded character pointer host variable called hvbcharp with two elements: len and data.

```c
struct {
    unsigned long len;
    char * data;
} hvbcharp;
```

#### Example: Array pointer host variable declarations

<table>
<thead>
<tr>
<th>Declaration</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>short *hvarrp1[6]</td>
<td>hvarrp1 is an array of 6 pointers that point to two bytes of storage each.</td>
</tr>
<tr>
<td>double *hvarrp2[3]</td>
<td>hvarrp2 is an array of 3 pointers that point to 8 bytes of storage each.</td>
</tr>
</tbody>
</table>
| struct {
    unsigned long len;
    char * data;
} hvbarrp3[5]; | hvbarrp3 is an array of 5 bounded character pointers. |

#### Example: Structure array host variable declaration

The following example code declares a table structure called tbl_struct.

```c
struct tbl_struct
{
    char colname[20];
    small int colno;
    small int coltype;
    small int collen;
};
```

The following example code declares a pointer to the structure tbl_struct. Storage is allocated dynamically for up to n rows.
struct tbl_struct *ptr_tbl_struct =
(struct tbl_struct *) malloc (sizeof (struct tbl_struct) * n);

Related tasks
Referencing pointer host variables in C programs
If you use the Db2 coprocessor, you can reference any declared pointer host variables in your SQL statements.

Equivalent SQL and C data types

When you declare host variables in your C programs, the precompiler uses equivalent SQL data types. When you retrieve data of a particular SQL data type into a host variable, you need to ensure that the host variable is of an equivalent data type.

The following table describes the SQL data type and the base SQLTYPE and SQLLEN values that the precompiler uses for host variables in SQL statements.

<table>
<thead>
<tr>
<th>C host variable data type</th>
<th>SQLTYPE of host variable</th>
<th>SQLLEN of host variable</th>
<th>SQL data type</th>
</tr>
</thead>
<tbody>
<tr>
<td>short int</td>
<td>500</td>
<td>2</td>
<td>SMALLINT</td>
</tr>
<tr>
<td>long int</td>
<td>496</td>
<td>4</td>
<td>INTEGER</td>
</tr>
<tr>
<td>long long</td>
<td>492</td>
<td>8</td>
<td>BIGINT⁵</td>
</tr>
<tr>
<td>decimal(p,s)²</td>
<td>484</td>
<td>p in byte 1, s in byte 2</td>
<td>DECIMAL(p,s)²</td>
</tr>
<tr>
<td>_Decimal32</td>
<td>996/997</td>
<td>4</td>
<td>DECFLOAT(16)⁷, ⁸</td>
</tr>
<tr>
<td>_Decimal64</td>
<td>996/997</td>
<td>8</td>
<td>DECFLOAT(16)⁸</td>
</tr>
<tr>
<td>_Decimal128</td>
<td>996/997</td>
<td>16</td>
<td>DECFLOAT(34)⁸</td>
</tr>
<tr>
<td>float</td>
<td>480</td>
<td>4</td>
<td>FLOAT (single precision)</td>
</tr>
<tr>
<td>double</td>
<td>480</td>
<td>8</td>
<td>FLOAT (double precision)</td>
</tr>
<tr>
<td>SQL TYPE IS BINARY(n), 1&lt;=n&lt;=255</td>
<td>912</td>
<td>n</td>
<td>BINARY(n)</td>
</tr>
<tr>
<td>SQL TYPE IS VARBINARY(n), 1&lt;=n&lt;=32704</td>
<td>908</td>
<td>n</td>
<td>VARBINARY(n)</td>
</tr>
<tr>
<td>Single-character form</td>
<td>452</td>
<td>1</td>
<td>CHAR(1)</td>
</tr>
<tr>
<td>NUL-terminated character form</td>
<td>460</td>
<td>n</td>
<td>VARCHAR (n-1)</td>
</tr>
<tr>
<td>C host variable data type</td>
<td>SQLTYPE of host variable</td>
<td>SQLLEN of host variable</td>
<td>SQL data type</td>
</tr>
<tr>
<td>---------------------------</td>
<td>--------------------------</td>
<td>-------------------------</td>
<td>---------------</td>
</tr>
<tr>
<td>VARCHAR structured form $1\leq n\leq255$</td>
<td>448</td>
<td>$n$</td>
<td>VARCHAR($n$)</td>
</tr>
<tr>
<td>VARCHAR structured form $n&gt;255$</td>
<td>456</td>
<td>$n$</td>
<td>VARCHAR($n$)</td>
</tr>
<tr>
<td>Single-graphic form</td>
<td>468</td>
<td>1</td>
<td>GRAPHIC(1)</td>
</tr>
<tr>
<td>NUL-terminated graphic form</td>
<td>400</td>
<td>$n$</td>
<td>VARGRAPHIC ($n-1$)</td>
</tr>
<tr>
<td>VARGRAPHIC structured form $1\leq n&lt;128$</td>
<td>464</td>
<td>$n$</td>
<td>VARGRAPHIC($n$)</td>
</tr>
<tr>
<td>VARGRAPHIC structured form $n&gt;127$</td>
<td>472</td>
<td>$n$</td>
<td>VARGRAPHIC($n$)</td>
</tr>
<tr>
<td>SQL TYPE IS RESULT_SET_LOCATOR</td>
<td>972</td>
<td>4</td>
<td>Result set locator$^3$</td>
</tr>
<tr>
<td>SQL TYPE IS TABLE LIKE table-name AS LOCATOR</td>
<td>976</td>
<td>4</td>
<td>Table locator$^3$</td>
</tr>
<tr>
<td>SQL TYPE IS BLOB_LOCATOR</td>
<td>960</td>
<td>4</td>
<td>BLOB locator$^3$</td>
</tr>
<tr>
<td>SQL TYPE IS CLOB_LOCATOR</td>
<td>964</td>
<td>4</td>
<td>CLOB locator$^3$</td>
</tr>
<tr>
<td>SQL TYPE IS DBCLOB_LOCATOR</td>
<td>968</td>
<td>4</td>
<td>DBCLOB locator$^3$</td>
</tr>
<tr>
<td>SQL TYPE IS XML AS BLOB(n)</td>
<td>404</td>
<td>$n$</td>
<td>BLOB($n$)</td>
</tr>
<tr>
<td>SQL TYPE IS CLOB(n)</td>
<td>408</td>
<td>$n$</td>
<td>CLOB($n$)</td>
</tr>
<tr>
<td>SQL TYPE IS DBCLOB(n)</td>
<td>412</td>
<td>$n$</td>
<td>DBCLOB($n$)$^4$</td>
</tr>
<tr>
<td>SQL TYPE IS XML AS BLOB(n)</td>
<td>404</td>
<td>0</td>
<td>XML</td>
</tr>
</tbody>
</table>
### Notes:

1. If a host variable includes an indicator variable, the SQLTYPE value is the base SQLTYPE value plus 1.
2. \( p \) is the *precision*; in SQL terminology, this is the total number of digits. In C, this is called the *size*.
   
   \( s \) is the *scale*; in SQL terminology, this is the number of digits to the right of the decimal point. In C, this is called the *precision*.

   C++ does not support the decimal data type.

3. Do not use this data type as a column type.

4. \( n \) is the number of double-byte characters.

5. No exact equivalent. Use DECIMAL(19,0).

6. The C data type long maps to the SQL data type BIGINT.

7. DFP host variable with a length of 4 is supported while DFP column can be defined only with length 8(DECFLOAT(16)) or 16(DECFLOAT(34)).

8. To use the decimal floating-point host data type, you must do the following:
   - Use z/OS 1.10 or later (z/OS V1R10 XL C/C++).
   - Compile with the C/C++ compiler option, DFP.
   - Specify the SQL compiler option to enable the Db2 coprocessor.
   - Specify C/C++ compiler option, ARCH(7). It is required by the DFP compiler option if the DFP type is used in the source.
   - Specify ‘DEFINE(__STDC_WANT_DEC_FP__)’ compiler option because DFP is not officially part of the C/C++ Language Standard.

---

The following table shows equivalent C host variables for each SQL data type. Use this table to determine the C data type for host variables that you define to receive output from the database. For example, if you retrieve TIMESTAMP data, you can define a variable of NUL-terminated character form or VARCHAR structured form.
This table shows direct conversions between SQL data types and C data types. However, a number of SQL data types are compatible. When you do assignments or comparisons of data that have compatible data types, Db2 converts those compatible data types.

<table>
<thead>
<tr>
<th>SQL data type</th>
<th>C host variable equivalent</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>SMALLINT</td>
<td>short int</td>
<td></td>
</tr>
<tr>
<td>INTEGER</td>
<td>long int</td>
<td></td>
</tr>
<tr>
<td>DECIMAL(p,s) or NUMERIC(p,s)</td>
<td>decimal</td>
<td>You can use the double data type if your C compiler does not have a decimal data type; however, double is not an exact equivalent.</td>
</tr>
<tr>
<td>REAL or FLOAT(n)</td>
<td>float</td>
<td>1&lt;=n&lt;=21</td>
</tr>
<tr>
<td>DOUBLE PRECISION or FLOAT(n)</td>
<td>double</td>
<td>22&lt;=n&lt;=53</td>
</tr>
<tr>
<td>DECFLOAT(16)</td>
<td>_Decimal32</td>
<td></td>
</tr>
<tr>
<td>DECFLOAT(34)</td>
<td>_Decimal128</td>
<td></td>
</tr>
<tr>
<td>BIGINT</td>
<td>long long, long long int, and sqlint64</td>
<td></td>
</tr>
<tr>
<td>BINARY(n)</td>
<td>SQL TYPE IS BINARY(n)</td>
<td>1&lt;=n&lt;=255</td>
</tr>
<tr>
<td></td>
<td></td>
<td>If data can contain character NULs (\0), certain C and C++ library functions might not handle the data correctly. Ensure that your application handles the data properly.</td>
</tr>
<tr>
<td>VARBINARY(n)</td>
<td>SQL TYPE IS VARBINARY(n)</td>
<td>1&lt;=n&lt;=32704</td>
</tr>
<tr>
<td>CHAR(1)</td>
<td>single-character form</td>
<td></td>
</tr>
<tr>
<td>CHAR(n)</td>
<td>no exact equivalent</td>
<td>If n&gt;1, use NUL-terminated character form</td>
</tr>
<tr>
<td>VARCHAR(n)</td>
<td>NUL-terminated character form</td>
<td>If data can contain character NULs (\0), use VARCHAR structured form. Allow at least n+1 to accommodate the NUL-terminator.</td>
</tr>
<tr>
<td>GRAPHIC(1)</td>
<td>single-graphic form</td>
<td></td>
</tr>
<tr>
<td>GRAPHIC(n)</td>
<td>no exact equivalent</td>
<td>If n&gt;1, use NUL-terminated graphic form. n is the number of double-byte characters.</td>
</tr>
<tr>
<td>VARGRAPHIC(n)</td>
<td>NUL-terminated graphic form</td>
<td>If data can contain graphic NUL values (\0\0), use VARGRAPHIC structured form. Allow at least n+1 to accommodate the NUL-terminator. n is the number of double-byte characters.</td>
</tr>
</tbody>
</table>

NVARGRAPHIC structured form

n is the number of double-byte characters.
Table 109. C host variable equivalents that you can use when retrieving data of a particular SQL data type (continued)

<table>
<thead>
<tr>
<th>SQL data type</th>
<th>C host variable equivalent</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>DATE</td>
<td>NUL-terminated character form</td>
<td>If you are using a date exit routine, that routine determines the length. Otherwise, allow at least 11 characters to accommodate the NUL-terminator.</td>
</tr>
<tr>
<td></td>
<td>VARCHAR structured form</td>
<td>If you are using a date exit routine, that routine determines the length. Otherwise, allow at least 10 characters.</td>
</tr>
<tr>
<td>TIME</td>
<td>NUL-terminated character form</td>
<td>If you are using a time exit routine, the length is determined by that routine. Otherwise, the length must be at least 7; to include seconds, the length must be at least 9 to accommodate the NUL-terminator.</td>
</tr>
<tr>
<td></td>
<td>VARCHAR structured form</td>
<td>If you are using a time exit routine, the length is determined by that routine. Otherwise, the length must be at least 6; to include seconds, the length must be at least 8.</td>
</tr>
<tr>
<td>TIMESTAMP</td>
<td>NUL-terminated character form</td>
<td>The length must be at least 20. To include microseconds, the length must be 27. If the length is less than 27, truncation occurs on the microseconds part.</td>
</tr>
<tr>
<td></td>
<td>VARCHAR structured form</td>
<td>The length must be at least 19. To include microseconds, the length must be 26. If the length is less than 26, truncation occurs on the microseconds part.</td>
</tr>
<tr>
<td>TIMESTAMP(0)</td>
<td>NUL-terminated character form</td>
<td>The length must be at least 20.</td>
</tr>
<tr>
<td></td>
<td>VARCHAR structured form</td>
<td>The length must be at least 19.</td>
</tr>
<tr>
<td>TIMESTAMP(p) p &gt; 0</td>
<td>NUL-terminated character form</td>
<td>The length must be at least 20. To include fractional seconds, the length must be 21+x where x is the number of fractional seconds to include; if x is less than p, truncation occurs on the fractional seconds part.</td>
</tr>
<tr>
<td></td>
<td>VARCHAR structured form</td>
<td>The length must be at least 19. To include fractional seconds, the length must be 20+x where x is the number of fractional seconds to include; if x is less than p, truncation occurs on the fractional seconds part.</td>
</tr>
<tr>
<td>TIMESTAMP(0) WITH TIME ZONE</td>
<td>NUL-terminated character form</td>
<td>The length must be at least 26.</td>
</tr>
<tr>
<td></td>
<td>VARCHAR structured form</td>
<td>The length must be at least 25.</td>
</tr>
<tr>
<td>SQL data type</td>
<td>C host variable equivalent</td>
<td>Notes</td>
</tr>
<tr>
<td>---------------</td>
<td>-----------------------------</td>
<td>-------</td>
</tr>
<tr>
<td>TIMESTAMP(p) WITH TIME ZONE</td>
<td>NUL-terminated character form</td>
<td>The length must be at least 27+p.</td>
</tr>
<tr>
<td></td>
<td>VARCHAR structured form</td>
<td>The length must be at least 26+p.</td>
</tr>
<tr>
<td>Result set locator</td>
<td>SQL TYPE IS RESULT_SET_LOCATOR</td>
<td>Use this data type only for receiving result sets. Do not use this data type as a column type.</td>
</tr>
<tr>
<td>Table locator</td>
<td>SQL TYPE IS TABLE LIKE table-name AS LOCATOR</td>
<td>Use this data type only in a user-defined function or stored procedure to receive rows of a transition table. Do not use this data type as a column type.</td>
</tr>
<tr>
<td>BLOB locator</td>
<td>SQL TYPE IS BLOB_LOCATOR</td>
<td>Use this data type only to manipulate data in BLOB columns. Do not use this data type as a column type.</td>
</tr>
<tr>
<td>CLOB locator</td>
<td>SQL TYPE IS CLOB_LOCATOR</td>
<td>Use this data type only to manipulate data in CLOB columns. Do not use this data type as a column type.</td>
</tr>
<tr>
<td>DBCLOB locator</td>
<td>SQL TYPE IS DBCLOB_LOCATOR</td>
<td>Use this data type only to manipulate data in DBCLOB columns. Do not use this data type as a column type.</td>
</tr>
<tr>
<td>BLOB(n)</td>
<td>SQL TYPE IS BLOB(n)</td>
<td>$1ns \leq 2147483647$</td>
</tr>
<tr>
<td>CLOB(n)</td>
<td>SQL TYPE IS CLOB(n)</td>
<td>$1ns \leq 2147483647$</td>
</tr>
<tr>
<td>DBCLOB(n)</td>
<td>SQL TYPE IS DBCLOB(n)</td>
<td>$n$ is the number of double-byte characters. $1ns \leq 1073741823$</td>
</tr>
<tr>
<td>XML</td>
<td>SQL TYPE IS XML AS BLOB(n)</td>
<td>$1ns \leq 2147483647$</td>
</tr>
<tr>
<td>XML</td>
<td>SQL TYPE IS XML AS CLOB(n)</td>
<td>$1ns \leq 2147483647$</td>
</tr>
<tr>
<td>XML</td>
<td>SQL TYPE IS XML AS DBCLOB(n)</td>
<td>$n$ is the number of double-byte characters. $1ns \leq 1073741823$</td>
</tr>
<tr>
<td>BLOB file reference</td>
<td>SQL TYPE IS BLOB_FILE</td>
<td>Use this data type only to manipulate data in BLOB columns. Do not use this data type as a column type.</td>
</tr>
<tr>
<td>CLOB file reference</td>
<td>SQL TYPE IS CLOB_FILE</td>
<td>Use this data type only to manipulate data in CLOB columns. Do not use this data type as a column type.</td>
</tr>
<tr>
<td>DBCLOB file reference</td>
<td>SQL TYPE IS DBCLOB_FILE</td>
<td>Use this data type only to manipulate data in DBCLOB columns. Do not use this data type as a column type.</td>
</tr>
<tr>
<td>XML BLOB file reference</td>
<td>SQL TYPE IS XML AS BLOB_FILE</td>
<td>Use this data type only to manipulate XML data as BLOB files. Do not use this data type as a column type.</td>
</tr>
<tr>
<td>XML CLOB file reference</td>
<td>SQL TYPE IS XML AS CLOB_FILE</td>
<td>Use this data type only to manipulate XML data as CLOB files. Do not use this data type as a column type.</td>
</tr>
</tbody>
</table>
Table 109. C host variable equivalents that you can use when retrieving data of a particular SQL data type (continued)

<table>
<thead>
<tr>
<th>SQL data type</th>
<th>C host variable equivalent</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>XML DBCLOB file reference</td>
<td>SQL TYPE IS XML AS DBCLOB_FILE</td>
<td>Use this data type only to manipulate XML data as DBCLOB files. Do not use this data type as a column type.</td>
</tr>
<tr>
<td>ROWID</td>
<td>SQL TYPE IS ROWID</td>
<td></td>
</tr>
</tbody>
</table>

**Related concepts**

Compatibility of SQL and language data types
The host variable data types that are used in SQL statements must be compatible with the data types of the columns with which you intend to use them.

LOB host variable, LOB locator, and LOB file reference variable declarations
When you write applications to manipulate LOB data, you need to declare host variables to hold the LOB data or LOB locator. Alternatively, you need to declare LOB file reference variables to point to the LOB data.

Host variable data types for XML data in embedded SQL applications (Db2 Programming for XML)

**Handling SQL error codes in C and C++ applications**

C and C++ applications can request more information about SQL error codes by using the DSNTIAR subroutine or issuing a GET DIAGNOSTICS statement.

**Procedure**

To request information about SQL errors in C and C++ applications, use the following approaches:

- You can use the subroutine DSNTIAR to convert an SQL return code into a text message.

  DSNTIAR takes data from the SQLCA, formats it into a message, and places the result in a message output area that you provide in your application program. For concepts and more information about the behavior of DSNTIAR, see “Displaying SQLCA fields by calling DSNTIAR” on page 529.

**DSNITAR syntax**

DSNITAR has the following syntax:

\[
\text{rc} = \text{DSNITAR}(&\text{sqlca}, &\text{message}, &\text{lrecl});
\]

**Parameters for DSNITAR**

The DSNITAR parameters have the following meanings:

- **&sqlca**
  An SQL communication area.

- **&message**
  An output area, in VARCHAR format, in which DSNITAR places the message text. The first halfword contains the length of the remaining area; its minimum value is 240.

  The output lines of text, each line being the length specified in &lrecl, are put into this area. For example, you could specify the format of the output area as:

```c
#define data_len 132
#define data_dim 10
int length_of_line = data_len;
struct error_struct {
  short int error_len;
  char error_text[data_dim][data_len];
} error_message = {data_dim * data_len};
rc = DSNITAR(&sqlca, &error_message, &length_of_line);
```
where error_message is the name of the message output area, data_dim is the number of lines in the message output area, and data_len is the length of each line.

&lrecl
A fullword containing the logical record length of output messages, between 72 and 240.

To inform your compiler that DSNTIAR is an assembler language program, include one of the following statements in your application.

For C, include:

```c
#pragma linkage (DSNTIAR,OS)
```

For C++, include a statement similar to this:

```c
extern "OS" short int DSNTIAR(struct sqlca *sqlca,
    struct error_struct *error_message,
    int *data_len);
```

Examples of calling DSNTIAR from an application appear in the Db2 sample C program DSN8BD3 and in the sample C++ program DSN8BE3. Both are in the library DSN8C10.SDSNSAMP. See “Sample applications supplied with Db2 for z/OS” on page 1024 for instructions on how to access and print the source code for the sample programs.

- If your CICS application requires CICS storage handling, you must use the subroutine DSNTIAC instead of DSNTIAR.

**DSNTIAC syntax**

DSNTIAC has the following syntax:

```
rc = DSNTIAC(&eib, &commarea, &sqlca, &message, &lrecl);
```

**Parameters for DSNTIAC**

DSNTIAC has extra parameters, which you must use for calls to routines that use CICS commands.

- **&eib**
  EXEC interface block

- **&commarea**
  communication area

For more information on these parameters, see the appropriate application programming guide for CICS. The remaining parameter descriptions are the same as those for DSNTIAR. Both DSNTIAC and DSNTIAR format the SQLCA in the same way.

You must define DSNTIA1 in the CSD. If you load DSNTIAR or DSNTIAC, you must also define them in the CSD. For an example of CSD entry generation statements for use with DSNTIAC, see job DSNTEJ5A.

The assembler source code for DSNTIAC and job DSNTEJ5A, which assembles and link-edits DSNTIAC, are in the data set `prefix.SDSNSAMP`.

- You can also use the MESSAGE_TEXT condition item field of the GET DIAGNOSTICS statement to convert an SQL return code into a text message.

Programs that require long token message support should code the GET DIAGNOSTICS statement instead of DSNTIAR. For more information about GET DIAGNOSTICS, see “Checking the execution of SQL statements by using the GET DIAGNOSTICS statement ” on page 534.

**Related tasks**
Handling SQL error codes
Application programs can request more information about SQL error codes from Db2.

**Related reference**
GET DIAGNOSTICS (Db2 SQL)
COBOL applications that issue SQL statements

You can code SQL statements in certain COBOL program sections. The allowable sections are shown in the following table.

<table>
<thead>
<tr>
<th>SQL statement</th>
<th>Program section</th>
</tr>
</thead>
<tbody>
<tr>
<td>BEGIN DECLARE SECTION</td>
<td>WORKING-STORAGE SECTION(^1) or LINKAGE SECTION</td>
</tr>
<tr>
<td>END DECLARE SECTION</td>
<td></td>
</tr>
<tr>
<td>INCLUDE SQLCA</td>
<td>WORKING-STORAGE SECTION(^1) or LINKAGE SECTION</td>
</tr>
<tr>
<td>INCLUDE text-file-name</td>
<td>PROCEDURE DIVISION or DATA DIVISION(^2)</td>
</tr>
<tr>
<td>DECLARE TABLE</td>
<td>DATA DIVISION or PROCEDURE DIVISION</td>
</tr>
<tr>
<td>DECLARE CURSOR</td>
<td></td>
</tr>
<tr>
<td>DECLARE VARIABLE</td>
<td>WORKING-STORAGE SECTION(^1)</td>
</tr>
<tr>
<td>Other</td>
<td>PROCEDURE DIVISION</td>
</tr>
</tbody>
</table>

**Notes:**

1. If you use the Db2 coprocessor, you can use the LOCAL-STORAGE SECTION wherever WORKING-STORAGE SECTION is listed in the table.
2. When including host variable declarations, the INCLUDE statement must be in the WORKING-STORAGE SECTION or the LINKAGE SECTION.

You cannot put SQL statements in the DECLARATIVES section of a COBOL program.

Each SQL statement in a COBOL program must begin with EXEC SQL and end with END-EXEC. If you are using the Db2 precompiler, the EXEC and SQL keywords must appear on one line, but the remainder of the statement can appear on subsequent lines. If you are using the Db2 coprocessor, the EXEC and SQL keywords can be on different lines. Do not include any tokens between the two keywords EXEC and SQL except for COBOL comments, including debugging lines. Do not include SQL comments between the keywords EXEC and SQL.

If the SQL statement appears between two COBOL statements, the period after END-EXEC is optional and might not be appropriate. If the statement appears in an IF...THEN set of COBOL statements, omit the ending period to avoid inadvertently ending the IF statement.

You might code an UPDATE statement in a COBOL program as follows:

```sql
EXEC SQL
    UPDATE DSN8C10.DEPT
    SET MGRNO = :MGR-NUM
    WHERE DEPTNO = :INT-DEPT
END-EXEC.
```

**Comments**

You can include COBOL comment lines (* in column 7) in SQL statements wherever you can use a blank.

Also, you can include SQL comments (‘ --’) in any embedded SQL statement. A space must precede the two hyphens (‘ --’) that begin the comment. For more information, see SQL comments (Db2 SQL).

**Restrictions:** If you are using the Db2 precompiler, be aware of the following restrictions:
• You cannot include COBOL comment lines between the keywords EXEC and SQL. The precompiler treats COBOL debugging lines and page-eject lines (/ in column 7) as comment lines. The Db2 coprocessor treats the debugging lines based on the COBOL rules, which depend on the WITH DEBUGGING mode setting.

• You cannot use COBOL inline comments that are identified by a floating comment indicator (*>). COBOL inline comments are interpreted correctly only when the Db2 coprocessor is used.

For an SQL INCLUDE statement, the Db2 precompiler treats any text that follows the period after END-EXEC, and on the same line as END-EXEC, as a comment. The Db2 coprocessor treats this text as part of the COBOL program syntax.

Debugging lines
The Db2 precompiler ignores the 'D' in column 7 on debugging lines and treats it as a blank. The Db2 coprocessor follows the COBOL language rules regarding debugging lines.

Continuation for SQL statements
The rules for continuing a character string constant from one line to the next in an SQL statement embedded in a COBOL program are the same as those for continuing a non-numeric literal in COBOL. However, you can use either a quote or an apostrophe as the first nonblank character in area B of the continuation line. The same rule applies for the continuation of delimited identifiers and does not depend on the string delimiter option.

To conform with SQL standard, delimit a character string constant with an apostrophe, and use a quote as the first nonblank character in area B of the continuation line for a character string constant.

Continued lines of an SQL statement can be in columns 8 through 72 when using the Db2 precompiler and columns 12 through 72 when using the Db2 coprocessor.

Delimiters
Delimit an SQL statement in your COBOL program with the beginning keyword EXEC SQL and an END-EXEC as shown in the following example code:

```cobol
EXEC SQL
  SQL-statement
END-EXEC.
```

COPY
If you use the Db2 precompiler, do not use a COBOL COPY statement within host variable declarations. If you use the Db2 coprocessor, you can use COBOL COPY.

REPLACE
If you use the Db2 precompiler, the REPLACE statement has no effect on SQL statements. It affects only the COBOL statements that the precompiler generates.

If you use the Db2 coprocessor, the REPLACE statement replaces text strings in SQL statements as well as in generated COBOL statements.

Declaring tables and views
Your COBOL program should include the statement DECLARE TABLE to describe each table and view the program accesses. You can use the Db2 declarations generator (DCLGEN) to generate the DECLARE TABLE statements. You should include the DCLGEN members in the DATA DIVISION.

Dynamic SQL in a COBOL program
In general, COBOL programs can easily handle dynamic SQL statements. COBOL programs can handle SELECT statements if the data types and the number of fields returned are fixed. If you want to use variable-list SELECT statements, use an SQLDA.

Including code
To include SQL statements or COBOL host variable declarations from a member of a partitioned data set, use the following SQL statement in the source code where you want to include the statements:

```cobol
EXEC SQL INCLUDE member-name END-EXEC.
```

If you are using the Db2 precompiler, you cannot nest SQL INCLUDE statements. In this case, do not use COBOL verbs to include SQL statements or host variable declarations, and do not use the SQL
INCLUDE statement to include CICS preprocessor related code. In general, if you are using the Db2 precompiler, use the SQL INCLUDE statement only for SQL-related coding. If you are using the COBOL Db2 coprocessor, none of these restrictions apply.

Use the 'EXEC SQL' and 'END-EXEC' keyword pair to include SQL statements only. COBOL statements, such as COPY or REPLACE, are not allowed.

Margins
You must code SQL statements that begin with EXEC SQL in columns 12 through 72. Otherwise the Db2 precompiler does not recognize the SQL statement.

Names
You can use any valid COBOL name for a host variable. Do not use external entry names or access plan names that begin with 'DSN', and do not use host variable names that begin with 'SQL'. These names are reserved for Db2.

Sequence numbers
The source statements that the Db2 precompiler generates do not include sequence numbers.

Statement labels
You can precede executable SQL statements in the PROCEDURE DIVISION with a paragraph name.

WHenever statement
The target for the GOTO clause in an SQL statement WHENEVER must be a section name or unqualified paragraph name in the PROCEDURE DIVISION.

Special COBOL considerations
The following considerations apply to programs written in COBOL:

• In a COBOL program that uses elements in a multi-level structure as host variable names, the Db2 precompiler generates the lowest two-level names.

• Using the COBOL compiler options DYNAM and NODYNAM depends on the operating environment.

TSO and IMS: You can specify the option DYNAM when compiling a COBOL program if you use the following guidelines. IMS and Db2 share a common alias name, DSNHLI, for the language interface module. You must do the following when you concatenate your libraries:
   - If you use IMS with the COBOL option DYNAM, be sure to concatenate the IMS library first.
   - If you run your application program only under , be sure to concatenate the Db2 library first.

CICS, CAF, and RRSAF: You must specify the NODYNAM option when you compile a COBOL program that either includes CICS statements or is translated by a separate CICS translator or the integrated CICS translator. In these cases, you cannot specify the DYNAM option. If your CICS program has a subroutine that is not translated by a separate CICS translator or the integrated CICS translator but contains SQL statements, you can specify the DYNAM option. However, in this case, you must concatenate the CICS libraries before the Db2 libraries.

You can compile COBOL stored procedures with either the DYNAM option or the NODYNAM option. If you use DYNAM, ensure that the correct Db2 language interface module is loaded dynamically by performing one of the following actions:
   - Use the ATTACH(RRSAF) precompiler option.
   - Copy the DSNRLI module into a load library that is concatenated in front of the Db2 libraries. Use the member name DSNHLI.

• To avoid truncating numeric values, use either of the following methods:
   - Use the COMP-5 data type for binary integer host variables.
   - Specify the COBOL compiler option:
     - TRUNC(OPT) if you are certain that the data being moved to each binary variable by the application does not have a larger precision than is defined in the PICTURE clause of the binary variable.
     - TRUNC(BIN) if the precision of data being moved to each binary variable might exceed the value in the PICTURE clause.
Db2 assigns values to binary integer host variables as if you had specified the COBOL compiler option TRUNC(BIN) or used the COMP-5 data type.

- If you are using the Db2 precompiler and your COBOL program contains several entry points or is called several times, the USING clause of the entry statement that executes before the first SQL statement executes must contain the SQLCA and all linkage section entries that any SQL statement uses as host variables.
- If you use the Db2 precompiler, no compiler directives should appear between the PROCEDURE DIVISION and the DECLARATIVES statement.
- Do not use COBOL figurative constants (such as ZERO and SPACE), symbolic characters, reference modification, and subscripts within SQL statements.
- Observe the rules for naming SQL identifiers, as described in SQL identifiers (Db2 SQL). However, for COBOL only, the names of SQL identifiers can follow the rules for naming COBOL words, as described in COBOL words with single-byte characters (COBOL) (Enterprise COBOL for z/OS Programming Guide). However, the names must not exceed the allowable length for the Db2 object.
- Surround hyphens used as subtraction operators with spaces. Db2 usually interprets a hyphen with no spaces around it as part of a host variable name.
- If you include an SQL statement in a COBOL PERFORM ... THRU paragraph and also specify the SQL statement WHENEVER ... GO, the COBOL compiler returns the warning message IGYOP3094. That message might indicate a problem. This usage is not recommended.
- If you are using the Db2 precompiler, all SQL statements and any host variables they reference must be within the first program when using nested programs or batch compilation.
- If you are using the Db2 precompiler, your COBOL programs must have a DATA DIVISION and a PROCEDURE DIVISION. Both divisions and the WORKING-STORAGE SECTION must be present in programs that contain SQL statements. However, if your COBOL programs requires the LOCAL-STORAGE SECTION, then the Db2 coprocessor should be used instead of the Db2 precompiler.
- The Db2 precompiler generates this COBOL variable:

  DSN-TMP2 PIC S9(18) COMP-3

The Db2 coprocessor generates this COBOL variable:

  SQL---SCRVALD DS 10P PIC S9(18) COMP-3

If you specify COBOL option RULES(NOEVENPACK), the COBOL compiler generates warning IGYDS1348-W, because those variables have an even number of packed decimal digits.

PSPI If your program uses the Db2 precompiler and uses parameters that are defined in LINKAGE SECTION as host variables to Db2 and the address of the input parameter might change on subsequent invocations of your program, your program must reset the variable SQL-INIT-FLAG. This flag is generated by the Db2 precompiler. Resetting this flag indicates that the storage must initialize when the next SQL statement executes. To reset the flag, insert the statement MOVE ZERO TO SQL-INIT-FLAG in the called program’s PROCEDURE DIVISION, ahead of any executable SQL statements that use the host variables. If you use the COBOL Db2 coprocessor, the called program does not need to reset SQL-INIT-FLAG. PSPI

Handling SQL error codes
Cobol applications can request more information about SQL errors from Db2. For more information, see “Handling SQL error codes in Cobol applications” on page 683.

Related concepts
Sample applications supplied with Db2 for z/OS
Db2 provides sample applications to help you with Db2 programming techniques and coding practices within each of the four environments: batch, TSO, IMS, and CICS. The sample applications contain various applications that might apply to managing a company.

DCLGEN (declarations generator)
Your program should declare the tables and views that it accesses. The Db2 declarations generator, DCLGEN, produces these DECLARE statements for C, COBOL, and PL/I programs, so that you do not need to code the statements yourself. DCLGEN also generates corresponding host variable structures.

Using host-variable arrays in SQL statements
Use host-variable arrays in embedded SQL statements to represent values that the program does not know until the query is executed. Host-variable arrays are useful for storing a set of retrieved values or for passing a set of values that are to be inserted into a table.

SQL identifiers (Db2 SQL)

Related tasks
Overview of programming applications that access Db2 for z/OS data
Applications that interact with Db2 must first connect to Db2. They can then read, add, or modify data or manipulate Db2 objects.

Including dynamic SQL in your program
Dynamic SQL is prepared and executed while the program is running.

Checking the execution of SQL statements by using the GET DIAGNOSTICS statement
One way to check whether an SQL statement executed successfully is to ask Db2 to return the diagnostic information about the last executed SQL statement.

Defining SQL descriptor areas (SQLDA)
If your program includes certain SQL statements, you must define at least one SQL descriptor area (SQLDA). Depending on the context in which it is used, the SQLDA stores information about prepared SQL statements or host variables. This information can then be read by either the application program or Db2.

Displaying SQLCA fields by calling DSNTIAR
If you use the SQLCA to check whether an SQL statement executed successfully, your program needs to read the data in the appropriate SQLCA fields. One easy way to read these fields is to use the assembler subroutine DSNTIAR.

Setting limits for system resource usage by using the resource limit facility (Db2 Performance)

COBOL programming examples

You can write Db2 programs in COBOL. These programs can access a local or remote Db2 subsystem and can execute static or dynamic SQL statements. This information contains several such programming examples.

To prepare and run these applications, use the JCL in prefix.SDSNSAMP as a model for your JCL.

Related reference
Assembler, C, C++, COBOL, PL/I, and REXX programming examples (Db2 Programming samples)

Sample COBOL dynamic SQL program

You can code dynamic varying-list SELECT statements in a COBOL program. Varying-List SELECT statements are statements for which you do not know the number or data types of columns that are to be returned when you write the program.

Introductory concepts

Submitting SQL statements to Db2 (Introduction to Db2 for z/OS)
Dynamic SQL applications (Introduction to Db2 for z/OS)

“Including dynamic SQL in your program” on page 496 describes three variations of dynamic SQL statements:

- Non-SELECT statements
- Fixed-List SELECT statements

In this case, you know the number of columns returned and their data types when you write the program.
• Varying-List SELECT statements.

In this case, you do not know the number of columns returned and their data types when you write the program.

This section documents a technique of coding varying list SELECT statements in COBOL. This example program does not support BLOB, CLOB, or DBCLOB data types.

**Pointers and based variables in the sample COBOL program**

COBOL has a POINTER type and a SET statement that provide pointers and based variables.

The SET statement sets a pointer from the address of an area in the linkage section or another pointer; the statement can also set the address of an area in the linkage section. UNLDBCU2 in “Example of the sample COBOL program” on page 624 provides these uses of the SET statement. The SET statement does not permit the use of an address in the WORKING-STORAGE section.

**Storage allocation for the sample COBOL program**

COBOL does not provide a means to allocate main storage within a program. You can achieve the same end by having an initial program which allocates the storage, and then calls a second program that manipulates the pointer. (COBOL does not permit you to directly manipulate the pointer because errors and abends are likely to occur.)

The initial program is extremely simple. It includes a working storage section that allocates the maximum amount of storage needed. This program then calls the second program, passing the area or areas on the CALL statement. The second program defines the area in the linkage section and can then use pointers within the area.

If you need to allocate parts of storage, the best method is to use indexes or subscripts. You can use subscripts for arithmetic and comparison operations.

**Example of the sample COBOL program**

The following example shows an example of the initial program UNLDBCU1 that allocates the storage and calls the second program UNLDBCU2. UNLDBCU2 then defines the passed storage areas in its linkage section and includes the USING clause on its PROCEDURE DIVISION statement.

Defining the pointers, then redefining them as numeric, permits some manipulation of the pointers that you cannot perform directly. For example, you cannot add the column length to the record pointer, but you can add the column length to the numeric value that redefines the pointer.

The following example is the initial program that allocates storage.

```cobol
**** UNLDBCU1- DB2 SAMPLE BATCH COBOL UNLOAD PROGRAM ************
* MODULE NAME = UNLDBCU1
* DESCRIPTIVE NAME = DB2 SAMPLE APPLICATION
* UNLOAD PROGRAM
* BATCH
* IBM ENTERPRISE COBOL FOR Z/OS
* COPYRIGHT = 5740-XYR (C) COPYRIGHT IBM CORP 1982, 1987
* REFER TO COPYRIGHT INSTRUCTIONS FORM NUMBER G120-2083
* STATUS = VERSION 1 RELEASE 3, LEVEL 0
* FUNCTION = THIS MODULE PROVIDES THE STORAGE NEEDED BY
* UNLDBCU2 AND CALLS THAT PROGRAM.
* NOTES =
* DEPENDENCIES = ENTERPRISE COBOL FOR Z/OS IS REQUIRED.
* SEVERAL NEW FACILITIES ARE USED.
* RESTRICTIONS =
* THE MAXIMUM NUMBER OF COLUMNS IS 750,
* WHICH IS THE SQL LIMIT.
```
DATA RECORDS ARE LIMITED TO 32700 BYTES, INCLUDING DATA, LENGTHS FOR VARCHAR DATA, AND SPACE FOR NULL INDICATORS.

MODULE TYPE = IBM ENTERPRISE COBOL PROGRAM
PROCESSOR = ENTERPRISE COBOL FOR Z/OS
MODULE SIZE = SEE LINK EDIT
ATTRIBUTES = REENTRANT

ENTRY POINT = UNLDBCU1
PURPOSE = SEE FUNCTION
LINKAGE = INVOKED FROM DSN RUN
INPUT = NONE
OUTPUT = NONE

EXIT-NORMAL = RETURN CODE 0 NORMAL COMPLETION
EXIT-ERROR =
RETURN CODE = NONE
ABEND CODES = NONE
ERROR-MESSAGES = NONE

EXTERNAL REFERENCES =
ROUTINES/SERVICES =
UNLDBCU2 - ACTUAL UNLOAD PROGRAM

DATA-AREAS = NONE
CONTROL-BLOCKS = NONE

TABLES = NONE
CHANGE-ACTIVITY = NONE

*PSEUDOCODE*

PROCEDURE
CALL UNLDBCU2.
END.

IDENTIFICATION DIVISION.
PROGRAM-ID. UNLDBCU1
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
DATA DIVISION.
WORKING-STORAGE SECTION.
01 WORKAREA-IND.
  02 WORKIND PIC S9(4) COMP OCCURS 750 TIMES.
  01 RECWORK.
    02 RECWORK-LEN PIC S9(8) COMP VALUE 32700.
    02 RECWORK-CHAR PIC X(1) OCCURS 32700 TIMES.
PROCEDURE DIVISION.
  CALL 'UNLDBCU2' USING WORKAREA-IND RECWORK.
  GOBACK.

The following example is the called program that does pointer manipulation.

**** UNLDBCU2- DB2 SAMPLE BATCH COBOL UNLOAD PROGRAM ***********
MODULE NAME = UNLDBCU2
DESCRIPTIVE NAME = DB2 SAMPLE APPLICATION
UNLOAD PROGRAM
BATCH
ENTERPRISE COBOL FOR Z/OS
COPYRIGHT = 5740-XYR (C) COPYRIGHT IBM CORP 1982, 1987
REFER TO COPYRIGHT INSTRUCTIONS FORM NUMBER G120-2083
STATUS = VERSION 1 RELEASE 3, LEVEL 0
FUNCTION = THIS MODULE ACCEPTS A TABLE NAME OR VIEW NAME AND UNLOADS THE DATA IN THAT TABLE OR VIEW.
* READ IN A TABLE NAME FROM SYSIN.  
* PUT DATA FROM THE TABLE INTO DD SYSREC01.  
* WRITE RESULTS TO SYSPRINT.  
*  
* NOTES =  
* DEPENDENCIES = IBM ENTERPRISE COBOL FOR Z/OS  
* IS REQUIRED.  
*  
* RESTRICTIONS =  
* THE SQLDA IS LIMITED TO 33016 BYTES.  
* THIS SIZE ALLOWS FOR THE DB2 MAXIMUM  
* OF 750 COLUMNS.  
*  
* DATA RECORDS ARE LIMITED TO 32700 BYTES,  
* INCLUDING DATA, LENGTHS FOR VARCHAR DATA,  
* AND SPACE FOR NULL INDICATORS.  
*  
* TABLE OR VIEW NAMES ARE ACCEPTED, AND ONLY  
* ONE NAME IS ALLOWED PER RUN.  
*  
* MODULE TYPE = ENTERPRISE COBOL FOR Z/OS  
* PROCESSOR = DB2 PRECOMPLIER, COBOL COMPILER  
* MODULE SIZE = SEE LINK EDIT  
* ATTRIBUTES = REENTRANT  
*  
* ENTRY POINT = UNLDBC02  
* PURPOSE = SEE FUNCTION  
* LINKAGE =  
* CALL 'UNLDBC02' USING WORKAREA-IND RECWORK.  
*  
* INPUT = SYMBOLIC LABEL/NAME = WORKAREA-IND  
* DESCRIPTION = INDICATOR VARIABLE ARRAY  
* 01 WORKAREA-IND.  
* 02 WORKIND PIC S9(4) COMP OCCURS 750 TIMES.  
*  
* SYMBOLIC LABEL/NAME = REWORK  
* DESCRIPTION = WORK AREA FOR OUTPUT RECORD  
* 01 REWORK.  
* 02 REWORK-LEN PIC S9(8) COMP.  
* 02 REWORK-CHAR PIC X(1) OCCURS 32700 TIMES.  
*  
* SYMBOLIC LABEL/NAME = SYSIN  
* DESCRIPTION = INPUT REQUESTS - TABLE OR VIEW  
*  
* OUTPUT = SYMBOLIC LABEL/NAME = SYSPRINT  
* DESCRIPTION = PRINTED RESULTS  
*  
* SYMBOLIC LABEL/NAME = SYSREC01  
* DESCRIPTION = UNLOADED TABLE DATA  
*  
* EXIT-NORMAL = RETURN CODE 0 NORMAL COMPLETION  
* EXIT-ERROR =  
* RETURN CODE = NONE  
* ABEND CODES = NONE  
* ERROR-MESSAGES =  
* DSNT4901 SAMPLE COBOL DATA UNLOAD PROGRAM RELEASE 3.0  
* - THIS IS THE HEADER, INDICATING A NORMAL  
* - START FOR THIS PROGRAM.  
*  
* DSNT4931 SQL ERROR, SQLCODE = NNNNNNNNN  
* - AN SQL ERROR OR WARNING WAS ENCOUNTERED  
* - ADDITIONAL INFORMATION FROM DSNTIAR  
* - FOLLOWS THIS MESSAGE.  
*  
* DSNT4951 SUCCESSFUL UNLOAD XXXXXXXX ROWS OF  
* TABLE TTTTTTTTTT  
* - THE UNLOAD WAS SUCCESSFUL. XXXXXXXX IS  
* - THE NUMBER OF ROWS UNLOADED. TTTTTTTTT  
* - IS THE NAME OF THE TABLE OR VIEW FROM  
* - WHICH IT WAS UNLOADED.  
*  
* DSNT4961 UNRECOGNIZED DATA TYPE CODE OF NNNNNNNNN  
* - THE PREPARE RETURNED AN INVALID DATA  
* - TYPE CODE. NNNNNNNNN IS THE CODE, PRINTED  
* - IN DECIMAL. USUALLY AN ERROR IN  
* - THIS ROUTINE OR A NEW DATA TYPE.  
*  
* DSNT4971 RETURN CODE FROM MESSAGE ROUTINE DSNTIAR  
* - THE MESSAGE FORMATTING ROUTINE DETECTED  
* - AN ERROR. SEE THAT ROUTINE FOR RETURN  
* - CODE INFORMATION. USUALLY AN ERROR IN  
* - THIS ROUTINE.  
*  
* DSNT4981 ERROR, NO VALID COLUMNS FOUND  
* - THE PREPARE RETURNED DATA WHICH DID NOT  
* - PRODUCE A VALID OUTPUT RECORD.  
* - USUALLY AN ERROR IN THIS ROUTINE.  

626  Db2 12 for z/OS: Application Programming and SQL Guide
DSNT499I NO ROWS FOUND IN TABLE OR VIEW
- THE CHOSEN TABLE OR VIEWS DID NOT RETURN ANY ROWS.
ERROR MESSAGES FROM MODULE DSNTIAR
- WHEN AN ERROR OCCURS, THIS MODULE PRODUCES CORRESPONDING MESSAGES.
OTHER MESSAGES:
- THE TABLE COULD NOT BE UNLOADED. EXITING.
EXTERNAL REFERENCES =
ROUTINES/SERVICES =
- DSNTIAR - TRANSLATE SQLCA INTO MESSAGES
DATA-AREAS = NONE
CONTROL-BLOCKS =
- SQLCA - SQL COMMUNICATION AREA
TABLES = NONE
CHANGE-ACTIVITY = NONE

*PSEUDOCODE*
PROCEDURE
  EXEC SQL DECLARE DT CURSOR FOR SEL END-EXEC.
  EXEC SQL DECLARE SEL STATEMENT END-EXEC.
  INITIALIZE THE DATA, OPEN FILES.
  OBTAIN STORAGE FOR THE SQLDA AND THE DATA RECORDS.
  READ A TABLE NAME.
  OPEN SYSREC01.
  BUILD THE SQL STATEMENT TO BE EXECUTED
  EXEC SQL PREPARE SQL STATEMENT INTO SQLDA END-EXEC.
  SET UP ADDRESSES IN THE SQLDA FOR DATA.
  INITIALIZE DATA RECORD COUNTER TO 0.
  EXEC SQL OPEN DT END-EXEC.
  DO WHILE SQLCODE IS 0.
  EXEC SQL FETCH DT USING DESCRIPTOR SQLDA END-EXEC.
  ADD IN MARKERS TO DENOTE NULLS.
  WRITE THE DATA TO SYSREC01.
  INCREMENT DATA RECORD COUNTER.
  END.
  EXEC SQL CLOSE DT END-EXEC.
  INDICATE THE RESULTS OF THE UNLOAD OPERATION.
  CLOSE THE SYSIN, SYSPRINT, AND SYSREC01 FILES.
END.

IDENTIFICATION DIVISION.
--------------
PROGRAM-ID. UNLDBC2
--------------
ENVIRONMENT DIVISION.
--------------
CONFIGURATION SECTION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
  SELECT SYSIN
  ASSIGN TO DA-S-SYSIN.
  SELECT SYSPRINT
  ASSIGN TO UT-S-SYSPRINT.
  SELECT SYSREC01
  ASSIGN TO DA-S-SYSREC01.

DATA DIVISION.
----------
FILE SECTION.
FD SYSIN
  RECORD CONTAINS 80 CHARACTERS
  BLOCK CONTAINS 0 RECORDS
  LABEL RECORDS ARE OMITTED
  RECORDING MODE IS F.
  01 CARDREC PIC X(80).
FD SYSPRINT
  RECORD CONTAINS 120 CHARACTERS
  LABEL RECORDS ARE OMITTED
  DATA RECORD IS MSGREC
  RECORDING MODE IS F.
  01 MSGREC PIC X(120).
FD SYSREC01
  RECORD CONTAINS 5 TO 32704 CHARACTERS
  LABEL RECORDS ARE OMITTED
  DATA RECORD IS REC01

Chapter 4. Embedded SQL programming 627
RECORDING MODE IS V.

01 REC01.
   02 REC01-LEN PIC S9(8) COMP.
   02 REC01-CHAR PIC X(1) OCCURS 1 TO 32700 TIMES
       DEPENDING ON REC01-LEN.
/

' WORKING-STORAGE SECTION.
*  ******************************************************
*  STRUCTURE FOR INPUT                                *
*  ******************************************************
01 IOAREA.
   02 TNAME         PIC X(72).
   02 FILLER        PIC X(08).
01 STMTBUF.
   49 STMTLEN       PIC S9(4) COMP VALUE 92.
   49 STMTCHAR      PIC X(92).
01 STMTBLD.
   02 FILLER        PIC X(20) VALUE 'SELECT * FROM'.
   02 STMTTAB      PIC X(72).
*  ******************************************************
*  REPORT HEADER STRUCTURE                            *
*  ******************************************************
01 HEADER.
   02 FILLER PIC X(35)
       VALUE ' DSNT490I SAMPLE COBOL DATA UNLOAD '.
   02 FILLER PIC X(85) VALUE 'PROGRAM RELEASE 3.0'.
01 MSG- SQLErr.
   02 FILLER PIC X(31)
       VALUE ' DSNT493I SQL ERROR, SQLCODE = '.
   02 MSG-MINUS PIC X(1).
   02 MSG-PRINT-CODE PIC 9(8).
   02 FILLER PIC X(81) VALUE '.
01 MSG-OTHER- ERR.
   02 FILLER PIC X(42)
       VALUE ' THE TABLE COULD NOT BE UNLOADED. EXITING.'.
   02 FILLER PIC X(78) VALUE '.
01 UNLOADED.
   02 FILLER PIC X(28)
       VALUE ' DSNT495I SUCCESSFUL UNLOAD '.
   02 ROWS PIC 9(8).
   02 FILLER PIC X(15) VALUE ' ROWS OF TABLE '.
   02 TABLENAM PIC X(72) VALUE '.
01 BADTYPE.
   02 FILLER PIC X(42)
       VALUE ' DSNT496I UNRECOGNIZED DATA TYPE CODE OF '.
   02 TYPCOD PIC 9(8).
   02 FILLER PIC X(71) VALUE '.
01 MSGRETCD.
   02 FILLER PIC X(42)
       VALUE ' DSNT497I RETURN CODE FROM MESSAGE ROUTINE'.
   02 FILLER PIC X(9) VALUE 'DSNTIAR '.
   02 RETCODE    PIC 9(8).
   02 FILLER PIC X(62) VALUE '.
01 MSGMNOCOL.
   02 FILLER PIC X(120)
       VALUE ' DSNT498I ERROR, NO VALID COLUMNS FOUND'.
01 MSG-NOROW.
   02 FILLER PIC X(120)
       VALUE ' DSNT499I NO ROWS FOUND IN TABLE OR VIEW'.
*  WORKAREAS                                          *
*  ******************************************************
77 NOT-FOUND         PIC S9(8) COMP VALUE +100.
*  ******************************************************
*  VARIABLES FOR ERROR-MESSAGE FORMATTING             *
*  ******************************************************
01 ERROR-MESSAGE.
   02 ERROR-LEN PIC S9(4) COMP VALUE +960.
   02 ERROR-TEXT PIC X(120) OCCURS 8 TIMES
       Indexed by ERROR-INDEX.
77 ERROR-TEXT-LEN    PIC S9(8) COMP VALUE +120.
*  DESCRIPTOR AREA                                      *
*  ******************************************************
01 SQLDA.
   02 SQLDAID PIC X(8) VALUE 'SQLDA '.
   02 SQLDABC PIC S9(8) COMPUTATIONAL VALUE 33016.
   02 SQLN PIC S9(4) COMPUTATIONAL VALUE 750.
   02 SQLD PIC S9(4) COMPUTATIONAL VALUE 0.
   02 SQLVAR OCCURS 1 TO 750 TIMES
DEPENDING ON SQLN.
03 SQLTYPE PIC S9(4) COMPUTATIONAL.
03 SQLLEN PIC S9(4) COMPUTATIONAL.
03 SQLDATA POINTER.
03 SQLIND POINTER.
03 SQLNAME.
  49 SQLNAMEL PIC S9(4) COMPUTATIONAL.
  49 SQLNAMEC PIC X(30).
* DATA TYPES FOUND IN SQLTYPE, AFTER REMOVING THE NULL BIT *
77 VARCTYPE PIC S9(4) COMP VALUE +448.
77 CHARTYPE PIC S9(4) COMP VALUE +452.
77 VARCHARS PIC S9(4) COMP VALUE +456.
77 GARYTYPE PIC S9(4) COMP VALUE +464.
77 GTYPE PIC S9(4) COMP VALUE +468.
77 LVARGETYP PIC S9(4) COMP VALUE +472.
77 FLOATYPE PIC S9(4) COMP VALUE +480.
77 DECNUM PIC S9(4) COMP VALUE +484.
77 INTTYPE PIC S9(4) COMP VALUE +496.
77 HWTYPE PIC S9(4) COMP VALUE +500.
77 DATETYP PIC S9(4) COMP VALUE +384.
77 TIMETYP PIC S9(4) COMP VALUE +388.
77 TIMESMP PIC S9(4) COMP VALUE +392.
*
01 RECPRTR POINTER.
01 RECNUM REDEFINES RECPRTR PICTURE S9(8) COMPUTATIONAL.
01 IRECPTR POINTER.
01 IRECNEM REDEFINES IRECPTR PICTURE S9(8) COMPUTATIONAL.
01 IPicture S9(4) COMPUTATIONAL.
01 J Picture S9(4) COMPUTATIONAL.
01 DUMMY Picture S9(4) COMPUTATIONAL.
01 MYTYPE Picture S9(4) COMPUTATIONAL.
01 COLUMIND Picture S9(4) COMPUTATIONAL.
01OLUMNLEN Picture S9(4) COMPUTATIONAL.
01 COLUMN-prec Picture S9(4) COMPUTATIONAL.
01 COLUMN-SCALE Picture S9(4) COMPUTATIONAL.
01 INDCOUNT Pic S9(4) COMPUTATIONAL.
01 ROWCOUNT Pic S9(4) COMPUTATIONAL.
01 ERR-FOUND Picture X(1).
01 WORKAREA2.
  02 WORKINDPTR POINTER OCCURS 750 TIMES.
**************************************************************
* DECLARE CURSOR AND STATEMENT FOR DYNAMIC SQL *
**************************************************************
* EXEC SQL DECLARE DT CURSOR FOR SEL END-EXEC.
* EXEC SQL DECLARE SEL STATEMENT END-EXEC.
* EXEC SQL INCLUDE SQLCA END-EXEC.
**************************************************************
* MAIN PROGRAM ROUTINE *
**************************************************************
SET IRECPTR TO ADDRESS OF REC1-CHAR(1).
/ PROCEDURE DIVISION USING LINKAREA-IND LINKAREA-REC.
* ************* SQL RETURN CODE HANDLING *************
EXEC SQL WHENEVER SQLError GOTO DBERROR END-EXEC.
EXEC SQL WHENEVER SQLWarning GOTO DBERROR END-EXEC.
EXEC SQL WHENEVER NOT FOUND CONTINUE END-EXEC.
* MAIN PROGRAM ROUTINE ************
SET IRECPTR TO ADDRESS OF REC1-CHAR(1).
**OPEN FILES**

MOVE 'N' TO ERR-FOUND.

**INITIALIZE**

OPEN INPUT SYSIN

OUTPUT SYSPRINT

OUTPUT SYSREC01.

**WRITE HEADER**

WRITE MSGREC FROM HEADER AFTER ADVANCING 2 LINES.

**GET FIRST INPUT**

READ SYSIN RECORD INTO IOAREA.

**MAIN ROUTINE**

PERFORM PROCESS-INPUT THROUGH IND-RESULT.

**CLOSE FILES**

CLOSE SYSIN

SYSPRINT

SYSREC01.

GOBACK.

---------------------------------------------------------------------

**PERFORMED SECTION:**

**PROCESSING FOR THE TABLE OR VIEW JUST READ**

---------------------------------------------------------------------

PROCESS-INPUT.

MOVE TNAME TO STMTTAB.

MOVE STMTBLD TO STMTCHAR.

MOVE +750 TO SQLN.

EXEC SQL PREPARE SEL INTO :SQLDA FROM :STMTBUF END-EXEC.

---------------------------------------------------------------------

**SET UP ADDRESSES IN THE SQLDA FOR DATA.**

---------------------------------------------------------------------

IF SQLD = ZERO THEN

WRITE MSGREC FROM MSGNOCOL AFTER ADVANCING 2 LINES

MOVE 'Y' TO ERR-FOUND

GO TO IND-RESULT.

MOVE ZERO TO ROWCOUNT.

MOVE ZERO TO REC1-LEN.

SET RECPTR TO IRECPTR.

MOVE ONE TO I.

PERFORM COLADDR UNTIL I > SQLD.

---------------------------------------------------------------------

**SET LENGTH OF OUTPUT RECORD.**

**EXEC SQL OPEN DT END-EXEC.**

**DO WHILE SQLCODE IS 0.**

**EXEC SQL FETCH DT USING DESCRIPTOR :SQLDA END-EXEC.**

**ADD IN MARKERS TO DENOTE NULLS.**

**WRITE THE DATA TO SYSREC01.**

**INCREMENT DATA RECORD COUNTER.**

END.

---------------------------------------------------------------------

**OPEN CURSOR**

EXEC SQL OPEN DT END-EXEC.

PERFORM BLANK-REC.

EXEC SQL FETCH DT USING DESCRIPTOR :SQLDA END-EXEC.

**NO ROWS FOUND**

**PRINT ERROR MESSAGE**

IF SQLCODE = NOT-FOUND

WRITE MSGREC FROM MSG-NOROW AFTER ADVANCING 2 LINES

MOVE 'Y' TO ERR-FOUND

ELSE

**WRITE ROW AND**

**CONTINUE UNTIL**

**NO MORE ROWS**

PERFORM WRITE-AND-FETCH UNTIL SQLCODE IS NOT EQUAL TO ZERO.

**EXEC SQL WHENEVER NOT FOUND GOTO CLOSEDT END-EXEC.**

**CLOSEDT.**
EXEC SQL CLOSE DT END-EXEC.
*
****************************************************************
* INDICATE THE RESULTS OF THE UNLOAD OPERATION. *
****************************************************************
IND-RESULT.
IF ERR-FOUND = 'N' THEN
  MOVE TNAME TO TABLENAM
  MOVE ROWCOUNT TO ROWS
  WRITE MSGREC FROM UNLOADED
  AFTER ADVANCING 2 LINES
ELSE
  WRITE MSGREC FROM MSG-OTHER-ERR
  AFTER ADVANCING 2 LINES
  MOVE +0012 TO RETURN-CODE
  GO TO PROG-END.
*
WRITE-AND-FETCH.
  ADD IN markers to denote NULLs.
  MOVE ONE TO INDCOUNT.
  PERFORM NULLCHK UNTIL INDCOUNT = SQLD.
  MOVE REC1-LEN TO REC01-LEN.
  WRITE REC01 FROM LINKAREA-REC.
  ADD ONE TO ROWCOUNT.
  PERFORM BLANK-REC.
  EXEC SQL FETCH DT USING DESCRIPTOR :SQLDA END-EXEC.
*
NULLCHK.
IF IND(INDCOUNT) < 0 THEN
  SET ADDRESS OF LINKAREA-QMARK TO WORKINDPTR(INDCOUNT)
  MOVE QMARK TO INDREC.
  ADD ONE TO INDCOUNT.
****************************************************************
* BLANK OUT RECORD TEXT FIRST *
****************************************************************
BLANK-REC.
  MOVE ONE TO J.
  PERFORM BLANK-MORE UNTIL J > REC1-LEN.
BLANK-MORE.
  MOVE ' ' TO REC1-CHAR(J).
  ADD ONE TO J.
*
COLADDR.
  SET SQLDATA(I) TO RECPTR.
****************************************************************
* DETERMINE THE LENGTH OF THIS COLUMN (COLUMN-LEN)
* THIS DEPENDS UPON THE DATA TYPE. MOST DATA TYPES HAVE
* THE LENGTH SET, BUT VARCHAR, GRAPHIC, VARGRAPHIC, AND
* DECIMAL DATA NEED TO HAVE THE BYTES CALCULATED.
* THE NULL ATTRIBUTE MUST BE SEPARATED TO SIMPLIFY MATTERS.
****************************************************************
  MOVE SQLLEN(I) TO COLUMN-LEN.
  COLUMN-IND IS 0 FOR NO NULLS AND 1 FOR NULLS
  DIVIDE SQLTYPE(I) BY TWO GIVING DUMMY REMAINDER COLUMN-IND.
  MYTYPE IS JUST THE SQLTYPE WITHOUT THE NULL BIT
  MOVE SQLTYPE(I) TO MYTYPE.
  SUBTRACT COLUMN-IND FROM MYTYPE.
  EVALUATE MYTYPE
  WHEN CHARTYPE CONTINUE,
  WHEN DATETYP CONTINUE,
  WHEN TIMETYP CONTINUE,
  WHEN TIMESTMP CONTINUE,
  WHEN FLOATYPE CONTINUE,
  WHEN VARTYPE
    ADD TWO TO COLUMN-LEN,
  WHEN VARTYPE
    ADD TWO TO COLUMN-LEN,
  WHEN GTYPE
    MULTIPLY COLUMN-LEN BY TWO GIVING COLUMN-LEN,
  WHEN VARGTYPE
    PERFORM CALC-VARG-LEN,
  WHEN LVARLTYPE
    PERFORM CALC-VARG-LEN,
  WHEN HWTYPED
    MOVE TWO TO COLUMN-LEN,
  WHEN INTTYPE
    MOVE FOUR TO COLUMN-LEN,
WHEN DECTYPE
   PERFORM CALC-DECIMAL-LEN,
WHEN OTHER
   PERFORM UNRECOGNIZED-ERROR,
END-EVALUATE.
ADD COLUMN-LEN TO RECNUM.
ADD COLUMN-LEN TO REC1-LEN.
****************************************************************
*                                                              *
*    IF THIS COLUMN CAN BE NULL, AN INDICATOR VARIABLE IS      *
*    NEEDED. WE ALSO RESERVE SPACE IN THE OUTPUT RECORD TO    *
*    NOTE THAT THE VALUE IS NULL.                              *
*                                                              *
****************************************************************
MOVE ZERO TO IND(I).
IF COLUMN-IND = ONE THEN
   SET SQLIND(I) TO ADDRESS OF IND(I)
   SET WORKINDPTR(I) TO REC PTR
   ADD ONE TO RECNUM
   ADD ONE TO REC1-LEN.
   ADD ONE TO I.
*        PERFORMED PARAGRAPH TO CALCULATE COLUMN LENGTH        *
*        FOR A DECIMAL DATA TYPE COLUMN                        *
CALC-DECIMAL-LEN.
   DIVIDE COLUMN-LEN BY 256 GIVING COLUMN-PREC
   REMAINDER COLUMN-SCALE.
   MOVE COLUMN-PREC TO COLUMN-LEN.
   ADD ONE TO COLUMN-LEN.
   DIVIDE COLUMN-LEN BY TWO GIVING COLUMN-LEN.
   PERFORMED PARAGRAPH TO CALCULATE COLUMN LENGTH              *
   FOR A VARGRAPHIC DATA TYPE COLUMN                           *
CALC-VARG-LEN.
   MULTIPLY COLUMN-LEN BY TWO GIVING COLUMN-LEN.
   ADD TWO TO COLUMN-LEN.
   PERFORMED PARAGRAPH TO NOTE AN UNRECOGNIZED                 *
   DATA TYPE COLUMN UNRECOGNIZED-ERROR.                       *
*        ERROR MESSAGE FOR UNRECOGNIZED DATA TYPE               *
*        MOVE SQLTYPE(I) TO TYPCOD                             *
*        MOVE 'Y' TO ERR-FOUND                                 *
*        WRITE MSGREC FROM BADTYPE                             *
*        AFTER ADVANCING 2 LINES                               *
*        GO TO IND-RESULT.                                    *
   **SQL ERROR
   MOVE 'Y' TO ERR-FOUND.
   MOVE SQLCODE TO MSG-PRINT-CODE.
   IF SQLCODE < 0 THEN MOVE '-' TO MSG-MINUS.
   WRITE MSGREC FROM MSG-SQLERR
   AFTER ADVANCING 2 LINES.
   CALL 'DSNITAR' USING SQLCA ERROR-MESSAGE ERROR-TEXT-LEN.
   IF RETURN-CODE = ZERO
      PERFORM ERROR-PRINT VARYING ERROR-INDEX
         FROM 1 BY 1 UNTIL ERROR-INDEX GREATER THAN 8
   ELSE
      **ERROR FOUND IN DSNITAR
      **PRINT ERROR MESSAGE
   END-IF.
   MOVE RETURN-CODE TO RETCODE
   WRITE MSGREC FROM MSGRETCO
   AFTER ADVANCING 2 LINES.
   GO TO IND-RESULT.
*        **PRINT MESSAGE TEXT                               *
*        PRINT MESSAGE TEXT
*        **PRINT MESSAGE TEXT
ERROR-PRINT.
WRITE MSGREC FROM ERROR-TEXT (ERROR-INDEX)
AFTER ADVANCING 1 LINE.
This example demonstrates how to access distributed data by using CONNECT statements in a COBOL program.

The following figure contains a sample COBOL program that uses two-phase commit to access distributed data.

```
IDENTIFICATION DIVISION.
PROGRAM-ID. TWOPHASE.
AUTHOR.
REMARKS.
*****************************************************************
*                                                               *
* MODULE NAME = TWOPHASE                                        *
*                                                               *
* DESCRIPTIVE NAME = DB2 SAMPLE APPLICATION USING               *
* TWO PHASE COMMIT AND THE DRDA DISTRIBUTED ACCESS METHOD WITH CONNECT STATEMENTS *
*                                                               *
* COPYRIGHT = 5665-DB2 (C) COPYRIGHT IBM CORP 1982, 1989 *
* REFER TO COPYRIGHT INSTRUCTIONS FORM NUMBER G120-2083 *
*                                                               *
* STATUS = VERSION 5                                           *
*                                                               *
* FUNCTION = THIS MODULE DEMONSTRATES DISTRIBUTED DATA ACCESS *
* USING 2 PHASE COMMIT BY TRANSFERRING AN EMPLOYEE           *
* FROM ONE LOCATION TO ANOTHER.                              *
*                                                               *
* NOTE: THIS PROGRAM ASSUMES THE EXISTENCE OF THE              *
* TABLE SYSADM.EMP AT LOCATIONS STLEC1 AND STLEC2.             *
*                                                               *
* MODULE TYPE = COBOL PROGRAM                                  *
* PROCESSOR = DB2 PRECOMPILER, ENTERPRISE COBOL FOR Z/OS       *
* MODULE SIZE = SEE LINK EDIT                                  *
* ATTRIBUTES = NOT REENTRANT OR REUSABLE                      *
*                                                               *
* ENTRY POINT =                                               *
* PURPOSE = TO ILLUSTRATE 2 PHASE COMMIT                      *
* LINKAGE = INVOKE FROM DSN RUN                               *
* INPUT = NONE                                                *
* OUTPUT =                                                   *
* SYMBOLIC LABEL/NAME = SYSPRINT                             *
* DESCRIPTION = PRINT OUT THE DESCRIPTION OF EACH           *
* STEP AND THE RESULTANT SQLCA                               *
*                                                               *
* EXIT NORMAL = RETURN CODE 0 FROM NORMAL COMPLETION         *
*                                                               *
* EXIT ERROR = NONE                                           *
*                                                               *
* EXTERNAL REFERENCES =                                       *
* ROUTINE SERVICES = NONE                                     *
* DATA AREAS = NONE                                           *
* CONTROL BLOCKS =                                           *
* SQLCA = SQL COMMUNICATION AREA                             *
*                                                               *
* TABLES = NONE                                               *
*                                                               *
* CHANGE ACTIVITY = NONE                                      *
*                                                               *
*                                                               *
* PSEUDOCODE                                                   *
*                                                               *
* MAINLINE.                                                   *
* Perform CONNECT-TO-SITE-1 to establish                      *
* a connection to the local connection.                      *
* If the previous operation was successful Then               *
* Do.                                                        *
* Perform PROCESS-CURSOR-SITE-1 to obtain the                *
* information about an employee that is                      *
* transferring to another location.                         *
* If the information about the employee was obtained        *
* successfully Then                                           *
* Do.                                                        *
* Perform UPDATE-ADDRESS to update the information           *
```

Chapter 4. Embedded SQL programming 633
* | to contain current information about the employee.
* | Perform CONNECT-TO-SITE-2 to establish a connection to the site where the employee is transferring to.
* | If the connection is established successfully Then Do.
* | | Perform CONNECT-TO-SITE-2 to establish a connection to the location where the employee is transferring to.
* | | End if the connection was established successfully.
* | | End if the employee information was obtained successfully.
* | Perform COMMIT-WORK to COMMIT the changes made to STLEC1 and STLEC2.
* * *
* PROG-END.
* Close the printer.
* Return.
*
* CONNECT-TO-SITE-1.
* Provide a text description of the following step.
* Establish a connection to the location where the employee is transferring from.
* Print the SQLCA out.
* * *
* PROCESS-CURSOR-SITE-1.
* Provide a text description of the following step.
* Open a cursor that will be used to retrieve information about the transferring employee from this site.
* Print the SQLCA out.
* If the cursor was opened successfully Then Do.
* | Perform FETCH-DELETE-SITE-1 to retrieve and delete the information about the transferring employee from this site.
* | Perform CLOSE-CURSOR-SITE-1 to close the cursor.
* | End if the cursor was opened successfully.
* *
* FETCH-DELETE-SITE-1.
* Provide a text description of the following step.
* Fetch information about the transferring employee.
* Print the SQLCA out.
* If the information was retrieved successfully Then Do.
* | Perform DELETE-SITE-1 to delete the employee at this site.
* | End if the information was retrieved successfully.
* *
* DELETE-SITE-1.
* Provide a text description of the following step.
* Delete the information about the transferring employee from this site.
* Print the SQLCA out.
* *
* CLOSE-CURSOR-SITE-1.
* Provide a text description of the following step.
* Close the cursor used to retrieve information about the transferring employee.
* Print the SQLCA out.
* *
* UPDATE-ADDRESS.
* Update the address of the employee.
* Update the city of the employee.
* Update the location of the employee.
* *
* CONNECT-TO-SITE-2.
* Provide a text description of the following step.
* Establish a connection to the location where the employee is transferring to.
* Print the SQLCA out.
* *
* PROCESS-SITE-2.
* Provide a text description of the following step.
* Insert the employee information at the location where the employee is being transferred to.
* Print the SQLCA out.
*
* COMMIT-WORK.
* COMMIT all the changes made to STLEC1 and STLEC2.

PROGRAM-ID. COMMIT-WORK.
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
  SELECT PRINTER, ASSIGN TO S-OUT1.
DATA DIVISION.
FILE SECTION.
FD  PRINTER
  RECORD CONTAINS 120 CHARACTERS
  DATA RECORD IS PRT-TC-RESULTS
  LABEL RECORD IS OMITTED.
  01 PRT-TC-RESULTS.
  03 PRT-BLANK PIC X(120).

WORKING-STORAGE SECTION.

** Variable declarations  *

01 H-EMPTBL.
   05 H-EMPNO   PIC X(6).
   05 H-NAME.
      49 H-NAME-LN PIC S9(4) COMP-4.
      49 H-NAME-DA PIC X(32).
   05 H-ADDRESS.
      49 H-ADDRESS-LN PIC S9(4) COMP-4.
      49 H-ADDRESS-DA PIC X(36).
   05 H-CITY.
      49 H-CITY-LN PIC S9(4) COMP-4.
      49 H-CITY-DA PIC X(36).
   05 H-EMPLOC PIC X(4).
   05 H-SSNO PIC X(11).
   05 H-BORN PIC X(10).
   05 H-SEX PIC X(1).
   05 H-HIRED PIC X(10).
   05 H-DEPTNO PIC X(3).
   05 H-JOBCODE PIC S9(3)V COMP-3.
   05 H-SRATE PIC S9(5) COMP.
   05 H-EDUC PIC S9(5) COMP.
   05 H-SAL PIC S9(6)V9(2) COMP-3.
   05 H-VALIDCHK PIC S9(6)V COMP-3.

01 H-EMPTBL-IND-TABLE.
   02 H-EMPTBL-IND PIC S9(4) COMP OCCURS 15 TIMES.

** Includes for the variables used in the COBOL standard language procedures and the SQLCA.  *

EXEC SQL INCLUDE COBSVAR END-EXEC.
EXEC SQL INCLUDE SQLCA END-EXEC.

** Declaration for the table that contains employee information  *

EXEC SQL DECLARE SYSDM.EMP TABLE
  (EMPNO   CHAR(6) NOT NULL,
   NAME VARCHAR(32),
   ADDRESS VARCHAR(36),
   CITY VARCHAR(36),
   EMPLOC CHAR(4) NOT NULL,
   SSNO CHAR(11),
   BORN DATE,
   SEX CHAR(1),
   HIRED CHAR(10),
   DEPTNO CHAR(3) NOT NULL,
   JOBCODE DECIMAL(3),
   SRATE SMALLINT,
   EDUC SMALLINT,
SAL DECIMAL(8,2) NOT NULL,
VALCHK DECIMAL(6))
END-EXEC.

*****************************************************************
* Constants                                                     *
*****************************************************************
77 SITE-1 PIC X(16) VALUE 'STLEC1'.
77 SITE-2 PIC X(16) VALUE 'STLEC2'.
77 TEMP-EMPNO PIC X(6) VALUE '080000'.
77 TEMP-ADDRESS-LN PIC 99 VALUE 15.
77 TEMP-CITY-LN PIC 99 VALUE 18.

*****************************************************************
* Declaration of the cursor that will be used to retrieve       *
* information about a transferring employee                     *
*****************************************************************
EXEC SQL DECLARE C1 CURSOR FOR
SELECT EMPNO, NAME, ADDRESS, CITY, EMPLOC,
SSNO, BORN, SEX, HIRED, DEPTNO, JOBCODE,
SRATE, EDUC, SAL, VALCHK
FROM   SYSADM.EMP
WHERE EMPNO = :TEMP-EMPNO
END-EXEC.

PROCEDURE DIVISION.
A101-HOUSE-KEEPING.
OPEN OUTPUT PRINTER.

*****************************************************************
* An employee is transferring from location STLEC1 to STLEC2.   *
* Retrieve information about the employee from STLEC1, delete   *
* the employee from STLEC1 and insert the employee at STLEC2    *
* using the information obtained from STLEC1.                   *
*****************************************************************

PERFORM CONNECT-TO-SITE-1
IF SQLCODE IS EQUAL TO 0
PERFORM PROCESS-CURSOR-SITE-1
IF SQLCODE IS EQUAL TO 0
PERFORM UPDATE-ADDRESS
PERFORM CONNECT-TO-SITE-2
IF SQLCODE IS EQUAL TO 0
PERFORM PROCESS-SITE-2.
PERFORM COMMIT-WORK.
PROG-END.
CLOSE PRINTER.
GOBACK.

*****************************************************************
* Establish a connection to STLEC1                              *
*****************************************************************
CONNECT-TO-SITE-1.
MOVE 'CONNECT TO STLEC1   ' TO STNAME
WRITE PRT-TC-RESULTS FROM STNAME
EXEC SQL
 CONNECT TO :SITE-1
 END-EXEC.
PERFORM PTSQLCA.

*****************************************************************
* When a connection has been established successfully at STLEC1,*
* open the cursor that will be used to retrieve information    *
* about the transferring employee.                            *
*****************************************************************

PROCESS-CURSOR-SITE-1.
MOVE 'OPEN CURSOR C1    ' TO STNAME
WRITE PRT-TC-RESULTS FROM STNAME
EXEC SQL
 OPEN C1
END-EXEC.
PERFORM PTSQLCA.
IF SQLCODE IS EQUAL TO ZERO
PERFORM FETCH-DELETE-SITE-1
PERFORM CLOSE-CURSOR-SITE-1.

******************************************************************************
* Retrieve information about the transferring employee.                      *
* Provided that the employee exists, perform DELETE-SITE-1 to               *
* delete the employee from STLEC1.                                         *
******************************************************************************

FETCH-DELETE-SITE-1.
MOVE 'FETCH C1      ' TO STNAME
WRITE PRT-TC-RESULTS FROM STNAME
EXEC SQL
   FETCH C1 INTO :H-EMPTBL:H-EMPTBL-IND
END-EXEC.
PERFORM PTSQLCA.
IF SQLCODE IS EQUAL TO ZERO
PERFORM DELETE-SITE-1.

******************************************************************************
* Delete the employee from STLEC1.                                          *
******************************************************************************

DELETE-SITE-1.
MOVE 'DELETE EMPLOYEE ' TO STNAME
WRITE PRT-TC-RESULTS FROM STNAME
MOVE 'DELETE EMPLOYEE     ' TO STNAME
EXEC SQL
   DELETE FROM SYSADM.EMP
   WHERE EMPNO = :TEMP-EMPNO
END-EXEC.
PERFORM PTSQLCA.

******************************************************************************
* Close the cursor used to retrieve information about the                  *
* transferring employee.                                                   *
******************************************************************************

CLOSE-CURSOR-SITE-1.
MOVE 'CLOSE CURSOR C1     ' TO STNAME
WRITE PRT-TC-RESULTS FROM STNAME
EXEC SQL
   CLOSE C1
END-EXEC.
PERFORM PTSQLCA.

******************************************************************************
* Update certain employee information in order to make it                  *
* current.                                                                *
******************************************************************************

UPDATE-ADDRESS.
MOVE TEMP-ADDRESS-LN      TO H-ADDRESS-LN.
MOVE '1500 NEW STREET'    TO H-ADDRESS-DA.
MOVE TEMP-CITY-LN         TO H-CITY-LN.
MOVE 'NEW CITY, CA 97804' TO H-CITY-DA.
MOVE 'SJCA'               TO H-EMPLOC.

******************************************************************************
* Establish a connection to STLEC2                                         *
******************************************************************************

CONNECT-TO-SITE-2.
MOVE 'CONNECT TO STLEC2     ' TO STNAME
WRITE PRT-TC-RESULTS FROM STNAME
EXEC SQL
   CONNECT TO :SITE-2
END-EXEC.
PERFORM PTSQLCA.

******************************************************************************
* Using the employee information that was retrieved from STLEC1            *
* and updated previously, insert the employee at STLEC2.                   *
******************************************************************************

Chapter 4. Embedded SQL programming 637
Sample COBOL program using aliases for three-part names

You can access distributed data by using aliases for three-part names in a COBOL program.

The following sample program demonstrates distributed access data using aliases for three-part names with two-phase commit.

 IDENTIFICATION DIVISION.
 PROGRAM-ID. TWOPHASE.
 AUTHOR.
 REMARKS.
*******************************************************************************
 * MODULE NAME = TWOPHASE
 * * DESCRIPITIVE NAME = DB2 SAMPLE APPLICATION USING ALIASES FOR THREE-PART NAMES
 * * FUNCTION = THIS MODULE DEMONSTRATES DISTRIBUTED DATA ACCESS
 * * USING 2 PHASE COMMIT BY TRANSFERRING AN EMPLOYEE FROM ONE LOCATION TO ANOTHER.
 * * NOTE: THIS PROGRAM ASSUMES THE EXISTENCE OF THE TABLE SYSADM.ALLEMPLOYEES AT LOCATIONS STLEC1 AND STLEC2.
 * * MODULE TYPE = COBOL PROGRAM
 * * PROCESSOR = DB2 PRECOMPILER, ENTERPRISE COBOL FOR Z/OS
 * * MODULE SIZE = SEE LINK EDIT
 * * ATTRIBUTES = NOT REENTRANT OR REUSABLE
 * ENTRY POINT =
* PURPOSE = TO ILLUSTRATE 2 PHASE COMMIT
* LINKAGE = INVOKE FROM DSN RUN
* INPUT = NONE
* OUTPUT = NONE
* SYMABLET/NAME = SYSprint
* DESCRIPTION = PRINT OUT THE DESCRIPTION OF EACH
* STEP AND THE RESULTANT SQLCA
* EXIT NORMAL = RETURN CODE 0 FROM NORMAL COMPLETION
* EXIT ERROR = NONE
* EXTERNAL REFERENCES =
* ROUTINE SERVICES = NONE
* DATA AREAS = NONE
* CONTROL BLOCKS =
* SQLCA - SQL COMMUNICATION AREA
* TABLES = NONE
* CHANGE ACTIVITY = NONE
* PSEUDOCODE
* MAINLINE.
* Perform PROCESS CURSOR SITE 1 to obtain the information
* about an employee that is transferring to another
* location. If the information about the employee was obtained
* successfully Then Do.
* | Perform UPDATE ADDRESS to update the information to
* | contain current information about the employee. Perform
* | PROCESS SITE 2 to insert the employee
* | information at the location where the employee is
* | transferring to. End if the employee information was obtained
* | successfully. Perform COMMIT WORK to COMMIT the changes made to STLEC1
* | and STLEC2.
* Do.
* If the cursor was opened successfully Then Do.
* | Perform FETCH DELETE SITE 1 to retrieve and
* | delete the information about the transferring
* | employee from this site. Perform
* | CLOSE CURSOR SITE 1 to close the cursor. End if the cursor was opened successfully.
* FETCH DELETE SITE 1.
* Provide a text description of the following step.
* Fetch information about the transferring employee.
* Print the SQLCA out. If the information was retrieved successfully Then Do.
* | Perform DELETE SITE 1 to delete the employee
* | at this site. End if the information was retrieved successfully.
* DELETE SITE 1.
* Provide a text description of the following step.
* Delete the information about the transferring employee
* from this site. Print the SQLCA out.
* CLOSE CURSOR SITE 1.
* Provide a text description of the following step.
* Close the cursor used to retrieve information about
* the transferring employee. Print the SQLCA out.
UPDATE-ADDRESS.
* Update the address of the employee.
* Update the city of the employee.
* Update the location of the employee.

PROCESS-SITE-2.
* Provide a text description of the following step.
* Insert the employee information at the location where the employee is being transferred to.
* Print the SQLCA out.

COMMIT WORK.
* COMMIT all the changes made to STLEC1 and STLEC2.

ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILECONTROL.
SELECT PRINTER, ASSIGN TO S-OUT1.

DATA DIVISION.
FILE SECTION.
FD PRINTER
RECORD CONTAINS 120 CHARACTERS
DATA RECORD IS PRT-TC-RESULTS
LABEL RECORD IS OMITTED.
01 PRT-TC-RESULTS.
03 PRT-BLANK PIC X(120).

WORKING-STORAGE SECTION.

EXEC SQL INCLUDE COBSVAR END-EXEC.
EXEC SQL INCLUDE SQLCA END-EXEC.

EXEC SQL DECLARE SYSADM.ALLEMPLOYEES TABLE
(EMPNO CHAR(6) NOT NULL,
 NAME VARCHAR(32),
 ADDRESS VARCHAR(36),
 CITY VARCHAR(36),
 EMPLOC CHAR(4) NOT NULL,
 SSNO CHAR(11),
 BORN DATE,
EXEC SQL DECLARE C1 CURSOR FOR
SELECT EMPNO, NAME, ADDRESS, CITY, EMPLOC,
SSNO, BORN, SEX, HIRED, DEPTNO, JOBCode,
SRATE, EDUC, SAL, VALCHK
FROM   EC1EMP
WHERE EMPNO = :TEMP-EMPNO
END-EXEC.

EXEC SQL DECLARE C1 CURSOR FOR
SELECT EMPNO, NAME, ADDRESS, CITY, EMPLOC,
SSNO, BORN, SEX, HIRED, DEPTNO, JOBCode,
SRATE, EDUC, SAL, VALCHK
FROM   EC1EMP
WHERE EMPNO = :TEMP-EMPNO
END-EXEC.

EXEC SQL DECLARE C1 CURSOR FOR
SELECT EMPNO, NAME, ADDRESS, CITY, EMPLOC,
SSNO, BORN, SEX, HIRED, DEPTNO, JOBCode,
SRATE, EDUC, SAL, VALCHK
FROM   EC1EMP
WHERE EMPNO = :TEMP-EMPNO
END-EXEC.

EXEC SQL DECLARE C1 CURSOR FOR
SELECT EMPNO, NAME, ADDRESS, CITY, EMPLOC,
SSNO, BORN, SEX, HIRED, DEPTNO, JOBCode,
SRATE, EDUC, SAL, VALCHK
FROM   EC1EMP
WHERE EMPNO = :TEMP-EMPNO
END-EXEC.
END-EXEC. PERFORM PSQLCA.
IF SQLCODE IS EQUAL TO ZERO
  PERFORM DELETE-SITE-1.

*****************************************************************************
* Delete the employee from STLEC1.                                      *
*****************************************************************************

DELETE-SITE-1.
MOVE 'DELETE EMPLOYEE ' TO STNAME
WRITE PRT-TC-RESULTS FROM STNAME
EXEC SQL
  DELETE FROM EC1EMP
  WHERE EMPNO = :TEMP-EMPNO
END-EXEC.
PERFORM PSQLCA.

*****************************************************************************
* Close the cursor used to retrieve information about the transferring *
* employee.                                                           *
*****************************************************************************

CLOSE-CURSOR-SITE-1.
MOVE 'CLOSE CURSOR C1' TO STNAME
WRITE PRT-TC-RESULTS FROM STNAME
EXEC SQL
  CLOSE C1
END-EXEC.
PERFORM PSQLCA.

*****************************************************************************
* Update certain employee information in order to make it current.     *
*****************************************************************************

UPDATE-ADDRESS.
MOVE TEMP-ADDRESS-LN TO H-ADDRESS-LN.
MOVE '1500 NEW STREET' TO H-ADDRESS-DA.
MOVE TEMP-CITY-LN TO H-CITY-LN.
MOVE 'NEW CITY, CA 97804' TO H-CITY-DA.
MOVE 'SJCA' TO H-EMPLOC.

*****************************************************************************
* Using the employee information that was retrieved from STLEC1 and *
* updated previously, insert the employee at STLEC2.                   *
*****************************************************************************

PROCESS-SITE-2.
MOVE 'INSERT EMPLOYEE ' TO STNAME
WRITE PRT-TC-RESULTS FROM STNAME
EXEC SQL
  INSERT INTO EC2EMP VALUES
    (:H-EMPNO,    :H-NAME,     :H-ADDRESS,    :H-CITY,    :H-EMPLOC,
END-EXEC.
PERFORM PSQLCA.

*****************************************************************************
* COMMIT any changes that were made at STLEC1 and STLEC2.              *
*****************************************************************************

COMMIT-WORK.
MOVE 'COMMIT WORK' TO STNAME
WRITE PRT-TC-RESULTS FROM STNAME
Example COBOL stored procedure with a GENERAL WITH NULLS linkage convention

You can call a stored procedure that uses the GENERAL WITH NULLS linkage convention from a COBOL program.

This example stored procedure does the following:

- Searches the Db2 SYSIBM.SYSROUTINES catalog table for a row that matches the input parameters from the client program. The two input parameters contain values for NAME and SCHEMA.
- Searches the Db2 catalog table SYSTABLES for all tables in which the value of CREATOR matches the value of input parameter SCHEMA. The stored procedure uses a cursor to return the table names.

The linkage convention for this stored procedure is GENERAL WITH NULLS.

The output parameters from this stored procedure contain the SQLCODE from the SELECT operation, and the value of the RUNOPTS column retrieved from the SYSIBM.SYSROUTINES table.

The CREATE PROCEDURE statement for this stored procedure might look like this:

```sql
CREATE PROCEDURE GETPRML(PROCNM CHAR(18) IN, SCHEMA CHAR(8) IN,
OUTCODE INTEGER OUT, PARMLST VARCHAR(254) OUT)
LANGUAGE COBOL
DETERMINISTIC
READS SQL DATA
EXTERNAL NAME "GETPRML"
COLLID GETPRML
ASUTIME NO LIMIT
PARAMETER STYLE GENERAL WITH NULLS
STAY RESIDENT NO
RUN OPTIONS "MSGFILE(OUTFILE),RPTSTG(ON),RPTOPTS(ON)"
WLM ENVIRONMENT SAMPPROG
PROGRAM TYPE MAIN
SECURITY DB2
RESULT SETS 2
COMMIT ON RETURN NO;
```

The following example is a COBOL stored procedure with linkage convention GENERAL WITH NULLS.
EXEC SQL DECLARE C1 CURSOR WITH RETURN FOR
   SELECT NAME FROM SYSIBM.SYSTABLES WHERE CREATOR=:INSCHEMA
END-EXEC.

EXEC SQL WITH RETURN FOR
   SELECT RUNOPTS INTO :PARMLST
   FROM SYSIBM.SYSROUTINES
   WHERE NAME=:PROCNM AND
   SCHEMA=:SCHEMA
END-EXEC

MOVE 0 TO PARMLST-IND

COPY SQLCODE INTO THE OUTPUT PARAMETER AREA

MOVE SQLCODE TO OUT-CODE
MOVE 0 TO OUT-CODE-IND.

OPEN CURSOR C1 TO CAUSE DB2 TO RETURN A RESULT SET
TO THE CALLER.

例：COBOL 存储过程使用一般连接约定

您可以从 COBOL 程序调用使用一般连接约定的存储过程。

此示例存储过程执行以下操作：
- 搜索 SYSCAT.SYSROUTINES 表以查找与来自客户端程序的输入参数匹配的一行。两个输入参数包含 NAME 和 SCHEMA 的值。
- 搜索 Db2 数据库表 SYSTABLES 中所有 CREATOR 值与输入参数 SCHEMA 匹配的表。存储过程使用游标将表名返回。

Example COBOL stored procedure with a GENERAL linkage convention

You can call a stored procedure that uses the GENERAL linkage convention from a COBOL program.

This example stored procedure does the following:
- Searches the catalog table SYSCAT.SYSROUTINES for a row matching the input parameters from the client program. The two input parameters contain values for NAME and SCHEMA.
- Searches the Db2 catalog table SYSTABLES for all tables in which the value of CREATOR matches the value of input parameter SCHEMA. The stored procedure uses a cursor to return the table names.
This stored procedure is able to return a NULL value for the output host variables.
The linkage convention for this stored procedure is GENERAL.
The output parameters from this stored procedure contain the SQLCODE from the SELECT operation, and
the value of the RUNOPTS column retrieved from the SYSROUTINES table.
The CREATE PROCEDURE statement for this stored procedure might look like this:

```sql
CREATE PROCEDURE GETPRML(PROCNM CHAR(18) IN, SCHEMA CHAR(8) IN,
OUTCODE INTEGER OUT, PARMLST VARCHAR(254) OUT)
LANGUAGE COBOL
DETERMINISTIC
READS SQL DATA
EXTERNAL NAME "GETPRML"
COLLID GETPRML
ASUTIME NO LIMIT
PARAMETER STYLE GENERAL
STAY RESIDENT NO
RUN OPTIONS "MSGFILE(OUTFILE),RPTSTG(ON),RPTOPTS(ON)"
WLM ENVIRONMENT SAMPPROG
PROGRAM TYPE MAIN
RESULT SETS 2
COMMIT ON RETURN NO;
```

CBL RENT
IDENTIFICATION DIVISION.
PROGRAM-ID. GETPRML.
AUTHOR. EXAMPLE.
DATE-WRITTEN. 03/25/98.

ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
DATA DIVISION.
FILE SECTION.
WORKING-STORAGE SECTION.

```
EXEC SQL INCLUDE SQLCA END-EXEC.

* DECLARE A HOST VARIABLE TO HOLD INPUT SCHEMA
01 INSCHEMA PIC X(8).

* DECLARE CURSOR FOR RETURNING RESULT SETS
EXEC SQL DECLARE C1 CURSOR WITH RETURN FOR
SELECT NAME FROM SYSIBM.SYSTABLES WHERE CREATOR=:INSCHEMA
END-EXEC.

* DECLARE THE INPUT PARAMETERS FOR THE PROCEDURE
01 PROCNM  PIC X(18).
01 SCHEMA  PIC X(8).

* DECLARE THE OUTPUT PARAMETERS FOR THE PROCEDURE
01 OUT-CODE PIC S9(9) USAGE BINARY.
01 PARMLST.
   49 PARMLST-LEN PIC S9(4) USAGE BINARY.
   49 PARMLST-TEXT PIC X(254).

PROCEDURE DIVISION USING PROCNM, SCHEMA,
OUT-CODE, PARMLST.

* Issue the SQL SELECT against the SYSIBM.SYSROUTINES
* DB2 catalog table.
EXEC SQL
CALL SELECT RUNOPTS INTO :PARMLST
FROM SYSIBM.ROUTINES
```

Chapter 4. Embedded SQL programming 645
Example COBOL program that calls a stored procedure

You can call the GETPRML stored procedure that uses the GENERAL WITH NULLS linkage convention from a COBOL program on a z/OS system.

Because the stored procedure returns result sets, this program checks for result sets and retrieves the contents of the result sets. The following figure contains the example COBOL program that calls the GETPRML stored procedure.
PARMBUF REDEFINES PARMLST.
   49 PARBLN   PIC S9(4) USAGE COMP.
   49 PARMARRY PIC X(127) OCCURS 2 TIMES.
01 NAME.
   49 NAMELEN   PIC S9(4) USAGE COMP.
   49 NAMETXT   PIC X(18).
77 PARMIND   PIC S9(4) COMP.
77 I         PIC S9(4) COMP.
77 NUMLINES   PIC S9(4) COMP.
******************************************************************************
* DECLARE A RESULT SET LOCATOR FOR THE RESULT SET *
* THAT IS RETURNED.                                *
******************************************************************************
01 LOC      USAGE SQL TYPE IS
            RESULT-SET-LOCATOR VARYING.
******************************************************************************
* SQL INCLUDE FOR SQLCA                             *
******************************************************************************
EXEC SQL INCLUDE SQLCA  END-EXEC.

PROCEDURE DIVISION.

*------------------
PROG-START.

OPEN OUTPUT REPOUT.
   *
   * OPEN OUTPUT FILE
   * MOVE 'DSN8EP2
   *       ' TO PROCNM.
   * INPUT PARAMETER -- PROCEDURE TO BE FOUND
   * MOVE SPACES TO SCHEMA.
   * INPUT PARAMETER -- SCHEMA IN SYSROUTINES
   * MOVE -1 TO PARMIND.
   * THE PARMLST PARAMETER IS AN OUTPUT PARM.
   * MARK PARMLST PARAMETER AS NULL, SO THE DB2
   * REQUESTER DOES NOT HAVE TO SEND THE ENTIRE
   * PARMLST VARIABLE TO THE SERVER. THIS
   * HELPS REDUCE NETWORK I/O TIME, BECAUSE
   * PARMLST IS FAIRLY LARGE.
   *
   * EXEC SQL
   * CALL GETPRML(:PROCNM,
   *      :SCHEMA,
   *      :OUT-CODE,
   *      :PARMLST INDICATOR :PARMIND)
   *
END-EXEC.

* MAKE THE CALL
* IF SQLCODE NOT EQUAL TO +466 THEN
* IF CALL RETURNED BAD SQLCODE
** MOVE SQLCODE TO BADCOD
** WRITE REPREC FROM SQLREC
** MOVE SQLERRMC TO ERRMCODE
** WRITE REPREC FROM ERRMREC
** ELSE
** PERFORM GET-PARMS
** PERFORM GET-RESULT-SET.
* PROG-END.

   * CLOSE OUTPUT FILE
   * GOBACK.

   * MAKE THE CALL
   * IF SQLCODE NOT EQUAL TO 0 THEN
   * IF OUT-CODE NOT EQUAL TO 0 THEN
   * DID GETPRML HIT AN ERROR?
   * MOVE OUT-CODE TO CALLCODE
   * WRITE REPREC FROM CALLREC
   * ELSE
   * EVERYTHING WORKED
   * DIVIDE 127 INTO PARMLEN GIVING NUMLINES ROUNDED
   * FIND OUT HOW MANY LINES TO PRINT
   * PERFORM PARMPR VARYING I
   * FROM 1 BY 1 UNTIL I GREATER THAN NUMLINES.

   * GET-RESULT-SET.
** ASSUME YOU KNOW THAT ONE RESULT SET IS RETURNED,
** AND YOU KNOW THE FORMAT OF THAT RESULT SET.
** ALLOCATE A CURSOR FOR THE RESULT SET, AND FETCH
** THE CONTENTS OF THE RESULT SET.
**
Defining the SQL communications area, SQLSTATE, and SQLCODE in COBOL

COBOL programs that contain SQL statements can include an SQL communications area (SQLCA) to check whether an SQL statement executed successfully. Alternatively, these programs can declare individual SQLCODE and SQLSTATE host variables.

About this task

If you specify the SQL processing option STDSQL(YES), do not define an SQLCA. If you do, Db2 ignores your SQLCA, and your SQLCA definition causes compile-time errors. If you specify the SQL processing option STDSQL(NO), include an SQLCA explicitly.

For COBOL programs, when you specify STDSQL(YES), you must declare an SQLCODE variable. Db2 declares an SQLCA area for you in the WORKING-STORAGE SECTION. Db2 controls the structure and location of the SQLCA.

If your application contains SQL statements and does not include an SQL communications area (SQLCA), you must declare individual SQLCODE and SQLSTATE host variables. Your program can use these variables to check whether an SQL statement executed successfully.

Procedure

Choose one of the following actions:

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>To define the SQL communications area:</strong></td>
<td>a. Code the SQLCA directly in the program or use the following SQL INCLUDE statement to request a standard SQLCA declaration:</td>
</tr>
<tr>
<td></td>
<td>EXEC SQL INCLUDE SQLCA</td>
</tr>
<tr>
<td></td>
<td>You can specify INCLUDE SQLCA or a declaration for SQLCODE wherever you can specify a 77 level or a record description entry in the WORKING-STORAGE SECTION.</td>
</tr>
<tr>
<td></td>
<td>Db2 sets the SQLCODE and SQLSTATE values in the SQLCA after each SQL statement executes. Your application should check these values to determine whether the last SQL statement was successful.</td>
</tr>
<tr>
<td><strong>To declare SQLCODE and SQLSTATE host variables:</strong></td>
<td>a. Declare the SQLCODE variable within a BEGIN DECLARE SECTION statement and an END DECLARE SECTION statement in your program declarations as PIC S9(9) BINARY, PIC S9(9) COMP-4, PIC S9(9) COMP-5, or PICTURE S9(9) COMP.</td>
</tr>
<tr>
<td></td>
<td>When you use the Db2 precompiler, you can declare a stand-alone SQLCODE variable in either the WORKING-STORAGE SECTION or LINKAGE SECTION. When you use the Db2 coprocessor, you can declare a stand-alone SQLCODE...</td>
</tr>
<tr>
<td>Option</td>
<td>Description</td>
</tr>
<tr>
<td>--------</td>
<td>-------------</td>
</tr>
<tr>
<td></td>
<td>variable in the WORKING-STORAGE SECTION, LINKAGE SECTION or LOCAL-STORAGE SECTION.</td>
</tr>
<tr>
<td></td>
<td>b. Declare the SQLSTATE variable within a BEGIN DECLARE SECTION statement and an END DECLARE SECTION statement in your program declarations as PICTURE X(5).</td>
</tr>
<tr>
<td><strong>Restriction:</strong></td>
<td>Do not declare an SQLSTATE variable as an element of a structure.</td>
</tr>
<tr>
<td><strong>Requirement:</strong></td>
<td>After you declare the SQLCODE and SQLSTATE variables, ensure that all SQL statements in the program are within the scope of the declaration of these variables.</td>
</tr>
</tbody>
</table>

**Related tasks**

Checking the execution of SQL statements
After executing an SQL statement, your program should check for any errors before you commit the data and handle the errors that they represent.

Checking the execution of SQL statements by using the SQLCA
One way to check whether an SQL statement executed successfully is to use the SQL communication area (SQLCA). This area is set apart for communication with Db2.

Checking the execution of SQL statements by using SQLCODE and SQLSTATE
Whenever an SQL statement executes, the SQLCODE and SQLSTATE fields of the SQLCA receive a return code.

Defining the items that your program can use to check whether an SQL statement executed successfully
If your program contains SQL statements, the program should define some infrastructure so that it can check whether the statements executed successfully. You can either include an SQL communications area (SQLCA), which contains SQLCODE and SQLSTATE variables, or declare individual SQLCODE and SQLSTATE host variables.

Defining SQL descriptor areas (SQLDA) in COBOL

If your program includes certain SQL statements, you must define at least one SQL descriptor area (SQLDA). Depending on the context in which it is used, the SQLDA stores information about prepared SQL statements or host variables. This information can then be read by either the application program or Db2.

**Procedure**

Perform one of the following actions:

- Code the SQLDA declarations directly in your program. When you use the Db2 precompiler, you must place SQLDA declarations in the WORKING-STORAGE SECTION or LINKAGE SECTION of your program, wherever you can specify a record description entry in that section. When you use the Db2 coprocessor, you must place SQLDA declarations in the WORKING-STORAGE SECTION, LINKAGE SECTION or LOCAL-STORAGE SECTION of your program, wherever you can specify a record description entry in that section.

- Call a subroutine that is written in C, PL/I, or assembler language and that uses the INCLUDE SQLDA statement to define the SQLDA. The subroutine can also include SQL statements for any dynamic SQL functions that you need.

**Restriction:**

- You must place SQLDA declarations before the first SQL statement that references the data descriptor, unless you use the TWOPASS SQL processing option.

**Related tasks**

Defining SQL descriptor areas (SQLDA)
If your program includes certain SQL statements, you must define at least one SQL descriptor area (SQLDA). Depending on the context in which it is used, the SQLDA stores information about prepared SQL statements or host variables. This information can then be read by either the application program or Db2.

**Related reference**
SQL descriptor area (SQLDA) (Db2 SQL)

**Declaring host variables and indicator variables in COBOL**
You can use host variables, host-variable arrays, and host structures in SQL statements in your program to pass data between Db2 and your application.

**Procedure**
To declare host variables, host-variable arrays, and host structures:
1. Declare the variables according to the following rules and guidelines:
   - You must explicitly declare all host variables and host-variable arrays that are used in SQL statements in the WORKING-STORAGE SECTION or LINKAGE SECTION of your program's DATA DIVISION.
   - You must explicitly declare each host variable and host-variable array before using them in an SQL statement.
   - You can specify OCCURS when defining an indicator structure, a host-variable array, or an indicator variable array. You cannot specify OCCURS for any other type of host variable.
   - You cannot implicitly declare any host variables through default typing or by using the IMPLICIT statement.
   - If you specify the ONEPASS SQL processing option, you must explicitly declare each host variable and each host-variable array before using them in an SQL statement. If you specify the TWOPASS precompiler option, you must declare each host variable before using it in the DECLARE CURSOR statement.
   - If you specify the STDSQL(YES) SQL processing option, you must precede the host language statements that define the host variables and host-variable arrays with the BEGIN DECLARE SECTION statement and follow the host language statements with the END DECLARE SECTION statement. Otherwise, these statements are optional.
   - Ensure that any SQL statement that uses a host variable or host-variable array is within the scope of the statement that declares that variable or array.
   - If you are using the Db2 precompiler, ensure that the names of host variables and host-variable arrays are unique within the program, even if the variables and variable arrays are in different blocks, classes, procedures, functions, or subroutines. You can qualify the names with a structure name to make them unique.
2. Optional: Define any associated indicator variables, arrays, and structures.

**Related tasks**
Declaring host variables and indicator variables
You can use host variables and indicator variables in SQL statements in your program to pass data between Db2 and your application.

**Host variables in COBOL**
In COBOL programs, you can specify numeric, character, graphic, binary, LOB, XML, and ROWID host variables. You can also specify result set and table locators and LOB and XML file reference variables.

**Restrictions:**
- Only some of the valid COBOL declarations are valid host variable declarations. If the declaration for a variable is not valid, any SQL statement that references the variable might result in the message UNDECLARED HOST VARIABLE.
• You can not use locators as column types.

The following locator data types are COBOL data types and SQL data types:

– Result set locator
– Table locator
– LOB locators
– LOB file reference variables

• One or more REDEFINES entries can follow any level 77 data description entry. However, you cannot use the names in these entries in SQL statements. Entries with the name FILLER are ignored.

**Recommendations:**

• Be careful of overflow. For example, suppose that you retrieve an INTEGER column value into a PICTURE S9(4) host variable and the column value is larger than 32767 or smaller than -32768. You get an overflow warning or an error, depending on whether you specify an indicator variable.

• Be careful of truncation. For example, if you retrieve an 80-character CHAR column value into a PICTURE X(70) host variable, the rightmost 10 characters of the retrieved string are truncated. Retrieving a double precision floating-point or decimal column value into a PIC S9(8) COMP host variable removes any fractional part of the value. Similarly, retrieving a column value with DECIMAL data type into a COBOL decimal variable with a lower precision might truncate the value.

• If your varying-length string host variables receive values whose length is greater than 9999 bytes, compile the applications in which you use those host variables with the option TRUNC(BIN). TRUNC(BIN) lets the length field for the string receive a value of up to 32767 bytes.

**Numeric host variables**

You can specify the following forms of numeric host variables:

• Floating-point numbers
• Integers and small integers
• Decimal numbers

The following diagram shows the syntax for declaring floating-point or real host variables.

```
01 variable-name
  77 level-1
    variable-name
    1
    USAGE IS COMPUTATIONAL-1
    2
    COMPUTATIONAL-1
    COMP-1
    COMPUTATIONAL-2
    3
    COMP-2
    VALUE IS numeric-constant

Notes:
1 level-1 indicates a COBOL level between 2 and 48.
2 COMPUTATIONAL-1 and COMP-1 are equivalent.
3 COMPUTATIONAL-2 and COMP-2 are equivalent.
```

The following diagram shows the syntax for declaring integer and small integer host variables.
Notes:

1. `level-1` indicates a COBOL level between 2 and 48.
2. The COBOL binary integer data types BINARY, COMPUTATIONAL, COMP, COMPUTATIONAL-4, and COMP-4 are equivalent.
3. COMPUTATIONAL-5 (and COMP-5) are equivalent to the other COBOL binary integer data types if you compile the other data types with TRUNC(BIN).
4. Any specification for scale is ignored.

The following diagram shows the syntax for declaring decimal host variables.
In COBOL, you declare the SMALLINT and INTEGER data types as a number of decimal digits. Db2 uses the full size of the integers (in a way that is similar to processing with the TRUNC(BIN) compiler option) and can place larger values in the host variable than would be allowed in the specified number of digits in the COBOL declaration. If you compile with TRUNC(OPT) or TRUNC(STD), ensure that the size of numbers in your application is within the declared number of digits.

For small integers that can exceed 9999, use S9(4) COMP-5 or compile with TRUNC(BIN). For large integers that can exceed 999,999,999, use S9(10) COMP-3 to obtain the decimal data type. If you use COBOL for integers that exceed the COBOL PICTURE, specify the column as decimal to ensure that the data types match and perform well.

If you are using a COBOL compiler that does not support decimal numbers of more than 18 digits, use one of the following data types to hold values of greater than 18 digits:

- A decimal variable with a precision less than or equal to 18, if the actual data values fit. If you retrieve a decimal value into a decimal variable with a scale that is less than the source column in the database, the fractional part of the value might be truncated.
- An integer or a floating-point variable, which converts the value. If you use an integer variable, you lose the fractional part of the number. If the decimal number might exceed the maximum value for an integer or if you want to preserve a fractional value, use a floating-point variable. Floating-point numbers are approximations of real numbers. Therefore, when you assign a decimal number to a floating-point variable, the result might be different from the original number.
- A character-string host variable. Use the CHAR function to retrieve a decimal value into it.

**Restriction:** The SQL data type DECFLOAT has no equivalent in COBOL.
Character host variables

You can specify the following forms of character host variables:

• Fixed-length strings
• Varying-length strings
• CLOBs

The following diagrams show the syntax for forms other than CLOBs.

The following diagram shows the syntax for declaring fixed-length character host variables.

```
01 variable-name PIC picture-string 2.
77 level-1 1
    PICTURE PIC IS picture-string 2
    USAGE DISPLAY
    VALUE IS character-constant
```

Notes:
1 level-1 indicates a COBOL level between 2 and 48.
2 The picture-string that is associated with these forms must be X(m) (or XX...X, with m instances of X), where m is up to COBOL's limitation. However, the maximum length of the CHAR data type (fixed-length character string) in Db2 is 255 bytes.

The following diagrams show the syntax for declaring varying-length character host variables.

```
01 variable-name .
77 level-1 1
```

Notes:
1 level-1 indicates a COBOL level between 2 and 48.
Notes:

1 You cannot use an intervening REDEFINE at level 49.
2 You cannot directly reference \texttt{var-1} as a host variable.
3 Db2 uses the full length of the S9(4) BINARY variable even though COBOL with TRUNC(STD) recognizes values up to only 9999. This behavior can cause data truncation errors when COBOL statements execute and might effectively limit the maximum length of variable-length character strings to 9999. Consider using the TRUNC(BIN) compiler option or USAGE COMP-5 to avoid data truncation.

Notes:

1 You cannot use an intervening REDEFINE at level 49.
2 You cannot directly reference \texttt{var-2} as a host variable.
3 For fixed-length strings, the picture-string must be \texttt{X(m)} (or XX, with \( m \) instances of X), where \( m \) is up to COBOL's limitation. However, the maximum length of the VARCHAR data type in Db2 varies depending on the data page size.

**Graphic character host variables**

You can specify the following forms of graphic host variables:

- Fixed-length strings
- Varying-length strings
DBCLOBs

The following diagrams show the syntax for forms other than DBCLOBs.

The following diagram shows the syntax for declaring fixed-length graphic host variables.

```
01 77 level-1 1

variable-name

PICTURE

IS

picture-string 2

USAGE IS DISPLAY-1 NATIONAL 3

VALUE IS graphic-constant
```

Notes:
1. `level-1` indicates a COBOL level between 2 and 48.
2. For fixed-length strings, the `picture-string` is G(`m`) or N(`m`) (or, `m` instances of GG...G or NN...N), where `m` is up to COBOL's limitation. However, the maximum length of the GRAPHIC data type (fixed-length graphic string) in Db2 is 127 double-bytes.
3. Use USAGE NATIONAL only for Unicode UTF-16 data. In the `picture-string` for USAGE NATIONAL, you must use N in place of G. USAGE NATIONAL is supported only by the Db2 coprocessor.

The following diagrams show the syntax for declaring varying-length graphic host variables.

```
01 variable-name

level-1 1

Notes:
1. `level-1` indicates a COBOL level between 2 and 48.
```

```
49 var-1 1

PICTURE

IS

S9(4) 2

S9999

USAGE IS

BINARY COMPUTATIONAL-4

COMP-4

COMPUTATIONAL-5

COMP-5

COMPUTATIONAL

COMP

VALUE

IS

numeric-constant
```

656 Db2 12 for z/OS: Application Programming and SQL Guide
Notes:

1 You cannot directly reference var-1 as a host variable.
2 Db2 uses the full length of the S9(4) BINARY variable even though COBOL with TRUNC(STD) recognizes values up to only 9999. This behavior can cause data truncation errors when COBOL statements execute and might effectively limit the maximum length of variable-length character strings to 9999. Consider using the TRUNC(BIN) compiler option or USAGE COMP-5 to avoid data truncation.

49 var-2 PICTURE
   IS picture-string
   PIC
   DISPLAY-1
   USAGE IS NATIONAL
   VALUE IS graphic-constant.

Notes:

1 You cannot directly reference var-2 as a host variable.
2 For fixed-length strings, the picture-string is G(m) or N(m) (or, m instances of GG...G or NN...N), where m is up to COBOL's limitation. However, the maximum length of the VARGRAPHIC data type in Db2 varies depending on the data page size.
3 Use USAGE NATIONAL only for Unicode UTF-16 data. In the picture-string for USAGE NATIONAL, you must use N in place of G. USAGE NATIONAL is supported only by the Db2 coprocessor.

Binary host variables

You can specify the following forms of binary host variables:

- Fixed-length strings
- Varying-length strings
- BLOBs

The following diagram shows the syntax for declaring BINARY and VARBINARY host variables.

Notes:

1 For BINARY host variables, the length must be in the range 1 - 255. For VARBINARY host variables, the length must be in the range 1 - 32704.
COBOL does not have variables that correspond to the SQL binary types BINARY and VARBINARY. To create host variables that can be used with these data types, use the SQL TYPE IS clause. The SQL precompiler replaces this declaration with a COBOL language structure in the output source member.

When you reference a BINARY or VARBINARY host variable in an SQL statement, you must use the variable that you specify in the SQL TYPE declaration. When you reference the host variable in a host language statement, you must use the variable that Db2 generates.

**Examples of binary variable declarations**
The following table shows examples of variables that Db2 generates when you declare binary host variables.

<table>
<thead>
<tr>
<th>Variable declaration that you include in your COBOL program</th>
<th>Corresponding variable that Db2 generates in the output source member</th>
</tr>
</thead>
<tbody>
<tr>
<td>01 BIN-VAR USAGE IS SQL TYPE IS BINARY(10).</td>
<td>01 BIN-VAR PIC X(10).</td>
</tr>
<tr>
<td>01 VBIN-VAR USAGE IS SQL TYPE IS VARBINARY(10).</td>
<td>01 VBIN-VAR. 49 VBIN-VAR-LEN PIC 9(4) USAGE BINARY. 49 VBIN-VAR-TEXT PIC X(10).</td>
</tr>
</tbody>
</table>

**Result set locators**
The following diagram shows the syntax for declaring result set locators.

```
01 variable-name

SQL TYPE IS RESULT-SET-LOCATOR

USAGE IS VARYING
```

**Table Locators**
The following diagram shows the syntax for declaring table locators.

```
01 level-1 variable-name

SQL TYPE IS TABLE LIKE table-name AS LOCATOR

USAGE IS
```

Notes:

1. level-1 indicates a COBOL level between 2 and 48.

**LOB variables and file reference variables**
The following diagram shows the syntax for declaring BLOB, CLOB, and DBCLOB variables and file reference variables.
XML data host and file reference variables

The following diagram shows the syntax for declaring BLOB, CLOB, and DBCLOB host variables and file reference variables for XML data types.

Notes:
1  level-1 indicates a COBOL level between 2 and 48.

ROWID host variables

The following diagram shows the syntax for declaring ROWID host variables.
Notes:

1 `level-1` indicates a COBOL level between 2 and 48.

Related concepts
Host variables
Use host variables to pass a single data item between Db2 and your application.

Related tasks
Storing LOB data in a table
Db2 handles LOB data differently than it handles other kinds of data. As a result, in some cases, you need to take additional actions when you define LOB columns and insert the LOB data.

Related reference
Limits in Db2 for z/OS (Db2 SQL)

Host-variable arrays in COBOL

In COBOL programs, you can specify numeric, character, graphic, LOB, XML, and ROWID host-variable arrays. You can also specify LOB locators and LOB and XML file reference variables.

Host-variable arrays can be referenced only as a simple reference in the following contexts. In syntax diagrams, `host-variable-array` designates a reference to a host-variable array.

• In a FETCH statement for a multiple-row fetch. See FETCH (Db2 SQL).
• In an INSERT statement with multiple rows of source data. This is also referred to as multiple-row insert. See INSERT (Db2 SQL).
• In a MERGE statement with multiple rows of source data. See MERGE (Db2 SQL).
• In an EXECUTE statement to provide a value for a parameter marker in a dynamic multi-row INSERT or MERGE. See EXECUTE (Db2 SQL).

Restriction: Only some of the valid COBOL declarations are valid host-variable array declarations. If the declaration for a variable array is not valid, any SQL statement that references the variable array might result in the message UNDECLARED HOST VARIABLE ARRAY.

Numeric host-variable arrays

You can specify the following forms of numeric host-variable arrays:

• Floating-point numbers
• Integers and small integers
• Decimal numbers

The following diagram shows the syntax for declaring floating-point host-variable arrays.
The following diagram shows the syntax for declaring integer and small integer host-variable arrays.

Notes:
1. `level-1` indicates a COBOL level between 2 and 48.
2. COMPUTATIONAL-1 and COMP-1 are equivalent.
3. COMPUTATIONAL-2 and COMP-2 are equivalent.
4. `dimension` must be an integer constant between 1 and 32767.
The following diagram shows the syntax for declaring decimal host-variable arrays.

```
level-1 1 variable-name PICTURE IS picture-string

USAGE IS

PACKED-DECIMAL

COMPUTATIONAL-3

COMP-3

DISPLAY SIGN IS LEADING SEPARATE CHARACTER

OCCURS dimension TIMES

VALUE IS numeric-constant
```

Notes:
1. *level-1* indicates a COBOL level between 2 and 48.
2. PACKED-DECIMAL, COMPUTATIONAL-3, and COMP-3 are equivalent. The *picture-string* that is associated with these types must have the form S9(i)V9(d) (or S9...9V9...9, with i and d instances of 9) or S9(i)V.
3. The *picture-string* that is associated with SIGN LEADING SEPARATE must have the form S9(i)V9(d) (or S9...9V9...9, with i and d instances of 9 or S9...9V with i instances of 9).
4. *dimension* must be an integer constant between 1 and 32767.

**Character host-variable arrays**

You can specify the following forms of character host-variable arrays:

- Fixed-length character strings
- Varying-length character strings
- CLOBs

The following diagrams show the syntax for forms other than CLOBs.

The following diagram shows the syntax for declaring fixed-length character string arrays.
**Notes:**

1. *level-1* indicates a COBOL level between 2 and 48.
2. The *picture-string* must be in the form X(\(m\)) (or XX...X, with \(m\) instances of X), where \(1 \leq m \leq 32767\) for fixed-length strings. However, the maximum length of the CHAR data type (fixed-length character string) in Db2 is 255 bytes.
3. *dimension* must be an integer constant between 1 and 32767.

The following diagrams show the syntax for declaring varying-length character string arrays.

**Notes:**

1. *level-1* indicates a COBOL level between 2 and 48.
2. *dimension* must be an integer constant between 1 and 32767.

**Notes:**

1. You cannot directly reference *var-1* as a host-variable array.
2. Db2 uses the full length of the S9(4) BINARY variable even though COBOL with TRUNC(STD) recognizes values up to only 9999. This behavior can cause data truncation errors when COBOL statements execute...
and might effectively limit the maximum length of variable-length character strings to 9999. Consider using the TRUNC(BIN) compiler option or USAGE COMP-5 to avoid data truncation.

Notes:
1 You cannot directly reference var-2 as a host-variable array.
2 The picture-string must be in the form X(m) (or XX…X, with m instances of X), where 1 <= m <= 32767 for fixed-length strings; for other strings, m cannot be greater than the maximum size of a varying-length character string.
3 You cannot use an intervening REDEFINE at level 49.

The following example shows declarations of a fixed-length character array and a varying-length character array.

```plaintext
01 OUTPUT-VARS.
   05 NAME OCCURS 10 TIMES.
      49 NAME-LEN   PIC S9(4) COMP-4 SYNC.
      49 NAME-DATA  PIC X(40).
   05 SERIAL-NUMBER PIC S9(9) COMP-4 OCCURS 10 TIMES.
```

**Graphic character host-variable arrays**

You can specify the following forms of graphic host-variable arrays:

- Fixed-length strings
- Varying-length strings
- DBCLOBs

The following diagrams show the syntax for forms other than DBCLOBs.

The following diagram shows the syntax for declaring fixed-length graphic string arrays.
The following diagrams show the syntax for declaring varying-length graphic string arrays.

Notes:
1. level-1 indicates a COBOL level between 2 and 48.
2. dimension must be an integer constant between 1 and 32767.
You cannot directly reference `var-1` as a host-variable array.

Db2 uses the full length of the S9(4) BINARY variable even though COBOL with TRUNC(STD) recognizes values up to only 9999. This behavior can cause data truncation errors when COBOL statements execute and might effectively limit the maximum length of variable-length character strings to 9999. Consider using the TRUNC(BIN) compiler option or USAGE COMP-5 to avoid data truncation.

```
囉 var-2 1
| PICTURE IS picture-string 2 USAGE IS
| DISPLAY-1 NATIONAL 3 4 VALUE IS graphic-constant .
```

Notes:
1. You cannot directly reference `var-2` as a host-variable array.
2. For fixed-length strings, the format for `picture-string` is G(m) or N(m) (or, m instances of GG...G or NN...N), where 1 <= m <= 127; for other strings, m cannot be greater than the maximum size of a varying-length graphic string.
3. Use USAGE NATIONAL only for Unicode UTF-16 data. In the `picture-string` for USAGE NATIONAL, you must use N in place of G.
4. You can use USAGE NATIONAL only if you are using the Db2 coprocessor.

**Binary host-variable arrays**

The following diagram shows the syntax for declaring binary host-variable arrays.

```
| level-1 1 variable-name SQL TYPE IS BINARY VARYING ( length 2 ) |
| OCCURS dimension 3 TIMES |
```

Notes:
1. `level-1` indicates a COBOL level between 2 and 48.
2. For BINARY host variables, the `length` must be in the range 1 to 255. For VARBINARY host variables, the `length` must be in the range 1 to 32704.
3. `dimension` must be an integer constant between 1 and 32767.

**LOB, locator, and file reference variable arrays**

The following diagram shows the syntax for declaring BLOB, CLOB, and DBCLOB host variable, locator, and file reference arrays.
XML host and file reference variable arrays

The following diagram shows the syntax for declaring BLOB, CLOB, and DBCLOB host variable and file reference arrays for XML data types.
ROWID variable arrays

The following diagram shows the syntax for declaring ROWID variable arrays.

```
level-1 1 variable-name
  |          | SQL TYPE IS  \\
  |          | ROWID         \\
  |          | OCCURS       \\
  | USAGE IS \\
  | dimension 2 TIMES \\
```

Notes:
1. level-1 indicates a COBOL level between 2 and 48.
2. dimension must be an integer constant between 1 and 32767.

Related concepts
- Using host-variable arrays in SQL statements
- Host-variable arrays
- Host-variable arrays in PL/I, C, C++, and COBOL (Db2 SQL)

Related tasks
- Inserting multiple rows of data from host-variable arrays
- Storing LOB data in a table
- Retrieving multiple rows of data into host-variable arrays

Host structures in COBOL

A COBOL host structure is a named set of host variables that are defined in your program's WORKING-STORAGE SECTION or LINKAGE SECTION.

Requirements: Host structure declarations in COBOL must satisfy the following requirements:

- COBOL host structures can have a maximum of two levels, even though the host structure might occur within a structure with multiple levels. However, you can declare a varying-length character string, which must be level 49.
- A host structure name can be a group name whose subordinate levels name elementary data items.
- If you are using the Db2 precompiler, do not declare host variables or host structures on any subordinate levels after one of the following items:
  - A COBOL item that begins in area A
- Any SQL statement (except SQL INCLUDE)
- Any SQL statement within an included member

When the Db2 precompiler encounters one of the preceding items in a host structure, it considers the structure to be complete.

When you write an SQL statement that contains a qualified host variable name (perhaps to identify a field within a structure), use the name of the structure followed by a period and the name of the field. For example, for structure B that contains field C1, specify B.C1 rather than C1 OF B or C1 IN B.

**Host structures**

The following diagram shows the syntax for declaring host structures.

```
\[level-1 \] \[ variable-name \].
\[level-2 \] \[ var-1 \] numeric-usage .
\[ numeric-usage IS \] PICTURE PIC IS picture-string .
\[ char-inner-variable . \]
\[varchar-inner-variables \]
\[ vargraphic-inner-variables \]
\[ SQL TYPE IS \] ROWID .
\[ SQL TYPE IS \] TABLE LIKE table-name AS LOCATOR .
\[ LOB data type . \]
```

Notes:

1. *level-1* indicates a COBOL level between 1 and 47.
2. *level-2* indicates a COBOL level between 2 and 48.
3. For elements within a structure, use any level 02 through 48 (rather than 01 or 77), up to a maximum of two levels.
4. Using a FILLER or optional FILLER item within a host structure declaration can invalidate the whole structure.

**Numeric usage items**

The following diagram shows the syntax for numeric-usage items that are used within declarations of host structures.
Integer and decimal usage items

The following diagram shows the syntax for integer and decimal usage items that are used within declarations of host structures.

CHAR inner variables

The following diagram shows the syntax for CHAR inner variables that are used within declarations of host structures.
**VARCHAR inner variables**

The following diagrams show the syntax for VARCHAR inner variables that are used within declarations of host structures.

Notes:

1. The number 49 has a special meaning to Db2. Do not specify another number.

**VARGRAPHIC inner variables**

The following diagrams show the syntax for VARGRAPHIC inner variables that are used within declarations of host structures.
Notes:

1 For fixed-length strings, the format of picture-string is G(m) or N(m) (or, m instances of GG...G or NN...N), where 1 <= m <= 127; for other strings, m cannot be greater than the maximum size of a varying-length graphic string.
2 Use USAGE NATIONAL for only Unicode UTF-16 data. In the picture-string for USAGE NATIONAL, you must use N in place of G.
3 You can use USAGE NATIONAL only if you are using the Db2 coprocessor.

LOB variables, locators, and file reference variables

The following diagram shows the syntax for LOB variables, locators, and file reference variables that are used within declarations of host structures.
LOB variables and file reference variables for XML data

The following diagram shows the syntax for LOB variables and file reference variables that are used within declarations of host structures for XML.

Example

In the following example, B is the name of a host structure that contains the elementary items C1 and C2.

```
01 A
02 B
  03 C1 PICTURE ...  
  03 C2 PICTURE ...
```

To reference the C1 field in an SQL statement, specify B.C1.

Related concepts

Host structures
Use host structures to pass a group of host variables between Db2 and your application.

**Indicator variables, indicator arrays, and host structure indicator arrays in COBOL**

A COBOL indicator variable is a 2-byte binary integer. A COBOL indicator variable array is an array in which each element is declared as a 2-byte binary integer. You can use indicator variable arrays to support COBOL host structures.

You declare indicator variables in the same way that you declare host variables.

The following diagram shows the syntax for declaring an indicator variable in COBOL.

```
01 variable-name PICTURE IS S9(4) S9999.
   USAGE IS BINARY COMPUTATIONAL-4 COMP-4 COMPUTATIONAL-5 COMP-5 COMPUTATIONAL COMP.
   VALUE IS constant.
```

The following diagram shows the syntax for declaring an indicator array in COBOL.

```
level-1 1 variable-name PICTURE IS S9(4) S9999.
   USAGE IS BINARY OCCURS dimension 2.
   VALUE IS constant.
```

Notes:

1. `level-1` must be an integer between 2 and 48.
2. `dimension` must be an integer constant between 1 and 32767.
Examples

Example 1:
The following example shows declarations of three variables, with an indicator variable declaration for each host variable.

```
77 DAYVAR     PIC S9(4) BINARY.
77 BGNVAR     PIC X(8).
77 ENDVAR     PIC X(8).
77 DAYVAR-IND PIC S9(4) BINARY.
77 BGNVAR-IND PIC S9(4) BINARY.
77 ENDVAR-IND PIC S9(4) BINARY.
```

The following FETCH statement retrieves values from a table into the three host variables. If a table column value is null, Db2 sets the corresponding indicator variable to -1.

```
EXEC SQL FETCH CLS_CURSOR INTO
    :DAYVAR:DAYVAR-IND,
    :BGNVAR:BGNVAR-IND,
    :ENDVAR:ENDVAR-IND
END-EXEC.
```

Example 2:
The following example shows a declaration of a host structure, and a corresponding structure that contains an indicator array with the same number of elements as the number of variables in the host structure.

```
01 CLS.
   10 DAYVAR    PIC S9(4) BINARY.
   10 BGNVAR    PIC X(8).
   10 ENDVAR    PIC X(8).
01 CLS-IND-STRUCT.
   02 CLS-IND   PIC S9(4) BINARY OCCURS 3 TIMES.
```

The following FETCH statement retrieves values from a table into the host structure. If a table value is null, Db2 sets the corresponding indicator array element to -1.

```
EXEC SQL FETCH CLS_CURSOR INTO
    :CLS:CLS-IND
END-EXEC.
```

Related concepts
Indicator variables, arrays, and structures
An indicator variable is associated with a particular host variable. Each indicator variable contains a small integer value that indicates some information about the associated host variable. Indicator arrays and structures serve the same purpose for host-variable arrays and structures.

Related tasks
Inserting null values into columns by using indicator variables or arrays
If you need to insert null values into a column, using an indicator variable or array is an easy way to do so. An indicator variable or array is associated with a particular host variable or host-variable array.

Controlling the CCSID for COBOL host variables
Setting the CCSID for COBOL host variables is slightly different than the process for other host languages. In COBOL, several other settings affect the CCSID.

Before you begin
This task applies to programs that use IBM Enterprise COBOL for z/OS and the Db2 coprocessor.

Procedure
Use one or more of the following items:
The NATIONAL data type
Use this data type to declare Unicode values in the UTF-16 format (CCSID 1200).

If you declare a host variable HV1 as USAGE NATIONAL, Db2 always handles HV1 as if you had used the following DECLARE VARIABLE statement:

```
DECLARE :HV1 VARIABLE CCSID 1200
```

The COBOL CODEPAGE compiler option
Use this option to specify the default EBCDIC CCSID of character data items.

The SQLCCSID compiler option
Use this option to control whether the CODEPAGE compiler option influences the processing of SQL host variables in your COBOL programs (available in Enterprise COBOL V3R4 or later).

When you specify the SQLCCSID compiler option, the COBOL Db2 coprocessor uses the CCSID that is specified in the CODEPAGE compiler option. All host variables of character data type, other than NATIONAL, are specified with that CCSID unless they are explicitly overridden by a DECLARE VARIABLE statement.

When you specify the NOSQLCCSID compiler option, the CCSID that is specified in the CODEPAGE compiler option is used for processing only COBOL statements within the COBOL program. That CCSID is not used for the processing of host variables in SQL statements. Db2 uses the CCSIDs that are specified through Db2 mechanisms and defaults as host variable data value encodings.

The DECLARE VARIABLE statement.
This statement explicitly sets the CCSID for individual host variables.

Example
Assume that the COBOL SQLCCSID compiler option is specified and that the COBOL CODEPAGE compiler option is specified as CODEPAGE(1141). The following code shows how you can control the CCSID:

```
DATA DIVISION.
  01  HV1  PIC N(10) USAGE NATIONAL.
  01  HV2  PIC X(20) USAGE DISPLAY.
  01  HV3  PIC X(30) USAGE DISPLAY.
  ...
  EXEC SQL
   DECLARE :HV3 VARIABLE CCSID 1047
   END-EXEC.
  ...
PROCEDURE DIVISION.
  ...
  EXEC SQL
   SELECT C1, C2, C3 INTO :HV1, :HV2, :HV3 FROM T1
   END-EXEC.
```

Each of the host variables have the following CCSIDs:

<table>
<thead>
<tr>
<th>HV1</th>
<th>1200</th>
</tr>
</thead>
<tbody>
<tr>
<td>HV2</td>
<td>1141</td>
</tr>
<tr>
<td>HV3</td>
<td>1047</td>
</tr>
</tbody>
</table>

Assume that the COBOL NOSQLCCSID compiler option is specified, the COBOL CODEPAGE compiler option is specified as CODEPAGE(1141), and the Db2 default single byte CCSID is set to 37. In this case, each of the host variables in this example have the following CCSIDs:

<table>
<thead>
<tr>
<th>HV1</th>
<th>1200</th>
</tr>
</thead>
<tbody>
<tr>
<td>HV2</td>
<td>37</td>
</tr>
</tbody>
</table>
Related reference

Host variables in COBOL

In COBOL programs, you can specify numeric, character, graphic, binary, LOB, XML, and ROWID host variables. You can also specify result set and table locators and LOB and XML file reference variables.

Compiler options (COBOL) (Enterprise COBOL for z/OS Programming Guide)

Equivalent SQL and COBOL data types

When you declare host variables in your COBOL programs, the precompiler uses equivalent SQL data types. When you retrieve data of a particular SQL data type into a host variable, you need to ensure that the host variable is of an equivalent data type.

The following table describes the SQL data type and the base SQLTYPE and SQLLEN values that the precompiler uses for host variables in SQL statements.

<table>
<thead>
<tr>
<th>COBOL host variable data type</th>
<th>SQLTYPE of host variable</th>
<th>SQLLEN of host variable</th>
<th>SQL data type</th>
</tr>
</thead>
<tbody>
<tr>
<td>COMP-1</td>
<td>480</td>
<td>4</td>
<td>REAL or FLOAT(n) 1≤n≤21</td>
</tr>
<tr>
<td>COMP-2</td>
<td>480</td>
<td>8</td>
<td>DOUBLE PRECISION, or FLOAT(n) 22≤n≤53</td>
</tr>
<tr>
<td>S9(i)V9(d) COMP-3 or S9(i)V9(d) PACKED-DECIMAL</td>
<td>484</td>
<td>i+d in byte 1, d in byte 2</td>
<td>DECIMAL(i+d,d) or NUMERIC(i+d,d)</td>
</tr>
<tr>
<td>S9(i)V9(d) DISPLAY SIGN LEADING SEPARATE</td>
<td>504</td>
<td>i+d in byte 1, d in byte 2</td>
<td>No exact equivalent. Use DECIMAL(i+d,d) or NUMERIC(i+d,d)</td>
</tr>
<tr>
<td>S9(i)V9(d) NATIONAL SIGN LEADING SEPARATE</td>
<td>504</td>
<td>i+d in byte 1, d in byte 2</td>
<td>No exact equivalent. Use DECIMAL(i+d,d) or NUMERIC(i+d,d)</td>
</tr>
<tr>
<td>S9(4) COMP-4, S9(4) COMP-5, S9(4) COMP, or S9(4) BINARY</td>
<td>500</td>
<td>2</td>
<td>SMALLINT</td>
</tr>
<tr>
<td>S9(9) COMP-4, S9(9) COMP-5, S9(9) COMP, or S9(9) BINARY</td>
<td>496</td>
<td>4</td>
<td>INTEGER</td>
</tr>
<tr>
<td>S9(18) COMP-4, S9(18) COMP-5, S9(18) COMP, or S9(18) BINARY</td>
<td>492</td>
<td>8</td>
<td>BIGINT</td>
</tr>
<tr>
<td>Fixed-length character data</td>
<td>452</td>
<td>n</td>
<td>CHAR(n)</td>
</tr>
<tr>
<td>Varying-length character data 1≤n≤255</td>
<td>448</td>
<td>n</td>
<td>VARCHAR(n)</td>
</tr>
<tr>
<td>Varying-length character data m&gt;255</td>
<td>456</td>
<td>m</td>
<td>VARCHAR(m)</td>
</tr>
<tr>
<td>Fixed-length graphic data</td>
<td>468</td>
<td>m</td>
<td>GRAPHIC(m)</td>
</tr>
<tr>
<td>Varying-length graphic data 1≤m≤127</td>
<td>464</td>
<td>m</td>
<td>VARGRAPHIC(m)</td>
</tr>
<tr>
<td>COBOL host variable data type</td>
<td>SQLTYPE of host variable</td>
<td>SQLLEN of host variable</td>
<td>SQL data type</td>
</tr>
<tr>
<td>------------------------------</td>
<td>---------------------------</td>
<td>--------------------------</td>
<td>---------------</td>
</tr>
<tr>
<td>Varying-length graphic data</td>
<td>1</td>
<td>&gt;127</td>
<td>VARGRAPHIC(m)</td>
</tr>
<tr>
<td>SQL TYPE is BINARY(n),</td>
<td>912</td>
<td>n</td>
<td>BINARY(n)</td>
</tr>
<tr>
<td>1≤n≤255</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SQL TYPE is VARBINARY(n),</td>
<td>908</td>
<td>n</td>
<td>VARBINARY(n)</td>
</tr>
<tr>
<td>1≤n≤32704</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SQL TYPE IS RESULT-SET-</td>
<td>972</td>
<td>4</td>
<td>Result set locator²</td>
</tr>
<tr>
<td>LOCATOR</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SQL TYPE IS TABLE LIKE</td>
<td>976</td>
<td>4</td>
<td>Table locator²</td>
</tr>
<tr>
<td>table-name AS LOCATOR</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SQL TYPE IS BLOB-LOCATOR</td>
<td>960</td>
<td>4</td>
<td>BLOB locator²</td>
</tr>
<tr>
<td>SQL TYPE IS CLOB-LOCATOR</td>
<td>964</td>
<td>4</td>
<td>CLOB locator²</td>
</tr>
<tr>
<td>SQL TYPE IS DBCLOB-LOCATOR</td>
<td>968</td>
<td>4</td>
<td>DBCLOB locator²</td>
</tr>
<tr>
<td>USAGE IS SQL TYPE IS BLOB(i)</td>
<td>404</td>
<td>i</td>
<td>BLOB(i)</td>
</tr>
<tr>
<td>1≤i≤2147483647</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>USAGE IS SQL TYPE IS CLOB(i)</td>
<td>408</td>
<td>i</td>
<td>CLOB(i)</td>
</tr>
<tr>
<td>1≤i≤2147483647</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>USAGE IS SQL TYPE IS DBCLOB(m)</td>
<td>412</td>
<td>i</td>
<td>DBCLOB(m)³</td>
</tr>
<tr>
<td>1≤m≤1073741823³</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SQL TYPE IS XML AS BLOB(i)</td>
<td>404</td>
<td>0</td>
<td>XML</td>
</tr>
<tr>
<td>SQL TYPE IS XML AS CLOB(i)</td>
<td>408</td>
<td>0</td>
<td>XML</td>
</tr>
<tr>
<td>SQL TYPE IS XML AS DBCLOB(i)</td>
<td>412</td>
<td>0</td>
<td>XML</td>
</tr>
<tr>
<td>SQL TYPE IS BLOB-FILE</td>
<td>916/917</td>
<td>267</td>
<td>BLOB file reference²</td>
</tr>
<tr>
<td>SQL TYPE IS CLOB-FILE</td>
<td>920/921</td>
<td>267</td>
<td>CLOB file reference²</td>
</tr>
<tr>
<td>SQL TYPE IS DBCLOB-FILE</td>
<td>924/925</td>
<td>267</td>
<td>DBCLOB file reference²</td>
</tr>
<tr>
<td>SQL TYPE IS XML AS BLOB-FILE</td>
<td>916/917</td>
<td>267</td>
<td>XML BLOB file reference²</td>
</tr>
<tr>
<td>SQL TYPE IS XML AS CLOB-FILE</td>
<td>920/921</td>
<td>267</td>
<td>XML CLOB file reference²</td>
</tr>
<tr>
<td>SQL TYPE IS XML AS DBCLOB-FILE</td>
<td>924/925</td>
<td>267</td>
<td>XML DBCLOB file reference²</td>
</tr>
<tr>
<td>SQL TYPE IS ROWID</td>
<td>904</td>
<td>40</td>
<td>ROWID</td>
</tr>
</tbody>
</table>
### Table 112. SQL data types, SQLLEN values, and SQLTYPE values that the precompiler uses for host variables in COBOL programs (continued)

<table>
<thead>
<tr>
<th>COBOL host variable data type</th>
<th>SQLTYPE of host variable</th>
<th>SQLLEN of host variable</th>
<th>SQL data type</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>SQLTYPE of host variable1</td>
<td>SQLLEN of host variable</td>
<td>SQL data type</td>
</tr>
<tr>
<td>Notes:</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1. If a host variable includes an indicator variable, the SQLTYPE value is the base SQLTYPE value plus 1.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2. Do not use this data type as a column type.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3. m is the number of double-byte characters.</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The following table shows equivalent COBOL host variables for each SQL data type. Use this table to determine the COBOL data type for host variables that you define to receive output from the database. For example, if you retrieve TIMESTAMP data, you can define a fixed-length character string variable of length n.

This table shows direct conversions between SQL data types and COBOL data types. However, a number of SQL data types are compatible. When you do assignments or comparisons of data that have compatible data types, Db2 converts those compatible data types.

### Table 113. COBOL host variable equivalents that you can use when retrieving data of a particular SQL data type

<table>
<thead>
<tr>
<th>SQL data type</th>
<th>COBOL host variable equivalent</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>SMALLINT</td>
<td>S9(4) COMP-4, S9(4) COMP-5, S9(4) COMP, or S9(4) BINARY</td>
<td></td>
</tr>
<tr>
<td>INTEGER</td>
<td>S9(9) COMP-4, S9(9) COMP-5, S9(9) COMP, or S9(9) BINARY</td>
<td></td>
</tr>
<tr>
<td>DECIMAL(p,s) or NUMERIC(p,s)</td>
<td>S9(p-s)V9(s) COMP-3 or S9(p-s)V9(s) PACKED-DECIMAL DISPLAY SIGN LEADING SEPARATE NATIONAL SIGN LEADING SEPARATE</td>
<td>p is precision; s is scale. 0≤s≤p≤31. If s=0, use S9(p) or S9(p). If s=p, use SV9(s). If the COBOL compiler does not support 31-digit decimal numbers, no exact equivalent exists. Use COMP-2.</td>
</tr>
<tr>
<td>REAL or FLOAT (n)</td>
<td>COMP-1</td>
<td>1≤n≤21</td>
</tr>
<tr>
<td>DOUBLE PRECISION, DOUBLE or FLOAT (n)</td>
<td>COMP-2</td>
<td>22≤n≤53</td>
</tr>
<tr>
<td>BIGINT</td>
<td>S9(18) COMP-4, S9(18) COMP-5, S9(18) COMP, or S9(18) BINARY</td>
<td></td>
</tr>
<tr>
<td>CHAR(n)</td>
<td>Fixed-length character string. For example,</td>
<td>1≤n≤255</td>
</tr>
<tr>
<td></td>
<td>01 VAR-NAME PIC X(n).</td>
<td></td>
</tr>
<tr>
<td>SQL data type</td>
<td>COBOL host variable equivalent</td>
<td>Notes</td>
</tr>
<tr>
<td>------------------</td>
<td>--------------------------------</td>
<td>----------------------------------------------------------------------</td>
</tr>
<tr>
<td>VARCHAR(n)</td>
<td>Varying-length character string. For example,</td>
<td>The inner variables must have a level of 49.</td>
</tr>
<tr>
<td></td>
<td>01 VAR-NAME PIC S9(4)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>49 VAR-LEN PIC S9(4)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>USAGE BINARY.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>49 VAR-TEXT PIC X(n).</td>
<td></td>
</tr>
<tr>
<td>GRAPHIC(n)</td>
<td>Fixed-length graphic string. For example,</td>
<td>n refers to the number of double-byte characters, not to the number of bytes. 1sns127</td>
</tr>
<tr>
<td></td>
<td>01 VAR-NAME PIC G(n)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>USAGE IS DISPLAY-1.</td>
<td></td>
</tr>
<tr>
<td>VARGRAPHIC(n)</td>
<td>Varying-length graphic string. For example,</td>
<td>n refers to the number of double-byte characters, not to the number of bytes. The inner variables must have a level of 49.</td>
</tr>
<tr>
<td></td>
<td>01 VAR-NAME PIC S9(4)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>49 VAR-LEN PIC S9(4)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>USAGE BINARY.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>49 VAR-TEXT PIC G(n)</td>
<td></td>
</tr>
<tr>
<td>BINARY(n)</td>
<td>SQL TYPE IS BINARY(n)</td>
<td>1sns255</td>
</tr>
<tr>
<td>VARBINARY(n)</td>
<td>SQL TYPE IS VARBINARY(n)</td>
<td>1sns32704</td>
</tr>
<tr>
<td>DATE</td>
<td>Fixed-length character string of length n. For example,</td>
<td>If you are using a date exit routine, i is determined by that routine. Otherwise, i must be at least 10.</td>
</tr>
<tr>
<td></td>
<td>01 VAR-NAME PIC X(n).</td>
<td></td>
</tr>
<tr>
<td>TIME</td>
<td>Fixed-length character string of length n. For example,</td>
<td>If you are using a time exit routine, n is determined by that routine. Otherwise, n must be at least 6; to include seconds, n must be at least 8.</td>
</tr>
<tr>
<td></td>
<td>01 VAR-NAME PIC X(n).</td>
<td></td>
</tr>
<tr>
<td>TIMESTAMP</td>
<td>Fixed-length character string of length n. For example,</td>
<td>n must be at least 19. To include microseconds, n must be 26; if n is less than 26, truncation occurs on the microseconds part.</td>
</tr>
<tr>
<td></td>
<td>01 VAR-NAME PIC X(n).</td>
<td></td>
</tr>
<tr>
<td>TIMESTAMP(0)</td>
<td>Fixed-length character string of length n. For example,</td>
<td>n must be at least 19.</td>
</tr>
<tr>
<td></td>
<td>01 VAR-NAME PIC X(n).</td>
<td></td>
</tr>
<tr>
<td>TIMESTAMP(p) p &gt; 0</td>
<td>Fixed-length character string of length n. For example,</td>
<td>n must be at least 19. To include fractional seconds, n must be 20+x where x is the number of fractional seconds to include; if x is less than p, truncation occurs on the fractional seconds part.</td>
</tr>
<tr>
<td></td>
<td>01 VAR-NAME PIC X(n).</td>
<td></td>
</tr>
</tbody>
</table>
Table 113. COBOL host variable equivalents that you can use when retrieving data of a particular SQL data type (continued)

<table>
<thead>
<tr>
<th>SQL data type</th>
<th>COBOL host variable equivalent</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>TIMESTAMP(0) WITH TIME ZONE</td>
<td>Varying-length character string. For example,</td>
<td>The inner variables must have a level of 49. ( n ) must be at least 25.</td>
</tr>
<tr>
<td></td>
<td>01 VAR-NAME.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>49 VAR-LEN PIC S9(4) USAGE</td>
<td></td>
</tr>
<tr>
<td></td>
<td>BINARY.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>49 VAR-TEXT PIC X(( n )).</td>
<td></td>
</tr>
<tr>
<td>TIMESTAMP(( p )) WITH TIME ZONE</td>
<td>Varying-length character string. For example,</td>
<td>The inner variables must have a level of 49. ( n ) must be at least ( 26 + p ).</td>
</tr>
<tr>
<td></td>
<td>01 VAR-NAME.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>49 VAR-LEN PIC S9(4) USAGE</td>
<td></td>
</tr>
<tr>
<td></td>
<td>BINARY.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>49 VAR-TEXT PIC X(( n )).</td>
<td></td>
</tr>
<tr>
<td>Result set locator</td>
<td>SQL TYPE IS RESULT-SET-LOCATOR</td>
<td>Use this data type only for receiving result sets. Do not use this data type as a column type.</td>
</tr>
<tr>
<td>Table locator</td>
<td>SQL TYPE IS TABLE LIKE</td>
<td>Use this data type only in a user-defined function or stored procedure to receive rows of a transition table. Do not use this data type as a column type.</td>
</tr>
<tr>
<td></td>
<td>table-name AS LOCATOR</td>
<td></td>
</tr>
<tr>
<td>BLOB locator</td>
<td>USAGE IS SQL TYPE IS</td>
<td>Use this data type only to manipulate data in BLOB columns. Do not use this data type as a column type.</td>
</tr>
<tr>
<td></td>
<td>BLOB-LOCATOR</td>
<td></td>
</tr>
<tr>
<td>CLOB locator</td>
<td>USAGE IS SQL TYPE IS</td>
<td>Use this data type only to manipulate data in CLOB columns. Do not use this data type as a column type.</td>
</tr>
<tr>
<td></td>
<td>CLOB-LOCATOR</td>
<td></td>
</tr>
<tr>
<td>DBCLOB locator</td>
<td>USAGE IS SQL TYPE IS</td>
<td>Use this data type only to manipulate data in DBCLOB columns. Do not use this data type as a column type.</td>
</tr>
<tr>
<td></td>
<td>DBCLOB-LOCATOR</td>
<td></td>
</tr>
<tr>
<td>BLOB(( i ))</td>
<td>USAGE IS SQL TYPE IS</td>
<td>( 1 \leq n \leq 2147483647 )</td>
</tr>
<tr>
<td></td>
<td>BLOB(( i ))</td>
<td></td>
</tr>
<tr>
<td>CLOB(( i ))</td>
<td>USAGE IS SQL TYPE IS</td>
<td>( 1 \leq n \leq 2147483647 )</td>
</tr>
<tr>
<td></td>
<td>CLOB(( i ))</td>
<td></td>
</tr>
<tr>
<td>DBCLOB(( i ))</td>
<td>USAGE IS SQL TYPE IS</td>
<td>( i ) is the number of double-byte characters. ( 1 \leq n \leq 1073741823 )</td>
</tr>
<tr>
<td></td>
<td>DBCLOB(( i ))</td>
<td></td>
</tr>
<tr>
<td>XML</td>
<td>SQL TYPE IS XML AS BLOB(( i ))</td>
<td>( 1 \leq n \leq 2147483647 )</td>
</tr>
<tr>
<td>XML</td>
<td>SQL TYPE IS XML AS CLOB(( i ))</td>
<td>( 1 \leq n \leq 2147483647 )</td>
</tr>
<tr>
<td>XML</td>
<td>SQL TYPE IS XML AS DBCLOB(( i ))</td>
<td>( i ) is the number of double-byte characters. ( 1 \leq n \leq 1073741823 )</td>
</tr>
<tr>
<td>BLOB file reference</td>
<td>USAGE IS SQL TYPE IS</td>
<td>Use this data type only to manipulate data in BLOB columns. Do not use this data type as a column type.</td>
</tr>
<tr>
<td></td>
<td>BLOB-FILE</td>
<td></td>
</tr>
</tbody>
</table>
### Table 113. COBOL host variable equivalents that you can use when retrieving data of a particular SQL data type (continued)

<table>
<thead>
<tr>
<th>SQL data type</th>
<th>COBOL host variable equivalent</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>CLOB file reference</td>
<td><code>USAGE IS SQL TYPE IS CLOB-FILE</code></td>
<td>Use this data type only to manipulate data in CLOB columns. Do not use this data type as a column type.</td>
</tr>
<tr>
<td>DBCLOB file reference</td>
<td><code>USAGE IS SQL TYPE IS DBCLOB-FILE</code></td>
<td>Use this data type only to manipulate data in DBCLOB columns. Do not use this data type as a column type.</td>
</tr>
<tr>
<td>XML BLOB file reference</td>
<td><code>SQL TYPE IS XML AS BLOB-FILE</code></td>
<td>Use this data type only to manipulate XML data as BLOB files. Do not use this data type as a column type.</td>
</tr>
<tr>
<td>XML CLOB file reference</td>
<td><code>SQL TYPE IS XML AS CLOB-FILE</code></td>
<td>Use this data type only to manipulate XML data as CLOB files. Do not use this data type as a column type.</td>
</tr>
<tr>
<td>XML DBCLOB file reference</td>
<td><code>SQL TYPE IS XML AS DBCLOB-FILE</code></td>
<td>Use this data type only to manipulate XML data as DBCLOB files. Do not use this data type as a column type.</td>
</tr>
<tr>
<td>ROWID</td>
<td><code>SQL TYPE IS ROWID</code></td>
<td></td>
</tr>
</tbody>
</table>

**Related concepts**

**Compatibility of SQL and language data types**
The host variable data types that are used in SQL statements must be compatible with the data types of the columns with which you intend to use them.

**LOB host variable, LOB locator, and LOB file reference variable declarations**
When you write applications to manipulate LOB data, you need to declare host variables to hold the LOB data or LOB locator. Alternatively, you need to declare LOB file reference variables to point to the LOB data.

**Host variable data types for XML data in embedded SQL applications (Db2 Programming for XML)**

**Object-oriented extensions in COBOL**

When you use object-oriented extensions in a COBOL application, you need to consider where to place SQL statements, the SQLCA, the SQLDA, and host variable declarations. You also need to consider the rules for host variables.

**Where to place SQL statements in your application:** A COBOL source data set or member can contain the following elements:

- Multiple programs
- Multiple class definitions, each of which contains multiple methods

You can put SQL statements in only the first program or class in the source data set or member. However, you can put SQL statements in multiple methods within a class. If an application consists of multiple data sets or members, each of the data sets or members can contain SQL statements.

**Where to place the SQLCA, SQLDA, and host variable declarations:** You can put the SQLCA, SQLDA, and SQL host variable declarations in the WORKING-STORAGE SECTION of a program, class, or method. An SQLCA or SQLDA in a class WORKING-STORAGE SECTION is global for all the methods of the class. An SQLCA or SQLDA in a method WORKING-STORAGE SECTION is local to that method only.

If a class and a method within the class both contain an SQLCA or SQLDA, the method uses the SQLCA or SQLDA that is local.
**Rules for host variables:** You can declare COBOL variables that are used as host variables in the WORKING-STORAGE SECTION or LINKAGE-SECTION of a program, class, or method. You can also declare host variables in the LOCAL-STORAGE SECTION of a method. The scope of a host variable is the method, class, or program within which it is defined.

**Handling SQL error codes in Cobol applications**

Cobol applications can request more information about SQL error codes by using the DSNTIAR subroutine or issuing a GET DIAGNOSTICS statement.

**Procedure**

To request more information about SQL errors from Cobol programs, use the following approaches:

- You can use the MESSAGE_TEXT condition item field of the GET DIAGNOSTICS statement to convert an SQL return code into a text message. Programs that require long token message support should code the GET DIAGNOSTICS statement instead of DSNTIAR.

- You can use the subroutine DSNTIAR to convert an SQL return code into a text message. DSNTIAR takes data from the SQLCA, formats it into a message, and places the result in a message output area that you provide in your application program.

**DSNTIAR syntax**

DSNTIAR has the following syntax:

```
CALL 'DSNTIAR' USING sqlca message lrecl.
```

**DSNTIAR parameters**

The DSNTIAR parameters have the following meanings:

- **sqlca**
  - An SQL communication area.

- **message**
  - An output area, in VARCHAR format, in which DSNTIAR places the message text. The first halfword contains the length of the remaining area; its minimum value is 240.

  The output lines of text, each line being the length specified in lrecl, are put into this area. For example, you could specify the format of the output area as:

  ```
  01  ERROR-MESSAGE.
  02  ERROR-LEN   PIC S9(4)   COMP VALUE +1328.
  02  ERROR-TEXT  PIC X(132) OCCURS 10 TIMES
    INDEXED BY ERROR-INDEX.
  77  ERROR-TEXT-LEN      PIC S9(9) COMP VALUE +132.
  ⋮
  CALL 'DSNTIAR' USING SQLCA ERROR-MESSAGE ERROR-TEXT-LEN.
  ```

  where ERROR-MESSAGE is the name of the message output area containing 10 lines of length 132 each, and ERROR-TEXT-LEN is the length of each line.

- **lrecl**
  - A fullword containing the logical record length of output messages, between 72 and 240.

  An example of calling DSNTIAR from an application appears in the Db2 sample assembler program DSN8BC3, which is contained in the library DSN8C10.

- If your CICS application requires CICS storage handling, you must use the subroutine DSNTIAC instead of DSNTIAR.

  If you call DSNTIAR dynamically from a CICS COBOL application program, be sure you do the following:

  - Compile the COBOL application with the NODYNAM option.
  - Define DSNTIAR in the CSD.
DSNTIAC syntax

If your CICS application requires CICS storage handling, you must use the subroutine DSNTIAC instead of DSNTIAR. DSNTIAC has the following syntax:

```
CALL 'DSNTIAC' USING eib commarea sqlca msg lrecl.
```

DSNTIAC parameters

DSNTIAC has extra parameters, which you must use for calls to routines that use CICS commands.

- **eib**: EXEC interface block
- **commarea**: communication area

For more information on these parameters, see the appropriate application programming guide for CICS. The remaining parameter descriptions are the same as those for DSNTIAR. Both DSNTIAC and DSNTIAR format the SQLCA in the same way.

You must define DSNTIA1 in the CSD. If you load DSNTIAR or DSNTIAC, you must also define them in the CSD. For an example of CSD entry generation statements for use with DSNTIAC, see job DSNTEJ5A.

The assembler source code for DSNTIAC and job DSNTEJ5A, which assembles and link-edits DSNTIAC, are in the data set prefix.SDSNSAMP.

Related tasks

Handling SQL error codes
Application programs can request more information about SQL error codes from Db2.

Related reference

GET DIAGNOSTICS (Db2 SQL)

Fortran applications that issue SQL statements

You can code SQL statements in a Fortran program wherever you can place executable statements. If the SQL statement is within an IF statement, the precompiler generates any necessary THEN and END IF statements.

Fortran source statements must be fixed-length 80-byte records. The Db2 precompiler does not support free-form source input.

Each SQL statement in a Fortran program must begin with EXEC SQL. The EXEC and SQL keywords must appear on one line, but the remainder of the statement can appear on subsequent lines.

You might code the UPDATE statement in a Fortran program as follows:

```
EXEC SQL
C  UPDATE DSN8C10.DEPT
C  SET MGRNO = :MGRNUM
C  WHERE DEPTNO = :INTDEPT
```

You cannot follow an SQL statement with another SQL statement or Fortran statement on the same line.

Fortran does not require blanks to delimit words within a statement, but the SQL language requires blanks. The rules for embedded SQL follow the rules for SQL syntax, which require you to use one or more blanks as a delimiter.

Comments

You can include Fortran comment lines within embedded SQL statements wherever you can use a blank, except between the keywords EXEC and SQL. You can include SQL comments in any embedded SQL statement. For more information, see SQL comments (Db2 SQL).
The Db2 precompiler does not support the exclamation point (!) as a comment recognition character in Fortran programs.

**Continuation for SQL statements**
The line continuation rules for SQL statements are the same as those for Fortran statements, except that you must specify EXEC SQL on one line. The SQL examples in this topic have Cs in the sixth column to indicate that they are continuations of EXEC SQL.

**Delimiters in Fortran**
Delimit an SQL statement in your Fortran program with the beginning keyword EXEC SQL and an end of line or end of last continued line.

**Declaring tables and views**
Your Fortran program should also include the DECLARE TABLE statement to describe each table and view the program accesses.

**Dynamic SQL in a Fortran program**
In general, Fortran programs can easily handle dynamic SQL statements. SELECT statements can be handled if the data types and the number of returned fields are fixed. If you want to use variable-list SELECT statements, you need to use an SQLDA, as described in “Defining SQL descriptor areas (SQLDA)” on page 476.

You can use a Fortran character variable in the statements PREPARE and EXECUTE IMMEDIATE, even if it is fixed-length.

**Including code**
To include SQL statements or Fortran host variable declarations from a member of a partitioned data set, use the following SQL statement in the source code where you want to include the statements:

```
EXEC SQL INCLUDE member-name
```

You cannot nest SQL INCLUDE statements. You cannot use the Fortran INCLUDE compiler directive to include SQL statements or Fortran host variable declarations.

**Margins**
Code the SQL statements between columns 7 through 72, inclusive. If EXEC SQL starts before the specified left margin, the Db2 precompiler does not recognize the SQL statement.

**Names**
You can use any valid Fortran name for a host variable. Do not use external entry names that begin with 'DSN' or host variable names that begin with 'SQL'. These names are reserved for Db2.

Do not use the word DEBUG, except when defining a Fortran DEBUG packet. Do not use the words FUNCTION, IMPLICIT, PROGRAM, and SUBROUTINE to define variables.

**Sequence numbers**
The source statements that the Db2 precompiler generates do not include sequence numbers.

**Statement labels**
You can specify statement numbers for SQL statements in columns 1 to 5. However, during program preparation, a labeled SQL statement generates a Fortran CONTINUE statement with that label before it generates the code that executes the SQL statement. Therefore, a labeled SQL statement should never be the last statement in a DO loop. In addition, you should not label SQL statements (such as INCLUDE and BEGIN DECLARE SECTION) that occur before the first executable SQL statement, because an error might occur.

**WHENEVER statement**
The target for the GOTO clause in the SQL WHENEVER statement must be a label in the Fortran source code and must refer to a statement in the same subprogram. The WHENEVER statement only applies to SQL statements in the same subprogram.

**Special Fortran considerations**
The following considerations apply to programs written in Fortran:
You cannot use the @PROCESS statement in your source code. Instead, specify the compiler options in the PARM field.

You cannot use the SQL INCLUDE statement to include the following statements: PROGRAM, SUBROUTINE, BLOCK, FUNCTION, or IMPLICIT.

Db2 supports Version 3 Release 1 (or later) of VS Fortran with the following restrictions:

- The parallel option is not supported. Applications that contain SQL statements must not use Fortran parallelism.
- You cannot use the byte data type within embedded SQL, because byte is not a recognizable host data type.

**Handling SQL error codes**

Fortran applications can request more information about SQL errors from Db2. For more information, see “Handling SQL error codes in C and C++ applications” on page 617.

**Related tasks**

- Overview of programming applications that access Db2 for z/OS data
- Applications that interact with Db2 must first connect to Db2. They can then read, add, or modify data or manipulate Db2 objects.
- Including dynamic SQL in your program
- Dynamic SQL is prepared and executed while the program is running.
- Handling SQL error codes
- Application programs can request more information about SQL error codes from Db2.
- Setting limits for system resource usage by using the resource limit facility (Db2 Performance)

### Defining the SQL communications area, SQLSTATE, and SQLCODE in Fortran

Fortran programs that contain SQL statements can include an SQL communications area (SQLCA) to check whether an SQL statement executed successfully. Alternatively, these programs can declare individual SQLCODE and SQLSTATE host variables.

**About this task**

If you specify the SQL processing option STDSQL(YES), do not define an SQLCA. If you do, Db2 ignores your SQLCA, and your SQLCA definition causes compile-time errors. If you specify the SQL processing option STDSQL(NO), include an SQLCA explicitly.

If your application contains SQL statements and does not include an SQL communications area (SQLCA), you must declare individual SQLCODE and SQLSTATE host variables. Your program can use these variables to check whether an SQL statement executed successfully.

**Procedure**

Choose one of the following actions:

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
</table>
| To define the SQL communications area: | a. Code the SQLCA directly in the program or use the following SQL INCLUDE statement to request a standard SQLCA declaration:  

```
EXEC SQL INCLUDE SQLCA
```

Db2 sets the SQLCODE and SQLSTATE values in the SQLCA after each SQL statement executes. Your application should check these values to determine whether the last SQL statement was successful. |
To declare SQLCODE and SQLSTATE host variables:

a. Declare the SQLCODE variable within a BEGIN DECLARE SECTION statement and an END DECLARE SECTION statement in your program declarations as INTEGER*4.
   This variable can also be called SQLCOD.

b. Declare the SQLSTATE variable within a BEGIN DECLARE SECTION statement and an END DECLARE SECTION statement in your program declarations as CHARACTER*5.
   This variable can also be called SQLCOD.

Restriction: Do not declare an SQLSTATE variable as an element of a structure.

Requirement: After you declare the SQLCODE and SQLSTATE variables, ensure that all SQL statements in the program are within the scope of the declaration of these variables.

Related tasks

Checking the execution of SQL statements
After executing an SQL statement, your program should check for any errors before you commit the data and handle the errors that they represent.

Checking the execution of SQL statements by using the SQLCA
One way to check whether an SQL statement executed successfully is to use the SQL communication area (SQLCA). This area is set apart for communication with Db2.

Checking the execution of SQL statements by using SQLCODE and SQLSTATE
Whenever an SQL statement executes, the SQLCODE and SQLSTATE fields of the SQLCA receive a return code.

Defining the items that your program can use to check whether an SQL statement executed successfully If your program contains SQL statements, the program should define some infrastructure so that it can check whether the statements executed successfully. You can either include an SQL communications area (SQLCA), which contains SQLCODE and SQLSTATE variables, or declare individual SQLCODE and SQLSTATE host variables.

Defining SQL descriptor areas in (SQLDA) Fortran
If your program includes certain SQL statements, you must define at least one SQL descriptor area (SQLDA). Depending on the context in which it is used, the SQLDA stores information about prepared SQL statements or host variables. This information can then be read by either the application program or Db2.

Procedure
Call a subroutine that is written in C, PL/I, or assembler language and that uses the INCLUDE SQLDA statement to define the SQLDA. The subroutine can also include SQL statements for any dynamic SQL functions that you need.

Restrictions:
- You must place SQLDA declarations before the first SQL statement that references the data descriptor, unless you use the TWOPASS SQL processing option.
- You cannot use the SQL INCLUDE statement for the SQLDA, because it is not supported in COBOL.

Related tasks
Defining SQL descriptor areas (SQLDA)
If your program includes certain SQL statements, you must define at least one SQL descriptor area (SQLDA). Depending on the context in which it is used, the SQLDA stores information about prepared SQL statements or host variables. This information can then be read by either the application program or Db2.

**Declaring host variables and indicator variables in Fortran**

You can use host variables, host-variable arrays, and host structures in SQL statements in your program to pass data between Db2 and your application.

**Procedure**

To declare host variables, host-variable arrays, and host structures:

1. Declare the variables according to the following rules and guidelines:
   - When you declare a character host variable, do not use an expression to define the length of the character variable. You can use a character host variable with an undefined length (for example, CHARACTER (*)). The length of any such variable is determined when the associated SQL statement executes.
   - Host variables must be scalar variables; they cannot be elements of vectors or arrays (subscripted variables).
   - Be careful when calling subroutines that might change the attributes of a host variable. Such alteration can cause an error while the program is running.
   - If you specify the ONEPASS SQL processing option, you must explicitly declare each host variable and each host-variable array before using them in an SQL statement. If you specify the TWOPASS precompiler option, you must declare each host variable before using it in the DECLARE CURSOR statement.
   - If you specify the STDSQL(YES) SQL processing option, you must precede the host language statements that define the host variables and host-variable arrays with the BEGIN DECLARE SECTION statement and follow the host language statements with the END DECLARE SECTION statement. Otherwise, these statements are optional.
   - Ensure that any SQL statement that uses a host variable or host-variable array is within the scope of the statement that declares that variable or array.
   - If you are using the Db2 precompiler, ensure that the names of host variables and host-variable arrays are unique within the program, even if the variables and variable arrays are in different blocks, classes, procedures, functions, or subroutines. You can qualify the names with a structure name to make them unique.

2. Optional: Define any associated indicator variables, arrays, and structures.

**Related tasks**

Declaring host variables and indicator variables

You can use host variables and indicator variables in SQL statements in your program to pass data between Db2 and your application.

**Host variables in Fortran**

In Fortran programs, you can specify numeric, character, LOB, and ROWID host variables. You can also specify result set and LOB locators.

**Restrictions:**

- Only some of the valid Fortran declarations are valid host variable declarations. If the declaration for a variable is not valid, any SQL statement that references the variable might result in the message UNDECLARED HOST VARIABLE.
- Fortran supports some data types with no SQL equivalent (for example, REAL*16 and COMPLEX). In most cases, you can use Fortran statements to convert between the unsupported data types and the data types that SQL allows.
- You cannot use locators as column types.
The following locator data types are Fortran data types and SQL data types:

- Result set locator
- LOB locators

- Because Fortran does not support graphic data types, Fortran applications can process only Unicode tables that use UTF-8 encoding.

Recommendations:

- Be careful of overflow. For example, if you retrieve an INTEGER column value into a INTEGER*2 host variable and the column value is larger than 32767 or -32768, you get an overflow warning or an error, depending on whether you provided an indicator variable.

- Be careful of truncation. For example, if you retrieve an 80-character CHAR column value into a CHARACTER*70 host variable, the rightmost ten characters of the retrieved string are truncated. Retrieving a double-precision floating-point or decimal column value into an INTEGER*4 host variable removes any fractional value.

**Numeric host variables**

The following diagram shows the syntax for declaring numeric host variables.

Restrictions:

- Fortran does not provide an equivalent for the decimal data type. To hold a decimal value, use one of the following variables:
  - An integer or floating-point variable, which converts the value. If you use an integer variable, you lose the fractional part of the number. If the decimal number can exceed the maximum value for an integer or you want to preserve a fractional value, use a floating-point variable. Floating-point numbers are approximations of real numbers. Therefore, when you assign a decimal number to a floating-point variable, the result might be different from the original number.
  - A character string host variable. Use the CHAR function to retrieve a decimal value into it.

- The SQL data type DECFLOAT has no equivalent in Fortran.

**Character host variables**

The following diagram shows the syntax for declaring character host variables other than CLOBs.
Result set locators
The following diagram shows the syntax for declaring result set locators.

LOB variables and locators
The following diagram shows the syntax for declaring BLOB and CLOB host variables and locators.

ROWID host variables
The following diagram shows the syntax for declarations of ROWID variables.

Constants
The syntax for constants in Fortran programs differs from the syntax for constants in SQL statements in the following ways:

- Fortran interprets a string of digits with a decimal point to be a real constant. An SQL statement interprets such a string to be a decimal constant. Therefore, use exponent notation when specifying a real (that is, floating-point) constant in an SQL statement.
• In Fortran, a real (floating-point) constant that has a length of 8 bytes uses a D as the exponent indicator (for example, 3.14159D+04). An 8-byte floating-point constant in an SQL statement must use an E (for example, 3.14159E+04).

**Related concepts**

*Host variables*

Use host variables to pass a single data item between Db2 and your application.

*Using host variables in SQL statements*

Use scalar host variables in embedded SQL statements to represent a single value. Host variables are useful for storing retrieved data or for passing values that are to be assigned or used for comparisons.

**Related tasks**

*Determining whether a retrieved value in a host variable is null or truncated*

Before your application manipulates the data that was retrieved from Db2 into a host variable, determine if the value is null. Also determine if it was truncated when assigned to the variable. You can use indicator variables to obtain this information.

*Inserting a single row by using a host variable*

Use host variables in your INSERT statement when you don’t know at least some of the values to insert until the program runs.

*Inserting null values into columns by using indicator variables or arrays*

If you need to insert null values into a column, using an indicator variable or array is an easy way to do so. An indicator variable or array is associated with a particular host variable or host-variable array.

*Storing LOB data in a table*

Db2 handles LOB data differently than it handles other kinds of data. As a result, in some cases, you need to take additional actions when you define LOB columns and insert the LOB data.

*Retrieving a single row of data into host variables*

If you know that your query returns only one row, you can specify one or more host variables to contain the column values of the retrieved row.

*Updating data by using host variables*

When you want to update a value in a Db2 table, but you do not know the exact value until the program runs, use host variables. Db2 can change a table value to match the current value of the host variable.

**Indicator variables in Fortran**

An indicator variable is a 2-byte integer (INTEGER*2). You declare indicator variables in the same way as host variables. You can mix the declarations of the two types of variables.

The following diagram shows the syntax for declaring an indicator variable in Fortran.

```
INTEGER*2, variable-name / numeric-constant /
```

**Example**

The following example shows a FETCH statement with the declarations of the host variables that are needed for the FETCH statement and their associated indicator variables.

```
EXEC SQL FETCH CLS_CURSOR INTO :CLSCD,
C :DAY :DAYIND,
C :BGN :BGNIND,
C :END :ENDIND
```

You can declare these variables as follows:
Related concepts
Indicator variables, arrays, and structures
An indicator variable is associated with a particular host variable. Each indicator variable contains a small integer value that indicates some information about the associated host variable. Indicator arrays and structures serve the same purpose for host-variable arrays and structures.

Related tasks
Inserting null values into columns by using indicator variables or arrays
If you need to insert null values into a column, using an indicator variable or array is an easy way to do so. An indicator variable or array is associated with a particular host variable or host-variable array.

Handling SQL error codes in Fortran applications
Fortran applications can request more information about SQL error codes by using the DSNTIAR subroutine or issuing a GET DIAGNOSTICS statement.

Procedure
To request more information about SQL errors from Fortran programs, use the following approaches:

- You can use the subroutine DSNTIR to convert an SQL return code into a text message. DSNTIR builds a parameter list and calls DSNTIAR for you.

  DSNTIAR takes data from the SQLCA, formats it into a message, and places the result in a message output area that you provide in your application program. For concepts and more information on the behavior of DSNTIAR, see “Displaying SQLCA fields by calling DSNTIAR” on page 529.

  **DSNTIAR syntax**
  DSNTIAR has the following syntax:

  ```fortran
  CALL DSNTIR ( error-length, message, return-code )
  ```

  **DSNTIAR parameters**
  The DSNTIR parameters have the following meanings:

  - **error-length**
    The total length of the message output area.

  - **message**
    An output area, in VARCHAR format, in which DSNTIAR places the message text. The first halfword contains the length of the remaining area; its minimum value is 240.

    The output lines of text are put into this area. For example, you could specify the format of the output area as:

    ```fortran
    INTEGER   ERRLEN /1320/
    CHARACTER*132 ERRTXT(10)
    INTEGER   ICODE
    ⋮
    CALL DSNTIR ( ERRLEN, ERRTXT, ICODE )
    ```

    where ERRLEN is the total length of the message output area, ERRTXT is the name of the message output area, and ICODE is the return code.

  - **return-code**
    Accepts a return code from DSNTIAR.

    An example of calling DSNTIR (which then calls DSNTIAR) from an application appears in the Db2 sample assembler program DSN8BF3, which is contained in the library DSN8C10.SDSNSAMP. See
“Sample applications supplied with Db2 for z/OS” on page 1024 for instructions on how to access and print the source code for the sample program.

- You can also use the MESSAGE_TEXT condition item field of the GET DIAGNOSTICS statement to convert an SQL return code into a text message. Programs that require long token message support should code the GET DIAGNOSTICS statement instead of DSNTIAR.

For more information about GET DIAGNOSTICS, see “Checking the execution of SQL statements by using the GET DIAGNOSTICS statement” on page 534.

**Related tasks**
Handling SQL error codes
Application programs can request more information about SQL error codes from Db2.

**Related reference**
GET DIAGNOSTICS (Db2 SQL)

### Equivalent SQL and Fortran data types

When you declare host variables in your Fortran programs, the precompiler uses equivalent SQL data types. When you retrieve data of a particular SQL data type into a host variable, ensure that the host variable is of an equivalent data type.

The following table describes the SQL data type and the base SQLTYPE and SQLLEN values that the precompiler uses for host variables in SQL statements.

<table>
<thead>
<tr>
<th>Fortran host variable data type</th>
<th>SQLTYPE of host variable</th>
<th>SQLLEN of host variable</th>
<th>SQL data type</th>
</tr>
</thead>
<tbody>
<tr>
<td>INTEGER*2</td>
<td>500</td>
<td>2</td>
<td>SMALLINT</td>
</tr>
<tr>
<td>INTEGER*4</td>
<td>496</td>
<td>4</td>
<td>INTEGER</td>
</tr>
<tr>
<td>REAL*4</td>
<td>480</td>
<td>4</td>
<td>FLOAT (single precision)</td>
</tr>
<tr>
<td>REAL*8</td>
<td>480</td>
<td>8</td>
<td>FLOAT (double precision)</td>
</tr>
<tr>
<td>CHARACTER*n</td>
<td>452</td>
<td>n</td>
<td>CHAR(n)</td>
</tr>
<tr>
<td>SQL TYPE IS RESULT_SET_LOCATOR</td>
<td>972</td>
<td>4</td>
<td>Result set locator. Do not use this data type as a column type.</td>
</tr>
<tr>
<td>SQL TYPE IS BLOB_LOCATOR</td>
<td>960</td>
<td>4</td>
<td>BLOB locator. Do not use this data type as a column type.</td>
</tr>
<tr>
<td>SQL TYPE IS CLOB_LOCATOR</td>
<td>964</td>
<td>4</td>
<td>CLOB locator. Do not use this data type as a column type.</td>
</tr>
<tr>
<td>SQL TYPE IS BLOB(n) 1≤n≤2147483647</td>
<td>404</td>
<td>n</td>
<td>BLOB(n)</td>
</tr>
<tr>
<td>SQL TYPE IS CLOB(n) 1≤n≤2147483647</td>
<td>408</td>
<td>n</td>
<td>CLOB(n)</td>
</tr>
<tr>
<td>SQL TYPE IS ROWID</td>
<td>904</td>
<td>40</td>
<td>ROWID</td>
</tr>
</tbody>
</table>

**Notes:**

1. If a host variable includes an indicator variable, the SQLTYPE value is the base SQLTYPE value plus 1.
The following table shows equivalent Fortran host variables for each SQL data type. Use this table to determine the Fortran data type for host variables that you define to receive output from the database. For example, if you retrieve TIMESTAMP data, you can define a variable of type CHARACTER*\(n\).

This table shows direct conversions between SQL data types and Fortran data types. However, a number of SQL data types are compatible. When you do assignments or comparisons of data that have compatible data types, Db2 converts those compatible data types.

Table 115. Fortran host variable equivalents that you can use when retrieving data of a particular SQL data type

<table>
<thead>
<tr>
<th>SQL data type</th>
<th>Fortran host variable equivalent</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>SMALLINT</td>
<td>INTEGER*2</td>
<td></td>
</tr>
<tr>
<td>INTEGER</td>
<td>INTEGER*4</td>
<td></td>
</tr>
<tr>
<td>BIGINT</td>
<td>not supported</td>
<td></td>
</tr>
<tr>
<td>DECIMAL((p, s)) or NUMERIC((p, s))</td>
<td>no exact equivalent</td>
<td>Use REAL*8</td>
</tr>
<tr>
<td>FLOAT((n)) single precision</td>
<td>REAL*4</td>
<td>1&lt;=(n)&lt;=21</td>
</tr>
<tr>
<td>FLOAT((n)) double precision</td>
<td>REAL*8</td>
<td>22&lt;=(n)&lt;=53</td>
</tr>
<tr>
<td>CHAR((n))</td>
<td>CHARACTER*(n)</td>
<td>1&lt;=(n)&lt;=255</td>
</tr>
<tr>
<td>VARCHAR((n))</td>
<td>no exact equivalent</td>
<td>Use a character host variable that is large enough to contain the largest expected VARCHAR value.</td>
</tr>
<tr>
<td>BINARY</td>
<td>not supported</td>
<td></td>
</tr>
<tr>
<td>VARBINARY</td>
<td>not supported</td>
<td></td>
</tr>
<tr>
<td>GRAPHIC((n))</td>
<td>not supported</td>
<td></td>
</tr>
<tr>
<td>VARGRAPHIC((n))</td>
<td>not supported</td>
<td></td>
</tr>
<tr>
<td>DATE</td>
<td>CHARACTER*(n)</td>
<td>If you are using a date exit routine, (n) is determined by that routine; otherwise, (n) must be at least 10.</td>
</tr>
<tr>
<td>TIME</td>
<td>CHARACTER*(n)</td>
<td>If you are using a time exit routine, (n) is determined by that routine. Otherwise, (n) must be at least 6; to include seconds, (n) must be at least 8.</td>
</tr>
<tr>
<td>TIMESTAMP</td>
<td>CHARACTER*(n)</td>
<td>(n) must be at least 19. To include microseconds, (n) must be 26; if (n) is less than 26, truncation occurs on the microseconds part.</td>
</tr>
<tr>
<td>TIMESTAMP(0)</td>
<td>CHARACTER*(n)</td>
<td>(n) must be at least 19.</td>
</tr>
<tr>
<td>TIMESTAMP((p)) (p &gt; 0)</td>
<td>CHARACTER*(n)</td>
<td>(n) must be at least 19. To include fractional seconds, (n) must be 20+(x) where (x) is the number of fractional seconds to include; if (x) is less than (p), truncation occurs on the fractional seconds part.</td>
</tr>
<tr>
<td>TIMESTAMP((p)) WITH TIME ZONE</td>
<td>no exact equivalent</td>
<td>Use a character host variable that is large enough to contain the largest expected timestamp with time zone value.</td>
</tr>
</tbody>
</table>
Table 115. Fortran host variable equivalents that you can use when retrieving data of a particular SQL data type (continued)

<table>
<thead>
<tr>
<th>SQL data type</th>
<th>Fortran host variable equivalent</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Result set locator</td>
<td>SQL TYPE IS RESULT_SET_LOCATOR</td>
<td>Use this data type only for receiving result sets. Do not use this data type as a column type.</td>
</tr>
<tr>
<td>BLOB locator</td>
<td>SQL TYPE IS BLOB_LOCATOR</td>
<td>Use this data type only to manipulate data in BLOB columns. Do not use this data type as a column type.¹</td>
</tr>
<tr>
<td>CLOB locator</td>
<td>SQL TYPE IS CLOB_LOCATOR</td>
<td>Use this data type only to manipulate data in CLOB columns. Do not use this data type as a column type.¹</td>
</tr>
<tr>
<td>DBCLOB locator</td>
<td>not supported</td>
<td></td>
</tr>
<tr>
<td>BLOB(n)</td>
<td>SQL TYPE IS BLOB(n)</td>
<td>1≤n≤2147483647¹</td>
</tr>
<tr>
<td>CLOB(n)</td>
<td>SQL TYPE IS CLOB(n)</td>
<td>1≤n≤2147483647¹</td>
</tr>
<tr>
<td>DBCLOB(n)</td>
<td>not supported</td>
<td></td>
</tr>
<tr>
<td>ROWID</td>
<td>SQL TYPE IS ROWID</td>
<td></td>
</tr>
<tr>
<td>XML</td>
<td>not supported</td>
<td></td>
</tr>
</tbody>
</table>

Related concepts
Compatibility of SQL and language data types
The host variable data types that are used in SQL statements must be compatible with the data types of the columns with which you intend to use them.

LOB host variable, LOB locator, and LOB file reference variable declarations
When you write applications to manipulate LOB data, you need to declare host variables to hold the LOB data or LOB locator. Alternatively, you need to declare LOB file reference variables to point to the LOB data.

PL/I applications that issue SQL statements
You can code SQL statements in a PL/I program wherever you can use executable statements.

The first statement of the PL/I program must be the PROCEDURE statement with OPTIONS(MAIN), unless the program is a stored procedure. A stored procedure application can run as a subroutine.

Each SQL statement in a PL/I program must begin with EXEC SQL and end with a semicolon (;). The EXEC and SQL keywords must appear must appear on one line, but the remainder of the statement can appear on subsequent lines.

You might code an UPDATE statement in a PL/I program as follows:

EXEC SQL UPDATE DSN8C10.DEPT
SET MGRNO = :MGR_NUM
WHERE DEPTNO = :INT_DEPT ;

Comments
You can include PL/I comments in embedded SQL statements wherever you can use a blank, except between the keywords EXEC and SQL. You can also include SQL comments in any SQL statement. For more information, see SQL comments (Db2 SQL).

To include DBCS characters in comments, you must delimit the characters by a shift-out and shift-in control character; the first shift-in character in the DBCS string signals the end of the DBCS string.
Continuation for SQL statements

The line continuation rules for SQL statements are the same as those for other PL/I statements, except that you must specify EXEC SQL on one line.

Delimiters for SQL statements

Delimit an SQL statement in your PL/I program with the beginning keyword EXEC SQL and a Semicolon (;).

Declaring tables and views

Your PL/I program should include a DECLARE TABLE statement to describe each table and view the program accesses. You can use the Db2 declarations generator (DCLGEN) to generate the DECLARE TABLE statements.

Including code

You can use SQL statements or PL/I host variable declarations from a member of a partitioned data set by using the following SQL statement in the source code where you want to include the statements:

```sql
EXEC SQL INCLUDE member-name;
```

You cannot nest SQL INCLUDE statements. Do not use the PL/I %INCLUDE statement to include SQL statements or host variable DCL statements. You must use the PL/I preprocessor to resolve any %INCLUDE statements before you use the Db2 precompiler. Do not use PL/I preprocessor directives within SQL statements.

Margins

Code SQL statements in columns 2 through 72, unless you have specified other margins to the Db2 precompiler. If EXEC SQL starts before the specified left margin, the Db2 precompiler does not recognize the SQL statement.

Names

You can use any valid PL/I name for a host variable. Do not use external entry names or access plan names that begin with 'DSN', and do not use host variable names that begin with 'SQL'. These names are reserved for Db2.

Sequence numbers

The source statements that the Db2 precompiler generates do not include sequence numbers. IEL0378I messages from the PL/I compiler identify lines of code without sequence numbers. You can ignore these messages.

Statement labels

You can specify a statement label for executable SQL statements. However, the INCLUDE text-file-name and END DECLARE SECTION statements cannot have statement labels.

WHENEVER statement

The target for the GOTO clause in an SQL statement WHENEVER must be a label in the PL/I source code and must be within the scope of any SQL statements that WHENEVER affects.

Using double-byte character set (DBCS) characters

The following considerations apply to using DBCS in PL/I programs with SQL statements:

- If you use DBCS in the PL/I source, Db2 rules for the following language elements apply:
  - Graphic strings
  - Graphic string constants
  - Host identifiers
  - Mixed data in character strings
  - MIXED DATA option
- The PL/I preprocessor transforms the format of DBCS constants. If you do not want that transformation, run the Db2 precompiler before the preprocessor.
- If you use graphic string constants or mixed data in dynamically prepared SQL statements, and if your application requires the PL/I Version 2 (or later) compiler, the dynamically prepared statements must use the PL/I mixed constant format.
-- If you prepare the statement from a host variable, change the string assignment to a PL/I mixed string.
-- If you prepare the statement from a PL/I string, change that to a host variable, and then change the string assignment to a PL/I mixed string.

Example:

```
SQLSTMT = 'SELECT <dbdb> FROM table-name' M;
EXEC SQL PREPARE STMT FROM :SQLSTMT;
```

• If you want a DBCS identifier to resemble a PL/I graphic string, you must use a delimited identifier.
• If you include DBCS characters in comments, you must delimit the characters with a shift-out and shift-in control character. The first shift-in character signals the end of the DBCS string.
• You can declare host variable names that use DBCS characters in PL/I application programs. The rules for using DBCS variable names in PL/I follow existing rules for DBCS SQL ordinary identifiers, except for length. The maximum length for a host variable is 128 Unicode bytes in Db2. For information about the rules for DBCS SQL ordinary identifiers, see the information about SQL identifiers.

Restrictions:

- DBCS variable names must contain DBCS characters only. Mixing single-byte character set (SBCS) characters with DBCS characters in a DBCS variable name produces unpredictable results.
- A DBCS variable name cannot continue to the next line.

• The PL/I preprocessor changes non-Kanji DBCS characters into extended binary coded decimal interchange code (EBCDIC) SBCS characters. To avoid this change, use Kanji DBCS characters for DBCS variable names, or run the PL/I compiler without the PL/I preprocessor.

Special PL/I considerations

The following considerations apply to programs written in PL/I:

• When compiling a PL/I program that includes SQL statements, you must use the PL/I compiler option CHARSET (60 EBCDIC).
• When compiling a PL/I program that uses BIGINT or LOB data types, specify the following compiler options: LIMITS(FIXEDBIN(63), FIXEDDEC(31))
• In unusual cases, the generated comments in PL/I can contain a semicolon. The semicolon generates compiler message IEL0239I, which you can ignore.
• The generated code in a PL/I declaration can contain the ADDR function of a field defined as character varying. This produces either message IBM10511 or IBM11801 W, both of which you can ignore.
• The precompiler generated code in PL/I source can contain the NULL() function. This produces message IEL0533I, which you can ignore unless you also use NULL as a PL/I variable. If you use NULL as a PL/I variable in a Db2 application, you must also declare NULL as a built-in function (DCL NULL BUILTIN;) to avoid PL/I compiler errors.
• The PL/I macro processor can generate SQL statements or host variable DCL statements if you run the macro processor before running the Db2 precompiler.

If you use the PL/I macro processor, do not use the PL/I *PROCESS statement in the source to pass options to the PL/I compiler. You can specify the needed options on the COPTION parameter of the DSNH command or the option PARM,PLI=OPTIONS of the EXEC statement in the DSNHPCLI procedure.
• Using the PL/I multitasking facility, in which multiple tasks execute SQL statements, causes unpredictable results.
• PL/I WIDECHAR host data type is supported through the Db2 coprocessor only.
• When you use PL/I WX widechar constant, Db2 supports only bigendian format. Thus, when you assign a constant to the widechar type host variable in PL/I, ensure that bigendian format is used. For example:

```
HVWC1 = '003100320033006100620063'WX;
```

Equivalent to:

```
HVWC1 = '123abc';
```

HVWC1 is defined as a WIDECHAR type host variable.

• PL/I SQL Preprocessor option, CCSID0 and NOCCSID0, usage consideration when used with the Db2 coprocessor.

  – When you use CCSID0 (default), it promotes compatibility with older PL/I programs, which used the Db2 precompiler. During program preparation, no CCSID value is associated with the host variable except for the WIDECHAR type host variable. For WIDECHAR type host variable, CCSID 1200 is always assigned by the PL/I SQL Preprocessor.

    During BIND and runtime, if no CCSID is associated with the host variable, the BIND option, ENCODING, which is meant for the application data, is used. If the ENCODING BIND option is not specified, then the default value for the ENCODING BIND option is used.

  – When you use NOCCSID0, a CCSID is associated with the host variable during program preparation. The CCSID is derived from the following items during program preparation:

    - DECLARE :hv VARIABLE CCSID xxxx specified.

    - Source CCSID, if no DECLARE VARIABLE ... CCSID xxxx is specified for the host variable. During BIND time, note the CCSID assigned to the host variable during program preparation is not known to the BIND process. For more information about BIND time CCSID resolution, see Determining the encoding scheme and CCSID of a string (Introduction to Db2 for z/OS).

    For host variable used in static SQL, ensuring accurate and matching CCSID is assigned/derived through DECLARE VARIABLE ... CCSID xxxx, source CCSID or ENCODING BIND option or the installation default.

    For parameter marker used in dynamic SQL, ensuring accurate CCSID for the corresponding host variable is assigned/derived through DECLARE VARIABLE ... CCSID xxxx, ENCODING BIND option or the installation default. The source CCSID has no influence on parameter marker.

Handling SQL error codes

PL/I applications can request more information about SQL errors from Db2. For more information, see “Handling SQL error codes in PL/I applications” on page 702.

Related concepts

DCLGEN (declarations generator)

Your program should declare the tables and views that it accesses. The Db2 declarations generator, DCLGEN, produces these DECLARE statements for C, COBOL, and PL/I programs, so that you do not need to code the statements yourself. DCLGEN also generates corresponding host variable structures.

Using host-variable arrays in SQL statements

Use host-variable arrays in embedded SQL statements to represent values that the program does not know until the query is executed. Host-variable arrays are useful for storing a set of retrieved values or for passing a set of values that are to be inserted into a table.

SQL identifiers (Db2 SQL)

Related tasks

Overview of programming applications that access Db2 for z/OS data

Applications that interact with Db2 must first connect to Db2. They can then read, add, or modify data or manipulate Db2 objects.

Including dynamic SQL in your program
Dynamic SQL is prepared and executed while the program is running.

Handling SQL error codes
Application programs can request more information about SQL error codes from Db2.

Setting limits for system resource usage by using the resource limit facility (Db2 Performance)

**PL/I programming examples**

You can write Db2 programs in PL/I. These programs can access a local or remote Db2 subsystem and can execute static or dynamic SQL statements. This information contains several such programming examples.

To prepare and run these applications, use the JCL in `prefix.SDSNSAMP` as a model for your JCL.

**Related reference**
Assembler, C, C++, COBOL, PL/I, and REXX programming examples (Db2 Programming samples)

**Example PL/I program that calls a stored procedure**

You can call the GETPRML stored procedure that uses the GENERAL WITH NULLS linkage convention from a PL/I program on a z/OS system.

The following figure contains the example PL/I program that calls the GETPRML stored procedure.
You can call a stored procedure that uses the GENERAL linkage convention from a PL/I program. This example stored procedure searches the Db2 SYSIBM.SYSROUTINES catalog table for a row that matches the input parameters from the client program. The two input parameters contain values for NAME and SCHEMA.

The linkage convention for this stored procedure is GENERAL.

The output parameters from this stored procedure contain the SQLCODE from the SELECT operation, and the value of the RUNOPTS column retrieved from the SYSIBM.SYSROUTINES table.

The CREATE PROCEDURE statement for this stored procedure might look like this:

```sql
CREATE PROCEDURE example_procedure (in_name IN VARCHAR2, in_schema IN VARCHAR2) RETURNS (out_sqlcode INTEGER, out_runopts CHAR(254)) AS
  PROCEDURE BODY;
END;
```
The following example is a PL/I stored procedure with linkage convention GENERAL.

```pli
*PROCESS SYSTEM(MVS);

GETPRML:
PROC(PROCNM, SCHEMA, OUT_CODE, PARMLST)
   OPTIONS(MAIN NOEXECOPS REENTRANT);
DECLARE PROCNM CHAR(18), /* INPUT parm -- PROCEDURE name */
   SCHEMA CHAR(8),     /* INPUT parm -- User's SCHEMA  */
   OUT_CODE FIXED BIN(31), /* OUTPUT -- SQLCODE from   */
   PARMLST CHAR(254)       /* OUTPUT -- RUNOPTS for    */
   VARYING;        /*  the matching row in     */
   /*  SYSIBM.SYSROUTINES      */
EXEC SQL INCLUDE SQLCA;

EXEC SQL
SELECT RUNOPTS INTO :PARMLST
   FROM SYSIBM.SYSROUTINES
   WHERE NAME=:PROCNM AND
   SCHEMA=:SCHEMA;

OUT_CODE = SQLCODE;         /* return SQLCODE to caller     */
RETURN;
END GETPRML;
```

**Example PL/I stored procedure with a GENERAL WITH NULLS linkage convention**

You can call a stored procedure that uses the GENERAL WITH NULLS linkage convention from a PL/I program.

This example stored procedure searches the Db2 SYSIBM.SYSROUTINES catalog table for a row that matches the input parameters from the client program. The two input parameters contain values for NAME and SCHEMA.

The linkage convention for this stored procedure is GENERAL WITH NULLS.

The output parameters from this stored procedure contain the SQLCODE from the SELECT operation, and the value of the RUNOPTS column retrieved from the SYSIBM.SYSROUTINES table.

The CREATE PROCEDURE statement for this stored procedure might look like this:

```pli
CREATE PROCEDURE GETPRML(PROCNM CHAR(18) IN, SCHEMA CHAR(8) IN,
   OUTCODE INTEGER OUT, PARMLST VARCHAR(254) OUT)
LANGUAGE PLI
DETERMINISTIC
READS SQL DATA
EXTERNAL NAME "GETPRML"
COLLID GETPRML
ASUTIME NO LIMIT
```
The following example is a PL/I stored procedure with linkage convention GENERAL WITH NULLS.

```pli
*PROCESS SYSTEM(MVS);
GETPRML:
  PROC(PROCNM, SCHEMA, OUT_CODE, PARMLST, INDICATORS)
    OPTIONS(MAIN NOEXECOPS REENTRANT);
    DECLARE PROCNM CHAR(18), /* INPUT parm -- PROCEDURE name */
       SCHEMA CHAR(8),     /* INPUT parm -- User's schema */
       OUT_CODE FIXED BIN(31), /* OUTPUT -- SQLCODE from */
               /*   the SELECT operation. */
       PARMLST CHAR(254)       /* OUTPUT -- PARMLIST for */
       VARYING;        /*  the matching row in */
       /*  SYSIBM.SYSROUTINES */
    DECLARE 1 INDICATORS,       /* Declare null indicators for */
               /* input and output parameters. */
       3 PROCNM_IND   FIXED BIN(15),
       3 SCHEMA_IND   FIXED BIN(15),
       3 OUT_CODE_IND FIXED BIN(15),
       3 PARMLST_IND  FIXED BIN(15);
    EXEC SQL INCLUDE SQLCA;
    IF PROCNM_IND<0 | SCHEMA_IND<0 THEN
      DO;
        /* If any input parm is NULL, */
        OUT_CODE = 9999;       /* Set output return code. */
        OUT_CODE_IND = 0;     /* Output return code is not NULL.*/
        PARMLST_IND = -1;     /* Assign NULL value to PARMLST. */
      END;
    ELSE
      DO;
        /* If input parms are not NULL, */
        EXEC SQL
           SELECT RUNOPTS INTO :PARMLST
               FROM SYSIBM.SYSROUTINES
               WHERE NAME=:PROCNM AND
               SCHEMA=:SCHEMA;
        PARMLST_IND = 0;       /* Mark PARMLST as not NULL. */
        OUT_CODE = SQLCODE;   /* return SQLCODE to caller */
        OUT_CODE_IND = 0;
        OUT_CODE_IND = 0;     /* Output return code is not NULL.*/
      END;
    END;
RETURN;
END GETPRML;
```

Handling SQL error codes in PL/I applications

PL/I applications can request more information about SQL error codes by using the DSNTIAR subroutine or issuing a GET DIAGNOSTICS statement.

Procedure

To request information about SQL errors in PL/I programs, use the following approaches:

- You can use the subroutine DSNTIAR to convert an SQL return code into a text message.
DSNTIAR takes data from the SQLCA, formats it into a message, and places the result in a message output area that you provide in your application program. For concepts and more information on the behavior of DSNTIAR, see “Displaying SQLCA fields by calling DSNTIAR” on page 529.

**DSNTIAR syntax**

```
CALL DSNTIAR (sqlca, message, lrecl);
```

**DSNTIAR parameters**

The DSNTIAR parameters have the following meanings:

- `sqlca`  
  An SQL communication area.

- `message`  
  An output area, in VARCHAR format, in which DSNTIAR places the message text. The first halfword contains the length of the remaining area; its minimum value is 240.

  The output lines of text, each line being the length specified in `lrecl`, are put into this area. For example, you could specify the format of the output area as:

  ```
  DCL DATA_LEN FIXED BIN(31) INIT(132);
  DCL DATA_DIM FIXED BIN(31) INIT(10);
  DCL 1 ERROR_MESSAGE AUTOMATIC,
       3 ERROR_LEN FIXED BIN(15) UNAL INIT((DATA_LEN*DATA_DIM)),
       3 ERROR_TEXT(DATA_DIM) CHAR(DATA_LEN);
  CALL DSNTIAR ( SQLCA, ERROR_MESSAGE, DATA_LEN );
  ```

  where `ERROR_MESSAGE` is the name of the message output area, `DATA_DIM` is the number of lines in the message output area, and `DATA_LEN` is the length of each line.

- `lrecl`  
  A fullword containing the logical record length of output messages, between 72 and 240.

Because DSNTIAR is an assembler language program, you must include the following directives in your PL/I application:

```
DCL DSNTIAR ENTRY OPTIONS (ASM,INTER,RETCODE);
```

An example of calling DSNTIAR from an application appears in the Db2 sample assembler program DSN8BP3, contained in the library DSN8C10.SDSNSAMP. See “Sample applications supplied with Db2 for z/OS” on page 1024 for instructions on how to access and print the source code for the sample program.

- If your CICS application requires CICS storage handling, you must use the subroutine DSNTIAC instead of DSNTIAR.

**DSNTIAC syntax**

DSNTIAC has the following syntax:

```
CALL DSNTIAC (eib, commarea, sqlca, msg, lrecl);
```

**DSNTIAC parameters**

DSNTIAC has extra parameters, which you must use for calls to routines that use CICS commands.

- `eib`  
  EXEC interface block

- `commarea`  
  communication area

For more information on these parameters, see the appropriate application programming guide for CICS. The remaining parameter descriptions are the same as those for DSNTIAR. Both DSNTIAC and DSNTIAR format the SQLCA in the same way.
You must define DSNTIA1 in the CSD. If you load DSNTIAR or DSNTIAC, you must also define them in the CSD. For an example of CSD entry generation statements for use with DSNTIAC, see job DSNTEJ5A.

The assembler source code for DSNTIAC and job DSNTEJ5A, which assembles and link-edits DSNTIAC, are in the data set prefix.SDSNSAMP.

- You can also use the MESSAGE_TEXT condition item field of the GET DIAGNOSTICS statement to convert an SQL return code into a text message. Programs that require long token message support should code the GET DIAGNOSTICS statement instead of DSNTIAR.

For more information about GET DIAGNOSTICS, see “Checking the execution of SQL statements by using the GET DIAGNOSTICS statement” on page 534.

Related tasks
Handling SQL error codes
Application programs can request more information about SQL error codes from Db2.

Related reference
GET DIAGNOSTICS (Db2 SQL)

Defining the SQL communications area, SQLSTATE, and SQLCODE in PL/I

PL/I programs that contain SQL statements can include an SQL communications area (SQLCA) to check whether an SQL statement executed successfully. Alternatively, these programs can declare individual SQLCODE and SQLSTATE host variables.

About this task

If you specify the SQL processing option STDSQL(YES), do not define an SQLCA. If you do, Db2 ignores your SQLCA, and your SQLCA definition causes compile-time errors. If you specify the SQL processing option STDSQL(NO), include an SQLCA explicitly.

If your application contains SQL statements and does not include an SQL communications area (SQLCA), you must declare individual SQLCODE and SQLSTATE host variables. Your program can use these variables to check whether an SQL statement executed successfully.

Procedure

Choose one of the following actions:

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>To define the SQL</td>
<td>a. Code the SQLCA directly in the program or use the following SQL INCLUDE statement to request a standard SQLCA declaration:</td>
</tr>
<tr>
<td>communications area:</td>
<td>EXEC SQL INCLUDE SQLCA</td>
</tr>
<tr>
<td></td>
<td>Db2 sets the SQLCODE and SQLSTATE values in the SQLCA after each SQL statement executes. Your application should check these values to determine whether the last SQL statement was successful.</td>
</tr>
<tr>
<td>To declare SQLCODE and</td>
<td>a. Declare the SQLCODE variable within a BEGIN DECLARE SECTION statement and an END DECLARE SECTION statement in your program declarations as BIN FIXED (31).</td>
</tr>
<tr>
<td>SQLSTATE host variables:</td>
<td>b. Declare the SQLSTATE variable within a BEGIN DECLARE SECTION statement and an END DECLARE SECTION statement in your program declarations as CHARACTER(5).</td>
</tr>
<tr>
<td></td>
<td><strong>Restriction:</strong> Do not declare an SQLSTATE variable as an element of a structure.</td>
</tr>
</tbody>
</table>
### Requirement
After you declare the SQLCODE and SQLSTATE variables, ensure that all SQL statements in the program are within the scope of the declaration of these variables.

### Related tasks
- Checking the execution of SQL statements
  After executing an SQL statement, your program should check for any errors before you commit the data and handle the errors that they represent.
- Checking the execution of SQL statements by using the SQLCA
  One way to check whether an SQL statement executed successfully is to use the SQL communication area (SQLCA). This area is set apart for communication with Db2.
- Checking the execution of SQL statements by using SQLCODE and SQLSTATE
  Whenever an SQL statement executes, the SQLCODE and SQLSTATE fields of the SQLCA receive a return code.
- Defining the items that your program can use to check whether an SQL statement executed successfully
  If your program contains SQL statements, the program should define some infrastructure so that it can check whether the statements executed successfully. You can either include an SQL communications area (SQLCA), which contains SQLCODE and SQLSTATE variables, or declare individual SQLCODE and SQLSTATE host variables.

### Defining SQL descriptor areas (SQLDA) in PL/I
If your program includes certain SQL statements, you must define at least one SQL descriptor area (SQLDA). Depending on the context in which it is used, the SQLDA stores information about prepared SQL statements or host variables. This information can then be read by either the application program or Db2.

#### Procedure
Code the SQLDA directly in the program, or use the following SQL INCLUDE statement to request a standard SQLDA declaration:

```sql
EXEC SQL INCLUDE SQLDA
```

**Restriction:** You must place SQLDA declarations before the first SQL statement that references the data descriptor, unless you use the TWOPASS SQL processing option.

#### Related tasks
- Defining SQL descriptor areas (SQLDA)
  If your program includes certain SQL statements, you must define at least one SQL descriptor area (SQLDA). Depending on the context in which it is used, the SQLDA stores information about prepared SQL statements or host variables. This information can then be read by either the application program or Db2.

#### Related reference
SQL descriptor area (SQLDA) (Db2 SQL)

### Declaring host variables and indicator variables in PL/I
You can use host variables, host-variable arrays, and host structures in SQL statements in your program to pass data between Db2 and your application.

#### Procedure
To declare host variables, host-variable arrays, and host structures:
1. Declare the variables according to the following rules and guidelines:
• If you specify the ONEPASS SQL processing option, you must explicitly declare each host variable and each host-variable array before using them in an SQL statement. If you specify the TWOPASS precompiler option, you must declare each host variable before using it in the DECLARE CURSOR statement.

• If you specify the STDSQL(YES) SQL processing option, you must precede the host language statements that define the host variables and host-variable arrays with the BEGIN DECLARE SECTION statement and follow the host language statements with the END DECLARE SECTION statement. Otherwise, these statements are optional.

• Ensure that any SQL statement that uses a host variable or host-variable array is within the scope of the statement that declares that variable or array.

• If you are using the Db2 precompiler, ensure that the names of host variables and host-variable arrays are unique within the program, even if the variables and variable arrays are in different blocks, classes, procedures, functions, or subroutines. You can qualify the names with a structure name to make them unique.

2. Optional: Define any associated indicator variables, arrays, and structures.

Related tasks
Declaring host variables and indicator variables
You can use host variables and indicator variables in SQL statements in your program to pass data between Db2 and your application.

**Host variables in PL/I**

In PL/I programs, you can specify numeric, character, graphic, binary, LOB, XML, and ROWID host variables. You can also specify result set, table, and LOB locators and LOB and XML file reference variables.

**Restrictions:**

• Only some of the valid PL/I declarations are valid host variable declarations. The precompiler uses the data attribute defaults that are specified in the PL/I DEFAULT statement. If the declaration for a host variable is not valid, any SQL statement that references the variable might result in the message UNDECLARED HOST VARIABLE.

• The alignment, scope, and storage attributes of host variables have the following restrictions:
  – A declaration with the EXTERNAL scope attribute and the STATIC storage attribute must also have the INITIAL storage attribute.
  – If you use the BASED storage attribute, you must follow it with a PL/I element-locator-expression.
  – Host variables can be STATIC, CONTROLLED, BASED, or AUTOMATIC storage class, or options. However, CICS requires that programs be reentrant.

Although the precompiler uses only the names and data attributes of variables and ignores the alignment, scope, and storage attributes, you should not ignore these restrictions. If you do ignore them, you might have problems compiling the PL/I source code that the precompiler generates.

• PL/I supports some data types with no SQL equivalent (COMPLEX and BIT variables, for example). In most cases, you can use PL/I statements to convert between the unsupported PL/I data types and the data types that SQL supports.

• You can not use locators as column types.

  The following locator data types are PL/I data types as well as SQL data types:
  – Result set locator
  – Table locator
  – LOB locators

• The precompiler does not support PL/I scoping rules.

**Recommendations:**
• Be careful of overflow. For example, if you retrieve an INTEGER column value into a BIN FIXED(15) host variable and the column value is larger than 32767 or smaller than -32768, you get an overflow warning or an error, depending on whether you provided an indicator variable.

• Be careful of truncation. For example, if you retrieve an 80-character CHAR column value into a CHAR(70) host variable, the rightmost ten characters of the retrieved string are truncated. Retrieving a double-precision floating-point or decimal column value into a BIN FIXED(31) host variable removes any fractional part of the value. Similarly, retrieving a column value with a DECIMAL data type into a PL/I decimal variable with a lower precision might truncate the value.

**Numeric host variables**

You can specify the following forms of numeric host variables:

• Floating-point numbers (Hexadecimal and Decimal)
• Integers and small integers
• Decimal numbers

The following diagram shows the syntax for declaring numeric host variables.

![Diagram showing syntax for declaring numeric host variables.

Notes:

1. You can specify a scale only for DECIMAL FIXED.
2. You can specify host variable attributes in any order that is acceptable to PL/I. For example, BIN FIXED(31), BINARY FIXED(31), BIN(31) FIXED, and FIXED BIN(31) are all acceptable.
3. The UNSIGNED attribute must not be specified with BINARY and FIXED for a numeric host variable declaration.

For binary floating-point or hexadecimal floating-point data types, use the FLOAT SQL processing option to specify whether the host variable is in IEEE binary floating-point or z/Architecture hexadecimal floating-point format. Db2 does not check if the format of the host variable contents match the format that you specified with the FLOAT SQL processing option. Therefore, you need to ensure that your floating-point host variable contents match the format that you specified with the FLOAT SQL processing option. Db2 converts all floating-point input data to z/Architecture hexadecimal floating-point format before storing it.

If the PL/I compiler that you are using does not support a decimal data type with a precision greater than 15, use one of the following variable types for decimal data:

• Decimal variables with precision less than or equal to 15, if the actual data values fit. If you retrieve a decimal value into a decimal variable with a scale that is less than the source column in the database, the fractional part of the value might truncate.
An integer or a floating-point variable, which converts the value. If you use an integer variable, you lose the fractional part of the number. If the decimal number can exceed the maximum value for an integer or you want to preserve a fractional value, use a floating-point variable. Floating-point numbers are approximations of real numbers. Therefore, when you assign a decimal number to a floating-point variable, the result might be different from the original number.

A character string host variable. Use the CHAR function to retrieve a decimal value into it.

To use the PL/I decimal floating-point host data types, you need to use the FLOAT(DFP) and ARCH(7) compiler options and the Db2 coprocessor. The maximum precision for extended DECIMAL FLOAT will be 34 (not 33 as it is for hexadecimal float). The maximum precision for short DECIMAL FLOAT will be 7 (not 6 as it is for hexadecimal float).

**Character host variables**

You can specify the following forms of character host variables:

- Fixed-length strings
- Varying-length strings
- CLOBs

The following diagram shows the syntax for declaring character host variables, other than CLOBs.

```
DECLARE
    DCL variable-name Character CHAR (length)
    VARYING VAR Alignment and/or Scope and/or Storage
```

**Graphic host variables**

You can specify the following forms of character host variables:

- Fixed-length strings
- Varying-length strings
- DBCLOBs

The following diagram shows the syntax for declaring graphic host variables other than DBCLOBs.

```
DECLARE
    DCL variable-name Graphic Widechar 1 (length)
    VARYING VAR Alignment and/or Scope and/or Storage
```

Notes:

1 Use WIDECHAR only for UNICODE UTF-16 data. WIDECHAR is supported only by the Db2 coprocessor.
**Binary host variables**

You can specify the following forms of binary host variables:

- Fixed-length strings
- Varying-length strings
- BLOBs

The following diagram shows the syntax for declaring BINARY host variables.

![Syntax diagram for BINARY host variables](image)

Notes:

1. For BINARY host variables, the length must be in the range 1 - 255. For VARBINARY host variables, the length must be in the range 1 - 32704.

PL/I does not have variables that correspond to the SQL binary data types BINARY and VARBINARY. To create host variables that can be used with these data types, use the SQL TYPE IS clause.

When you reference a BINARY or VARBINARY host variable in an SQL statement, you must use the variable that you specify in the SQL TYPE declaration. When you reference the host variable in a host language statement, you must use the variable that Db2 generates.

**Examples of binary variable declarations**

The following table shows examples of variables that Db2 generates when you declare binary host variables.

<table>
<thead>
<tr>
<th>Variable declaration that you include in your PL/I program</th>
<th>Corresponding variable that Db2 generates in the output source member</th>
</tr>
</thead>
<tbody>
<tr>
<td>DCL BIN_VAR SQL TYPE IS BINARY(10);</td>
<td>DCL BIN_VAR CHAR(10);</td>
</tr>
<tr>
<td>DCL VBIN_VAR SQL TYPE IS VARBINARY(10);</td>
<td>DCL VBIN_VAR CHAR(10) VAR;</td>
</tr>
</tbody>
</table>

**Result set locators**

The following diagram shows the syntax for declaring result set locators.
Table locators
The following diagram shows the syntax for declaring table locators.

LOB variables, locators, and file reference variables
The following diagram shows the syntax for declaring BLOB, CLOB, and DBCLOB host variables, locators, and file reference variables.

Notes:
1 A single PL/I declaration that contains a LOB variable declaration is limited to no more than 1000 lines of source code.
2 Variable attributes such as STATIC and AUTOMATIC are ignored if specified on a LOB variable declaration.
**Note:** Variable attributes such as STATIC and AUTOMATIC are ignored if specified on a LOB variable declaration.

**XML data host and file reference variables**

The following diagram shows the syntax for declaring BLOB, CLOB, and DBCLOB host variables and file reference variables for XML data types.

**ROWID host variables**

The following diagram shows the syntax for declaring ROWID host variables.

**Related concepts**

Host variables
Use host variables to pass a single data item between Db2 and your application.

Using host variables in SQL statements
Use scalar host variables in embedded SQL statements to represent a single value. Host variables are useful for storing retrieved data or for passing values that are to be assigned or used for comparisons.

Decimal floating-point (DECFLOAT) (Db2 SQL)

**Related tasks**

Determining whether a retrieved value in a host variable is null or truncated
Before your application manipulates the data that was retrieved from Db2 into a host variable, determine if the value is null. Also determine if it was truncated when assigned to the variable. You can use indicator variables to obtain this information.

Inserting a single row by using a host variable
Use host variables in your INSERT statement when you don't know at least some of the values to insert until the program runs.

Inserting null values into columns by using indicator variables or arrays
If you need to insert null values into a column, using an indicator variable or array is an easy way to do so. An indicator variable or array is associated with a particular host variable or host-variable array.

Storing LOB data in a table
Db2 handles LOB data differently than it handles other kinds of data. As a result, in some cases, you need to take additional actions when you define LOB columns and insert the LOB data.

Retrieving a single row of data into host variables
If you know that your query returns only one row, you can specify one or more host variables to contain the column values of the retrieved row.

Retrieving a single row of data into a host structure
If you know that your query returns multiple column values for only one row, you can specify a host structure to contain the column values.

Updating data by using host variables
When you want to update a value in a Db2 table, but you do not know the exact value until the program runs, use host variables. Db2 can change a table value to match the current value of the host variable.

**Host-variable arrays in PL/I**

In PL/I programs, you can specify numeric, character, graphic, binary, LOB, XML, and ROWID host-variable arrays. You can also specify LOB locators and LOB and XML file reference variables.

Host-variable arrays can be referenced only as a simple reference in the following contexts. In syntax diagrams, host-variable-array designates a reference to a host-variable array.

- In a FETCH statement for a multiple-row fetch. See FETCH (Db2 SQL).
- In an INSERT statement with multiple rows of source data. This is also referred to as multiple-row insert. See INSERT (Db2 SQL).
- In a MERGE statement with multiple rows of source data. See MERGE (Db2 SQL).
- In an EXECUTE statement to provide a value for a parameter marker in a dynamic multi-row INSERT or MERGE. See EXECUTE (Db2 SQL).

**Restrictions:**

- Only some of the valid PL/I declarations are valid host variable declarations. The precompiler uses the data attribute defaults that are specified in the PL/I DEFAULT statement. If the declaration for a host variable is not valid, any SQL statement that references the host-variable array might result in the message UNDECLARED HOST VARIABLE ARRAY.
- The alignment, scope, and storage attributes of host-variable arrays have the following restrictions:
  - A declaration with the EXTERNAL scope attribute and the STATIC storage attribute must also have the INITIAL storage attribute.
  - If you use the BASED storage attribute, you must follow it with a PL/I element-locator-expression.
  - Host variables can be STATIC, CONTROLLED, BASED, or AUTOMATIC storage class, or options. However, CICS requires that programs be reentrant.

Although the precompiler uses only the names and data attributes of variable arrays and ignores the alignment, scope, and storage attributes, you should not ignore these restrictions. If you do ignore them, you might have problems compiling the PL/I source code that the precompiler generates.

- You must specify the ALIGNED attribute when you declare varying-length character arrays or varying-length graphic arrays that are to be used in multiple-row INSERT and FETCH statements.

**Numeric host-variable arrays**

The following diagram shows the syntax for declaring numeric host-variable arrays.
Example

The following example shows a declaration of an indicator array.

```
DCL IND_ARRAY(100) BIN FIXED(15);   /* DCL ARRAY of 100 indicator variables */
```

To use the PL/I decimal floating-point host data types, you need to use the FLOAT(DFP) and ARCH(7) compiler options and the Db2 coprocessor. The maximum precision for extended DECIMAL FLOAT will be 34 (not 33 as it is for hexadecimal float). The maximum precision for short DECIMAL FLOAT will be 7 (not 6 as it is for hexadecimal float).

Character host-variable arrays

The following diagram shows the syntax for declaring character host-variable arrays other than CLOBs.
Example
The following example shows the declarations needed to retrieve 10 rows of the department number and name from the department table:

```
DCL DEPTNO(10)    CHAR(3);  /* Array of ten CHAR(3) variables */
DCL DEPTNAME(10)  CHAR(29) VAR; /* Array of ten VARCHAR(29) variables */
```

Graphic host-variable arrays
The following diagram shows the syntax for declaring graphic host-variable arrays other than DBCLOBs.

Binary host-variable arrays
The following diagram shows the syntax for declaring binary variable arrays.

Notes:
1 $dimension$ must be an integer constant between 1 and 32767.
LOB, locator, and file reference variable arrays

The following diagram shows the syntax for declaring BLOB, CLOB, and DBCLOB host variable, locator, and file reference variable arrays.

Notes:

1. dimension must be an integer constant between 1 and 32767.

XML host and file reference variable arrays

The following diagram shows the syntax for declaring BLOB, CLOB, and DBCLOB host-variable arrays and file reference variable arrays for XML data types.
**DCL**

```
DECLARE variable-name ( dimension )
```

SQL TYPE IS XML AS

```
BINARY LARGE OBJECT ( length K M G )
```

Notes:

1. `dimension` must be an integer constant between 1 and 32767.

**ROWID variable arrays**

The following diagram shows the syntax for declaring ROWID variable arrays.

```
DECLARE variable-name ( dimension )
```

SQL TYPE IS ROWID

Notes:

1. `dimension` must be an integer constant between 1 and 32767.

**Related concepts**

Using host-variable arrays in SQL statements

Use host-variable arrays in embedded SQL statements to represent values that the program does not know until the query is executed. Host-variable arrays are useful for storing a set of retrieved values or for passing a set of values that are to be inserted into a table.

Host-variable arrays

You can use host-variable arrays to pass a data array between Db2 and your application. A host-variable array is a data array that is declared in the host language to be used within an SQL statement.

Host-variable arrays in PL/I, C, C++, and COBOL (Db2 SQL)
Decimal floating-point (DECFLOAT) (Db2 SQL)

Related tasks
Inserting multiple rows of data from host-variable arrays
Use host-variable arrays in your INSERT statement when you do not know at least some of the values to insert until the program runs.

Storing LOB data in a table
Db2 handles LOB data differently than it handles other kinds of data. As a result, in some cases, you need to take additional actions when you define LOB columns and insert the LOB data.

Retrieving multiple rows of data into host-variable arrays
If you know that your query returns multiple rows, you can specify host-variable arrays to store the retrieved column values.

Host structures in PL/I
A PL/I host structure is a structure that contains subordinate levels of scalars. You can use the name of the structure as shorthand notation to reference the list of scalars.

Requirements: Host structure declarations in PL/I must satisfy the following requirements:

• Host structures are limited to two levels.
• You must terminate the host structure variable by ending the declaration with a semicolon, as in the following example:

```
DCL 1 A,
   2 B CHAR,
   2 (C, D) CHAR;
DCL (E, F) CHAR;
```

• You can specify host variable attributes in any order that is acceptable to PL/I. For example, BIN FIXED(31), BIN(31) FIXED, and FIXED BIN(31) are all acceptable.

When you reference a host variable, you can qualify it with a structure name. For example, you can specify STRUCTURE.FIELD.

Host structures
The following diagram shows the syntax for declaring host structures.

Data types
The following diagram shows the syntax for data types that are used within declarations of host structures.
LOB data types

The following diagram shows the syntax for LOB data types that are used within declarations of host structures.

LOB data types for XML data

The following diagram shows the syntax for LOB data types that are used within declarations of host structures for XML data.
Example

In the following example, B is the name of a host structure that contains the scalars C1 and C2.

```
DCL  1 A,
    2 B,
    3 C1 CHAR(...),
    3 C2 CHAR(...);
```

Related concepts

Host structures
Use host structures to pass a group of host variables between Db2 and your application.

Indicator variables in PL/I

An indicator variable is a 2-byte integer (or an integer declared as BIN FIXED(15)). An indicator variable array is an array of 2-byte integers. You declare indicator variables in the same way as host variables. You can mix the declarations of the two types of variables.

The following diagram shows the syntax for declaring an indicator variable in PL/I.

```
DECLARE DCL (variable-name) BINARY FIXED(15) ;
```

Notes:

1 You can specify host variable attributes in any order that is acceptable to PL/I. For example, BIN FIXED(31), BIN(31) FIXED, and FIXED BIN(31) are all acceptable.

The following diagram shows the syntax for declaring an indicator array in PL/I.

```
DECLARE DCL variable-name (dimension) BINARY FIXED(15) ;
```
Example

The following example shows a FETCH statement with the declarations of the host variables that are needed for the FETCH statement and their associated indicator variables.

EXEC SQL FETCH CLS_CURSOR INTO :CLS_CD,
       :DAY :DAY_IND,
       :BGN :BGN_IND,
       :END :END_IND;

You can declare these variables as follows:

DCL CLS_CD    CHAR(7);
DCL DAY       BIN FIXED(15);
DCL BGN       CHAR(8);
DCL END       CHAR(8);
DCL (DAY_IND, BGN_IND, END_IND)   BIN FIXED(15);

Related concepts
Indicator variables, arrays, and structures
An indicator variable is associated with a particular host variable. Each indicator variable contains a small integer value that indicates some information about the associated host variable. Indicator arrays and structures serve the same purpose for host-variable arrays and structures.

Related tasks
Inserting null values into columns by using indicator variables or arrays
If you need to insert null values into a column, using an indicator variable or array is an easy way to do so. An indicator variable or array is associated with a particular host variable or host-variable array.

Equivalent SQL and PL/I data types

When you declare host variables in your PL/I programs, the precompiler uses equivalent SQL data types. When you retrieve data of a particular SQL data type into a host variable, you need to ensure that the host variable is of an equivalent data type.

The following table describes the SQL data type and the base SQLTYPE and SQLLEN values that the precompiler uses for host variables in SQL statements.

<table>
<thead>
<tr>
<th>PL/I host variable data type</th>
<th>SQLTYPE of host variable</th>
<th>SQLLEN of host variable</th>
<th>SQL data type</th>
</tr>
</thead>
<tbody>
<tr>
<td>BIN FIXED(n) 1≤n≤15</td>
<td>500</td>
<td>2</td>
<td>SMALLINT</td>
</tr>
<tr>
<td>BIN FIXED(n) 16≤n≤31</td>
<td>496</td>
<td>4</td>
<td>INTEGER</td>
</tr>
<tr>
<td>FIXED BIN(63)</td>
<td>492</td>
<td>8</td>
<td>BIGINT</td>
</tr>
<tr>
<td>DEC FIXED(p,s) 0≤p≤31 and 0≤ss ≤31</td>
<td>484</td>
<td>p in byte 1, s in byte 2</td>
<td>DECIMAL(p,s)</td>
</tr>
<tr>
<td>DEC FLOAT (p) where 1 ≤ p ≤ 7</td>
<td>996/997</td>
<td>4</td>
<td>DECFLOAT(16)</td>
</tr>
<tr>
<td>DEC FLOAT (p) where 8 ≤ p ≤ 16</td>
<td>996/997</td>
<td>8</td>
<td>DECFLOAT(16)</td>
</tr>
<tr>
<td>DEC FLOAT (p) where 17 ≤ p</td>
<td>996/997</td>
<td>16</td>
<td>DECFLOAT(34)</td>
</tr>
</tbody>
</table>

1 *dimension* must be an integer constant between 1 and 32767.
Table 117. SQL data types, SQLLEN values, and SQLTYPE values that the precompiler uses for host variables in PL/I programs (continued)

<table>
<thead>
<tr>
<th>PL/I host variable data type</th>
<th>SQLTYPE of host variable “1” on page 724</th>
<th>SQLLEN of host variable</th>
<th>SQL data type</th>
</tr>
</thead>
<tbody>
<tr>
<td>BIN FLOAT(p) 1≤p≤21</td>
<td>480</td>
<td>4</td>
<td>REAL or FLOAT(n) 1≤n≤21</td>
</tr>
<tr>
<td>BIN FLOAT(p) 22≤p≤53</td>
<td>480</td>
<td>8</td>
<td>DOUBLE PRECISION or FLOAT(n) 22≤n≤53</td>
</tr>
<tr>
<td>DEC FLOAT(m) 1≤m≤6</td>
<td>480</td>
<td>4</td>
<td>FLOAT (single precision)</td>
</tr>
<tr>
<td>DEC FLOAT(m) 7≤m≤16</td>
<td>480</td>
<td>8</td>
<td>FLOAT (double precision)</td>
</tr>
<tr>
<td>CHAR(n)</td>
<td>452</td>
<td>n</td>
<td>CHAR(n)</td>
</tr>
<tr>
<td>CHAR(n) VARYING 1≤n≤255</td>
<td>448</td>
<td>n</td>
<td>VARCHAR(n)</td>
</tr>
<tr>
<td>CHAR(n) VARYING n&gt;255</td>
<td>456</td>
<td>n</td>
<td>VARCHAR(n)</td>
</tr>
<tr>
<td>GRAPHIC(n)</td>
<td>468</td>
<td>n</td>
<td>GRAPHIC(n)</td>
</tr>
<tr>
<td>GRAPHIC VARYING(n)</td>
<td>464</td>
<td>n</td>
<td>VARGRAPHIC(n)</td>
</tr>
<tr>
<td>SQL TYPE IS BINARY(n), 1≤n≤255</td>
<td>912</td>
<td>n</td>
<td>BINARY(n)</td>
</tr>
<tr>
<td>SQL TYPE IS VARBINARY(n), 1≤n≤32704</td>
<td>908</td>
<td>n</td>
<td>VARBINARY(n)</td>
</tr>
<tr>
<td>SQL TYPE IS RESULT_SET_LOCATOR</td>
<td>972</td>
<td>4</td>
<td>Result set locator “3” on page 724</td>
</tr>
<tr>
<td>SQL TYPE IS TABLE LIKE tablename AS LOCATOR</td>
<td>976</td>
<td>4</td>
<td>Table locator “3” on page 724</td>
</tr>
<tr>
<td>SQL TYPE IS BLOB_LOCATOR</td>
<td>960</td>
<td>4</td>
<td>BLOB locator “3” on page 724</td>
</tr>
<tr>
<td>SQL TYPE IS CLOB_LOCATOR</td>
<td>964</td>
<td>4</td>
<td>CLOB locator “3” on page 724</td>
</tr>
<tr>
<td>SQL TYPE IS DBCLOB_LOCATOR</td>
<td>968</td>
<td>4</td>
<td>DBCLOB locator “3” on page 724</td>
</tr>
<tr>
<td>SQL TYPE IS BLOB(n) 1≤n≤2147483647</td>
<td>404</td>
<td>n</td>
<td>BLOB(n)</td>
</tr>
<tr>
<td>SQL TYPE IS CLOB(n) 1≤n≤2147483647</td>
<td>408</td>
<td>n</td>
<td>CLOB(n)</td>
</tr>
<tr>
<td>SQL TYPE IS DBCLOB(n) 1≤n≤1073741823 “4” on page 724</td>
<td>412</td>
<td>n</td>
<td>DBCLOB(n) “4” on page 724</td>
</tr>
<tr>
<td>SQL TYPE IS XML AS BLOB(n)</td>
<td>404</td>
<td>0</td>
<td>XML</td>
</tr>
<tr>
<td>SQL TYPE IS XML AS CLOB(n)</td>
<td>408</td>
<td>0</td>
<td>XML</td>
</tr>
<tr>
<td>SQL TYPE IS XML AS DBCLOB(n)</td>
<td>412</td>
<td>0</td>
<td>XML</td>
</tr>
<tr>
<td>SQL TYPE IS BLOB_FILE</td>
<td>916/917</td>
<td>267</td>
<td>BLOB file reference “3” on page 724</td>
</tr>
<tr>
<td>SQL TYPE IS CLOB_FILE</td>
<td>920/921</td>
<td>267</td>
<td>CLOB file reference “3” on page 724</td>
</tr>
</tbody>
</table>
### Table 117. SQL data types, SQLLEN values, and SQLTYPE values that the precompiler uses for host variables in PL/I programs (continued)

<table>
<thead>
<tr>
<th>PL/I host variable data type</th>
<th>SQLTYPE of host variable</th>
<th>SQLLEN of host variable</th>
<th>SQL data type</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQL TYPE IS DBCLOB_FILE</td>
<td>924/925</td>
<td>267</td>
<td>DBCLOB file reference “3” on page 724</td>
</tr>
<tr>
<td>SQL TYPE IS XML AS BLOB_FILE</td>
<td>916/917</td>
<td>267</td>
<td>XML BLOB file reference “3” on page 724</td>
</tr>
<tr>
<td>SQL TYPE IS XML AS CLOB_FILE</td>
<td>920/921</td>
<td>267</td>
<td>XML CLOB file reference “3” on page 724</td>
</tr>
<tr>
<td>SQL TYPE IS XML AS DBCLOB_FILE</td>
<td>924/925</td>
<td>267</td>
<td>XML DBCLOB file reference “3” on page 724</td>
</tr>
<tr>
<td>SQL TYPE IS ROWID</td>
<td>904</td>
<td>40</td>
<td>ROWID</td>
</tr>
<tr>
<td>WIDECHAR(n)</td>
<td>468</td>
<td>n</td>
<td>GRAPHIC(n) &quot;5” on page 724</td>
</tr>
<tr>
<td>WIDECHAR VARYING(n)</td>
<td>464</td>
<td>n</td>
<td>VARGRAPHIC(n) &quot;5” on page 724</td>
</tr>
</tbody>
</table>

The following table shows equivalent PL/I host variables for each SQL data type. Use this table to determine the PL/I data type for host variables that you define to receive output from the database. For example, if you retrieve TIMESTAMP data, you can define a variable of type CHAR(n).

This table shows direct conversions between SQL data types and PL/I data types. However, a number of SQL data types are compatible. When you do assignments or comparisons of data that have compatible data types, Db2 converts those compatible data types.

### Table 118. PL/I host variable equivalents that you can use when retrieving data of a particular SQL data type

<table>
<thead>
<tr>
<th>SQL data type</th>
<th>PL/I host variable equivalent</th>
<th>Remarks</th>
</tr>
</thead>
<tbody>
<tr>
<td>SMALLINT</td>
<td>BIN FIXED(n)</td>
<td>1≤n≤15</td>
</tr>
<tr>
<td>INTEGER</td>
<td>BIN FIXED(n)</td>
<td>16≤n≤31</td>
</tr>
<tr>
<td>BIGINT</td>
<td>FIXED BIN(63)</td>
<td>“7” on page 724</td>
</tr>
<tr>
<td>DECIMAL(p,s) or NUMERIC(p,s)</td>
<td>If p&lt;16: DEC FIXED(p) or DEC FIXED(p,s)</td>
<td>p is precision; s is scale. 1≤ps≤31 and 0≤s≤p</td>
</tr>
<tr>
<td>DEC FLOAT(16)</td>
<td>DEC FLOAT (p)</td>
<td>1≤p≤7</td>
</tr>
<tr>
<td>DEC FLOAT(16)</td>
<td>DEC FLOAT (p)</td>
<td>8≤p≤16</td>
</tr>
<tr>
<td>DEC FLOAT(34)</td>
<td>DEC FLOAT (p)</td>
<td>17≤p</td>
</tr>
<tr>
<td>REAL or FLOAT(n)</td>
<td>BIN FLOAT(p) or DEC FLOAT(m)</td>
<td>1≤ns21, 1≤ps21, and 1≤ms6</td>
</tr>
<tr>
<td>DOUBLE PRECISION, DOUBLE, or FLOAT(n)</td>
<td>BIN FLOAT(p) or DEC FLOAT(m)</td>
<td>22≤ns53, 22≤ps53, and 7≤ms16</td>
</tr>
<tr>
<td>CHAR(n)</td>
<td>CHAR(n)</td>
<td>1≤ns255</td>
</tr>
<tr>
<td>VARCHAR(n)</td>
<td>CHAR(n) VAR</td>
<td></td>
</tr>
<tr>
<td>GRAPHIC(n)</td>
<td>GRAPHIC(n) or WIDECHAR(n) &quot;2” on page 724</td>
<td>n refers to the number of double-byte characters, not to the number of bytes.</td>
</tr>
<tr>
<td>SQL data type</td>
<td>PL/I host variable equivalent</td>
<td>Remarks</td>
</tr>
<tr>
<td>---------------------</td>
<td>-------------------------------</td>
<td>-------------------------------------------------------------------------</td>
</tr>
<tr>
<td>VARGRAPHIC(n)</td>
<td>GRAPHIC(n) VARYING or WIDECHAR(n) VARYING</td>
<td>( n ) refers to the number of double-byte characters, not to the number of bytes.</td>
</tr>
<tr>
<td>BINARY(n)</td>
<td>SQL TYPE IS BINARY(n)</td>
<td>( 1 \leq n \leq 255 )</td>
</tr>
<tr>
<td>VARBINARY(n)</td>
<td>SQL TYPE IS VARBINARY(n)</td>
<td>( 1 \leq n \leq 32704 )</td>
</tr>
<tr>
<td>DATE</td>
<td>CHAR(n)</td>
<td>If you are using a date exit routine, that routine determines ( n ); otherwise, ( n ) must be at least 10.</td>
</tr>
<tr>
<td>TIME</td>
<td>CHAR(n)</td>
<td>If you are using a time exit routine, that routine determines ( n ). Otherwise, ( n ) must be at least 6; to include seconds, ( n ) must be at least 8.</td>
</tr>
<tr>
<td>TIMESTAMP</td>
<td>CHAR(n)</td>
<td>( n ) must be at least 19. To include microseconds, ( n ) must be 26; if ( n ) is less than 26, the microseconds part is truncated.</td>
</tr>
<tr>
<td>TIMESTAMP(0)</td>
<td>CHAR(n)</td>
<td>( n ) must be at least 19.</td>
</tr>
<tr>
<td>TIMESTAMP(p) ( p &gt; 0 )</td>
<td>CHAR(n)</td>
<td>( n ) must be at least 19. To include fractional seconds, ( n ) must be ( 20 + x ) where ( x ) is the number of fractional seconds to include; if ( x ) is less than ( p ), truncation occurs on the fractional seconds part.</td>
</tr>
<tr>
<td>TIMESTAMP(0) WITH TIME ZONE</td>
<td>CHAR(n) VAR</td>
<td>( n ) must be at least 25.</td>
</tr>
<tr>
<td>TIMESTAMP(p) WITH TIME ZONE</td>
<td>CHAR(n) VAR</td>
<td>( n ) must be at least ( 26 + p ).</td>
</tr>
<tr>
<td>Result set locator</td>
<td>SQL TYPE IS RESULT_SET_LOCATOR</td>
<td>Use this data type only for receiving result sets. ( &quot;3&quot; ) on page 724</td>
</tr>
<tr>
<td>Table locator</td>
<td>SQL TYPE IS TABLE LIKE table-name AS LOCATOR</td>
<td>Use this data type only in a user-defined function or stored procedure to receive rows of a transition table. ( &quot;3&quot; ) on page 724</td>
</tr>
<tr>
<td>BLOB locator</td>
<td>SQL TYPE IS BLOB_LOCATOR</td>
<td>Use this data type only to manipulate data in BLOB columns. ( &quot;3&quot; ) on page 724, ( &quot;6&quot; ) on page 724, ( &quot;7&quot; ) on page 724</td>
</tr>
<tr>
<td>CLOB locator</td>
<td>SQL TYPE IS CLOB_LOCATOR</td>
<td>Use this data type only to manipulate data in CLOB columns. ( &quot;3&quot; ) on page 724, ( &quot;6&quot; ) on page 724, ( &quot;7&quot; ) on page 724</td>
</tr>
<tr>
<td>DBCLOB locator</td>
<td>SQL TYPE IS DBCLOB_LOCATOR</td>
<td>Use this data type only to manipulate data in DBCLOB columns. ( &quot;3&quot; ) on page 724, ( &quot;6&quot; ) on page 724, ( &quot;7&quot; ) on page 724</td>
</tr>
<tr>
<td>BLOB(n)</td>
<td>SQL TYPE IS BLOB(n)</td>
<td>( 1 \leq n \leq 2147483647 )</td>
</tr>
</tbody>
</table>
Table 118. PL/I host variable equivalents that you can use when retrieving data of a particular SQL data type (continued)

<table>
<thead>
<tr>
<th>SQL data type</th>
<th>PL/I host variable equivalent</th>
<th>Remarks</th>
</tr>
</thead>
<tbody>
<tr>
<td>CLOB(n)</td>
<td>SQL TYPE IS CLOB(n)</td>
<td>1≤n≤2147483647 &quot;6&quot; on page 724, &quot;7&quot; on page 724</td>
</tr>
<tr>
<td>DBCLOB(n)</td>
<td>SQL TYPE IS DBCLOB(n)</td>
<td>n is the number of double-byte characters. 1≤n≤1073741823 &quot;5&quot; on page 724, &quot;7&quot; on page 724</td>
</tr>
<tr>
<td>XML</td>
<td>SQL TYPE IS XML AS BLOB(n)</td>
<td>1≤n≤2147483647</td>
</tr>
<tr>
<td>XML</td>
<td>SQL TYPE IS XML AS CLOB(n)</td>
<td>1≤n≤2147483647</td>
</tr>
<tr>
<td>XML</td>
<td>SQL TYPE IS XML AS DBCLOB(n)</td>
<td>n is the number of double-byte characters. 1≤n≤1073741823 &quot;6&quot; on page 724</td>
</tr>
<tr>
<td>BLOB file reference</td>
<td>SQL TYPE IS BLOB_FILE</td>
<td>Use this data type only to manipulate data in BLOB columns. &quot;3&quot; on page 724, &quot;6&quot; on page 724, &quot;7&quot; on page 724</td>
</tr>
<tr>
<td>CLOB file reference</td>
<td>SQL TYPE IS CLOB_FILE</td>
<td>Use this data type only to manipulate data in CLOB columns. &quot;3&quot; on page 724, &quot;6&quot; on page 724, &quot;7&quot; on page 724</td>
</tr>
<tr>
<td>DBCLOB file reference</td>
<td>SQL TYPE IS DBCLOB_FILE</td>
<td>Use this data type only to manipulate data in DBCLOB columns. &quot;3&quot; on page 724, &quot;6&quot; on page 724, &quot;7&quot; on page 724</td>
</tr>
<tr>
<td>XML BLOB file reference</td>
<td>SQL TYPE IS XML AS BLOB_FILE</td>
<td>Use this data type only to manipulate XML data as BLOB files. &quot;3&quot; on page 724</td>
</tr>
<tr>
<td>XML CLOB file reference</td>
<td>SQL TYPE IS XML AS CLOB_FILE</td>
<td>Use this data type only to manipulate XML data as CLOB files. &quot;3&quot; on page 724</td>
</tr>
<tr>
<td>XML DBCLOB file reference</td>
<td>SQL TYPE IS XML AS DBCLOB_FILE</td>
<td>Use this data type only to manipulate XML data as DBCLOB files. &quot;3&quot; on page 724</td>
</tr>
<tr>
<td>ROWID</td>
<td>SQL TYPE IS ROWID</td>
<td></td>
</tr>
</tbody>
</table>

Table notes:
The following notes apply as indicated to Table 117 on page 720 and Table 118 on page 722.

1. If a host variable includes an indicator variable, the SQLTYPE value is the base SQLTYPE value plus 1.
2. If p=0, Db2 interprets it as DECIMAL(31). For example, Db2 interprets a PL/I data type of DEC FIXED(0,0) to be DECIMAL(31,0), which equates to the SQL data type of DECIMAL(31,0).
3. Do not use this data type as a column type.
4. n is the number of double-byte characters.
5. CCSID 1200 is always assigned to WIDECHAR type host var.
6. The data type conversions can be used only if the Db2 coprocessor is used, and the PL/I compiler options FLOAT(DFP) and ARCH(7) are specified.
7. Specify the following compiler options when you compile your program: LIMITS(FIXEDBIN(63), FIXEDDEC(31)).

Related concepts
Compatibility of SQL and language data types
The host variable data types that are used in SQL statements must be compatible with the data types of the columns with which you intend to use them.

LOB host variable, LOB locator, and LOB file reference variable declarations
When you write applications to manipulate LOB data, you need to declare host variables to hold the LOB data or LOB locator. Alternatively, you need to declare LOB file reference variables to point to the LOB data.

Host variable data types for XML data in embedded SQL applications (Db2 Programming for XML)

REXX applications that issue SQL statements
You can code SQL statements in a REXX programs wherever you can use REXX commands.

Db2 REXX Language Support supports all dynamic SQL statements and the following static SQL statements:

• CALL
• CLOSE
• CONNECT
• DECLARE CURSOR
• DESCRIBE prepared statement or table
• DESCRIBE CURSOR
• DESCRIBE INPUT
• DESCRIBE PROCEDURE
• EXECUTE
• EXECUTE IMMEDIATE
• FETCH
• OPEN
• PREPARE
• RELEASE connection
• SET CONNECTION
• SET CURRENT PACKAGE PATH
• SET CURRENT PACKAGESET
• SET host-variable = CURRENT DATE
• SET host-variable = CURRENT DEGREE
• SET host-variable = CURRENT MEMBER
• SET host-variable = CURRENT PACKAGESET
• SET host-variable = CURRENT PATH
• SET host-variable = CURRENT SERVER
• SET host-variable = CURRENT SQLID
• SET host-variable = CURRENT TIME
• SET host-variable = CURRENT TIMESTAMP
• SET host-variable = CURRENT TIMEZONE

Each SQL statement in a REXX program must begin with EXEC SQL, in either upper-, lower-, or mixed-case. One of the following items must follow EXEC SQL:

• An SQL statement enclosed in single or double quotation marks.
• A REXX variable that contains an SQL statement. The REXX variable must not be preceded by a colon.
For example, you can use either of the following methods to execute the COMMIT statement in a REXX program:

```plaintext
EXECSQL "COMMIT"
```

```plaintext
rexxvar="COMMIT"
EXECSQL rexxvar
```

The following dynamic statements must be executed using EXECUTE IMMEDIATE or PREPARE and EXECUTE under DSNREXX:

- DECLARE GLOBAL TEMPORARY TABLE
- SET CURRENT DEBUG MODE
- SET CURRENT DECFLOAT Rounding MODE
- SET CURRENT MAINTAINED TABLE TYPES FOR OPTIMIZATION
- SET CURRENT QUERY ACCELERATION
- SET CURRENT REFRESH AGE
- SET CURRENT ROUTINE VERSION
- SET SCHEMA

You cannot execute a SELECT, INSERT, UPDATE, MERGE, or DELETE statement that contains host variables. Instead, you must execute PREPARE on the statement, with parameter markers substituted for the host variables, and then use the host variables in an EXECUTE, OPEN, or FETCH statement. See “Host variables” on page 477 for more information.

An SQL statement follows rules that apply to REXX commands. The SQL statement can optionally end with a semicolon and can be enclosed in single or double quotation marks, as in the following example:

```plaintext
'EXECSQL COMMIT';
```

**Comments**

You cannot include REXX comments (/* ... */) or SQL comments (--) within SQL statements. However, you can include REXX comments anywhere else in the program.

**Delimiters for SQL statements**

Delimit SQL statements in REXX program by preceding the statement with EXECSQL. If the statement is in a literal string, enclose it in single or double quotation marks.

**Continuation for SQL statements**

SQL statements that span lines follow REXX rules for statement continuation. You can break the statement into several strings, each of which fits on a line, and separate the strings with commas or with concatenation operators followed by commas. For example, either of the following statements is valid:

```plaintext
EXECSQL,
 "UPDATE DSN8C10.DEPT",
 "SET MGRNO = '000010';",
 "WHERE DEPTNO = 'D11';"
```

```plaintext
"EXECSQL "||",
 "UPDATE DSN8C10.DEPT "||",
 "SET MGRNO = '000010'"||",
 "WHERE DEPTNO = 'D11'
```

**Including code**

The EXECSQL INCLUDE statement is not valid for REXX. You therefore cannot include externally defined SQL statements in a program.

**Margins**

Like REXX commands, SQL statements can begin and end anywhere on a line.
You can use any valid REXX name that does not end with a period as a host variable. However, host variable names should not begin with ‘SQL’, ‘RDI’, ‘DSN’, ‘RXML’, or ‘QRW’. Variable names can be at most 64 bytes.

### Nulls

A REXX null value and an SQL null value are different. The REXX language has a null string (a string of length 0) and a null clause (a clause that contains only blanks and comments). The SQL null value is a special value that is distinct from all nonnull values and denotes the absence of a value. Assigning a REXX null value to a Db2 column does not make the column value null.

### Statement labels

You can precede an SQL statement with a label, in the same way that you label REXX commands.

### Handling SQL error codes

REXX applications can request more information about SQL errors from Db2. For more information, see “Handling SQL error codes in REXX applications” on page 750.

### Related tasks

- Overview of programming applications that access Db2 for z/OS data
- Applications that interact with Db2 must first connect to Db2. They can then read, add, or modify data or manipulate Db2 objects.
- Including dynamic SQL in your program
- Dynamic SQL is prepared and executed while the program is running.
- Handling SQL error codes
- Application programs can request more information about SQL error codes from Db2.
- Setting limits for system resource usage by using the resource limit facility (Db2 Performance)

### REXX programming examples

You can write Db2 programs in REXX. These programs can access a local or remote Db2 subsystem and can execute static or dynamic SQL statements. This information contains several such programming examples.

To prepare and run these applications, use the JCL in `prefix.SDSNSAMP` as a model for your JCL.

### Related reference

- Assembler, C, C++, COBOL, PL/I, and REXX programming examples (Db2 Programming samples)
- Db2 for z/OS Exchange

### Sample Db2 REXX application

You can use a REXX application to accept a table name as input and produce a SELECT, INSERT, or UPDATE SQL statement or a LOAD utility statement for the specified table as output.

The following example shows a complete Db2 REXX application named DRAW. DRAW must be invoked from the command line of an ISPF edit session. DRAW takes a table or view name as input and produces a SELECT, INSERT, or UPDATE SQL statement or a LOAD utility control statement that includes the columns of the table as output.

#### DRAW syntax:

```plaintext
%DRAW object-name (SSID=ssid TYPE=[SELECT|INSERT|UPDATE|LOAD])
```

#### DRAW parameters:

- `SSID`: The Db2 subsystem identifier.
- `object-name`: The name of the table or view.
- `TYPE`: The type of statement to generate (SELECT, INSERT, UPDATE, LOAD).

---

**Chapter 4. Embedded SQL programming** 727
**object-name**
The name of the table or view for which DRAW builds an SQL statement or utility control statement. The name can be a one-, two-, or three-part name. The table or view to which object-name refers must exist before DRAW can run.

*object-name* is a required parameter.

**SSID=ssid**
Specifies the name of the local Db2 subsystem.

S can be used as an abbreviation for SSID.

If you invoke DRAW from the command line of the edit session in SPUFI, SSID=ssid is an optional parameter. DRAW uses the subsystem ID from the DB2I Defaults panel.

**TYPE=operation-type**
The type of statement that DRAW builds.

T can be used as an abbreviation for TYPE.

*operation-type* has one of the following values:

**SELECT**
Builds a SELECT statement in which the result table contains all columns of *object-name*.

S can be used as an abbreviation for SELECT.

**INSERT**
Builds a template for an INSERT statement that inserts values into all columns of *object-name*. The template contains comments that indicate where the user can place column values.

I can be used as an abbreviation for INSERT.

**UPDATE**
Builds a template for an UPDATE statement that updates columns of *object-name*. The template contains comments that indicate where the user can place column values and qualify the update operation for selected rows.

U can be used as an abbreviation for UPDATE.

**LOAD**
Builds a template for a LOAD utility control statement for *object-name*.

L can be used as an abbreviation for LOAD.

**TYPE=operation-type** is an optional parameter. The default is TYPE=SELECT.

**DRAW data sets:**

**Edit data set**
The data set from which you issue the DRAW command when you are in an ISPF edit session. If you issue the DRAW command from a SPUFI session, this data set is the data set that you specify in field 1 of the main SPUFI panel (DSNESPO1). The output from the DRAW command goes into this data set.

**DRAW return codes:**

**Return code**

**Meaning**

0
Successful completion.

12
An error occurred when DRAW edited the input file.

20
One of the following errors occurred:
- No input parameters were specified.
- One of the input parameters was not valid.
• An SQL error occurred when the output statement was generated.

**Examples of DRAW invocation:**

Generate a SELECT statement for table DSN8C10.EMP at the local subsystem. Use the default DB2I subsystem ID.

The DRAW invocation is:

```
DRAW DSN8C10.EMP (TYPE=SELECT
```

The output is:

```
SELECT "EMPNO" , "FIRSTNAME" , "MIDINIT" , "LASTNAME" , "WORKDEPT" , "PHONENO" , "HIREDATE" , "JOB" , "EDLEVEL" , "SEX" , "BIRTHDATE" , "SALARY" , "BONUS" , "COMM"
FROM DSN8C10.EMP
```

Generate a template for an INSERT statement that inserts values into table DSN8C10.EMP at location SAN_JOSE. The local subsystem ID is DSN.

The DRAW invocation is:

```
DRAW SAN_JOSE.DSN8C10.EMP (TYPE=INSERT SSID=DSN
```

The output is:

```
INSERT INTO SAN_JOSE.DSN8C10.EMP ( "EMPNO" , "FIRSTNAME" , "MIDINIT" , "LASTNAME" , "WORKDEPT" , "PHONENO" , "HIREDATE" , "JOB" , "EDLEVEL" , "SEX" , "BIRTHDATE" , "SALARY" , "BONUS" , "COMM" )
VALUES (
    -- ENTER VALUES BELOW        COLUMN NAME        DATA TYPE
    , "EMPNO"                  CHAR(6) NOT NULL
    , "FIRSTNAME"              VARCHAR(12) NOT NULL
    , "MIDINIT"                CHAR(1) NOT NULL
    , "LASTNAME"               VARCHAR(15) NOT NULL
    , "WORKDEPT"               CHAR(3)
    , "PHONENO"                CHAR(4)
    , "HIREDATE"               DATE
    , "JOB"                    CHAR(8)
    , "EDLEVEL"                SMALLINT
    , "SEX"                    CHAR(1)
    , "BIRTHDATE"              DATE
    , "SALARY"                 DECIMAL(9,2)
    , "BONUS"                  DECIMAL(9,2)
)
```

Generate a template for an UPDATE statement that updates values of table DSN8C10.EMP. The local subsystem ID is DSN.

The DRAW invocation is:

```
DRAW DSN8C10.EMP (TYPE=UPDATE SSID=DSN
```

The output is:

```
UPDATE DSN8C10.EMP
SET
    -- ENTER VALUES BELOW        COLUMN NAME        DATA TYPE
    , "EMPNO"                  CHAR(6) NOT NULL
    , "FIRSTNAME"              VARCHAR(12) NOT NULL
    , "MIDINIT"                CHAR(1) NOT NULL
    , "LASTNAME"               VARCHAR(15) NOT NULL
    , "WORKDEPT"               CHAR(3)
    , "PHONENO"                CHAR(4)
    , "HIREDATE"               DATE
    , "JOB"                    CHAR(8)
    , "EDLEVEL"                SMALLINT
    , "SEX"                    CHAR(1)
    , "BIRTHDATE"              DATE
    , "SALARY"                 DECIMAL(9,2)
    , "BONUS"                  DECIMAL(9,2)
```
Generate a LOAD control statement to load values into table DSN8C10.EMP. The local subsystem ID is DSN.

The draw invocation is:

```
DRAW DSN8C10.EMP (TYPE=LOAD SSID=DSN)
```

The output is:

```
LOAD DATA INDDN SYSREC INTO TABLE DSN8C10.EMP

<table>
<thead>
<tr>
<th>Column</th>
<th>Position</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>EMPNO</td>
<td>1</td>
<td>CHAR(6)</td>
</tr>
<tr>
<td>FIRSTNAME</td>
<td>8</td>
<td>VARCHAR</td>
</tr>
<tr>
<td>MIDINIT</td>
<td>21</td>
<td>CHAR(1)</td>
</tr>
<tr>
<td>LASTNAME</td>
<td>23</td>
<td>VARCHAR</td>
</tr>
<tr>
<td>WORKDEPT</td>
<td>39</td>
<td>CHAR(3)</td>
</tr>
<tr>
<td>PHONENO</td>
<td>43</td>
<td>CHAR(4)</td>
</tr>
<tr>
<td>HIREDATE</td>
<td>48</td>
<td>DATE EXTERNAL</td>
</tr>
<tr>
<td>JOB</td>
<td>59</td>
<td>CHAR(8)</td>
</tr>
<tr>
<td>EDLEVEL</td>
<td>68</td>
<td>SMALLINT</td>
</tr>
<tr>
<td>SEX</td>
<td>71</td>
<td>CHAR(1)</td>
</tr>
<tr>
<td>BIRTHDATE</td>
<td>73</td>
<td>DATE EXTERNAL</td>
</tr>
<tr>
<td>SALARY</td>
<td>84</td>
<td>DECIMAL EXTERNAL(9,2)</td>
</tr>
<tr>
<td>BONUS</td>
<td>90</td>
<td>DECIMAL EXTERNAL(9,2)</td>
</tr>
<tr>
<td>COMM</td>
<td>96</td>
<td>DECIMAL EXTERNAL(9,2)</td>
</tr>
</tbody>
</table>

DRAW source code:

```
/* REXX *******************************************************************/
L1 = WHEREAMI()
/*
DRAW creates basic SQL queries by retrieving the description of a
   table. You must specify the name of the table or view to be queried.
   You can specify the type of query you want to compose. You might need
to specify the name of the DB2 subsystem.
   >>>--DRAW-----tablename-----|---------------------------|------->_<
       |-(--Ssid=subsystem-name-|-|
       |       |Select-+    |
       |Type=-|Insert-|----|
       |Update-|
       |Load--+
Ssid=subsystem-name
   subsystem-name specified the name of the DB2 subsystem.
Select
   Composes a basic query for selecting data from the columns of a
table or view. If TYPE is not specified, SELECT is assumed.
Using SELECT with the DRAW command produces a query that would
retrieve all rows and all columns from the specified table. You
can then modify the query as needed.
   A SELECT query of EMP composed by DRAW looks like this:
   SELECT "EMPNO", "FIRSTNAME", "MIDINIT", "LASTNAME", "WORKDEPT",
   "PHONENO", "HIREDATE", "JOB", "EDLEVEL", "SEX", "BIRTHDATE",
   "SALARY", "BONUS", "COMM"
   FROM DSN8C10.EMP
   If you include a location qualifier, the query looks like this:
   SELECT "EMPNO", "FIRSTNAME", "MIDINIT", "LASTNAME", "WORKDEPT",
   "PHONENO", "HIREDATE", "JOB", "EDLEVEL", "SEX", "BIRTHDATE",
   "SALARY", "BONUS", "COMM"
   FROM STLEC1.DSN8C10.EMP
   To use this SELECT query, type the other clauses you need. If
   you are selecting from more than one table, use a DRAW command
   for each table name you want represented.
```
Insert
Composes a basic query to insert data into the columns of a table or view.
The following example shows an INSERT query of EMP that DRAW composed:

```
INSERT INTO DSN8C10.EMP  ( "EMPNO" , "FIRSTNME" , "MIDINIT" , "LASTNAME" , "WORKDEPT" , "PHONENO" , "HIREDATE" , "JOB" , "EDLEVEL" , "SEX" , "BIRTHDATE" , "SALARY" , "BONUS" , "COMM" )
VALUES (", -- ENTER VALUES BELOW " COLUMN NAME " DATA TYPE
  , -- EMPNO  CHAR(6) NOT NULL
  , -- FIRSTNME  VARCHAR(12) NOT NULL
  , -- MIDINIT  CHAR(1) NOT NULL
  , -- LASTNAME  VARCHAR(15) NOT NULL
  , -- WORKDEPT  CHAR(3)
  , -- PHONENO  CHAR(4)
  , -- HIREDATE  DATE
  , -- JOB  CHAR(8)
  , -- EDLEVEL  SMALLINT
  , -- SEX  CHAR(1)
  , -- BIRTHDATE  DATE
  , -- SALARY  DECIMAL(9,2)
  , -- BONUS  DECIMAL(9,2)
  , -- COMM  DECIMAL(9,2)
)
```

To insert values into EMP, type values to the left of the column names.

Update
Composes a basic query to change the data in a table or view.
The following example shows an UPDATE query of EMP composed by DRAW:

```
UPDATE DSN8C10.EMP  SET
  -- COLUMN NAME " ENTER VALUES BELOW 
  , "EMPNO"= -- CHAR(6) NOT NULL
  , "FIRSTNME"= -- VARCHAR(12) NOT NULL
  , "MIDINIT"= -- CHAR(1) NOT NULL
  , "LASTNAME"= -- VARCHAR(15) NOT NULL
  , "WORKDEPT"= -- CHAR(3)
  , "PHONENO"= -- CHAR(4)
  , "HIREDATE"= -- DATE
  , "JOB"= -- CHAR(8)
  , "EDLEVEL"= -- SMALLINT
  , "SEX"= -- CHAR(1)
  , "BIRTHDATE"= -- DATE
  , "SALARY"= -- DECIMAL(9,2)
  , "BONUS"= -- DECIMAL(9,2)
  , "COMM"= -- DECIMAL(9,2)
WHERE
```

To use this UPDATE query, type the changes you want to make to the right of the column names, and delete the lines you do not need. Be sure to complete the WHERE clause.

Load
Composes a load statement to load the data in a table.
The following example shows a LOAD statement of EMP composed by DRAW:

```
LOAD DATA INDDN SYSREC INTO TABLE DSN8C10.EMP
  ( "EMPNO" POSITION( 1) CHAR(6)
  , "FIRSTNME" POSITION( 8) VARCHAR
  , "MIDINIT" POSITION(21) CHAR(1)
  , "LASTNAME" POSITION(23) VARCHAR
  , "WORKDEPT" POSITION(39) CHAR(3)
  , "PHONENO" POSITION(43) CHAR(4)
  , "HIREDATE" POSITION(48) DATE EXTERNAL
  , "JOB" POSITION(59) CHAR(8)
  , "EDLEVEL" POSITION(68) SMALLINT
  , "SEX" POSITION(71) CHAR(1)
  , "BIRTHDATE" POSITION(73) DATE EXTERNAL
  , "SALARY" POSITION(84) DECIMAL EXTERNAL(9,2)
  , "BONUS" POSITION(90) DECIMAL EXTERNAL(9,2)
  , "COMM" POSITION(96) DECIMAL EXTERNAL(9,2)
)
```

Chapter 4. Embedded SQL programming 731
To use this LOAD statement, type the changes you want to make, and delete the lines you do not need.

L2 = WHEREAMI()

ADDRESS ISPEXC
"ISREDIT MACRO (ARGS) NOPROCESS"
IF ARGS = "" Then
Do
Do I = L1+2 To L2-2;Say SourceLine(I);End
Exit (20)
End

PARSE Upper Var Args Table "(" Parms
Parms = Translate(Parms," ",")")"
Type = "SELECT" /* Default */
SSID = "" /* Default */
"VGET (DSNEOV01)"
IF RC = 0 Then SSID = DSNEOV01
IF (Parms <> "$") Then
Do Until(Parms = "")
Parse Var Parms Var "=" Value Parms
IF Var = "T" | Var = "TYPE" Then Type = Value
ELSE
IF Var = "S" | Var = "SSID" Then SSID = Value
ELSE
Exit (20)
End

"CONTROL ERRORS RETURN"
"ISREDIT (LEFTBND,RIGHTBND) = BOUNDS"
"ISREDIT (LRECL) = DATA_WIDTH" /*LRECL*/
BndSize = RightBnd - LeftBnd + 1
IF BndSize > 72 Then BndSize = 72
"ISREDIT PROCESS DEST"
Select
WHEN rc = 0 Then
'ISREDIT (ZDEST) = LINENUM .ZDEST'
WHEN rc <= 8 Then /* No A or B entered */
Do
zedsmsg = 'Enter "A"/"B" line cmd'
zedlmsg = 'DRAW requires an "A" or "B" line command'
'SETMSG MSG(ISRZ001)'
Exit 12
End
WHEN rc < 20 Then /* Conflicting line commands - edit sets message */
Exit 12
WHEN rc = 20 Then
zdest = 0
Otherwise
Exit 12
End

SQLTYPE. = "UNKNOWN TYPE"
VCHTYPE = 448; SQLTYPES.VCHTYPE = 'VARCHAR'
CHTYPE = 452; SQLTYPES.CHTYPE = 'CHAR'
LVCHTYPE = 456; SQLTYPES.LVCHTYPE = 'VARCHAR'
VGRTYPE = 464; SQLTYPES.VGRTPY = 'VARGRAPHIC'
GRTYPE = 468; SQLTYPES.GRTYPE = 'GRAPHIC'
LVGRTYPE = 472; SQLTYPES.LVGRTYPE = 'VARGRAPHIC'
FLOTYPE = 480; SQLTYPES.FLOTYPE = 'FLOAT'
DCTYPE = 484; SQLTYPES.DCTYPE = 'DECIMAL'
INTYPE = 496; SQLTYPES.INTYPE = 'INTEGER'
SMTYPE = 500; SQLTYPES.SMTYPE = 'SMALLINT'
DATYPE = 384; SQLTYPES.DATYPE = 'DATE'
TITYPE = 388; SQLTYPES.TITYPE = 'TIME'
TSTYPE = 392; SQLTYPES.TSTYPE = 'TIMESTAMP'
Address TSO "SUBCOM DSNREXX" /* HOST CMD ENV AVAILABLE? */
IF RC THEN
Address DSNREXX "CONNECT" SSID
"ISREDIT MACRO (ARGS) NOPROCESS"
IF SQLCODE ^= 0 Then Call SQLCA
Address DSNREXX "EXECSQL DESCRIBE TABLE :TABLE INTO :SQLDA"
IF SQLCODE ^= 0 Then Call SQLCA
Address DSNREXX "EXECSQL COMMIT"
Address DSNREXX "DISCONNECT"
IF SQLCODE ^= 0 Then Call SQLCA
Select
When (Left(Type,1) = "S") Then
  Call DrawSelect
When (Left(Type,1) = "I") Then
  Call DrawInsert
When (Left(Type,1) = "U") Then
  Call DrawUpdate
When (Left(Type,1) = "L") Then
  Call DrawLoad
Otherwise EXIT (20)
End
Do I = LINE.0 To 1 By -1
  LINE = COPIES(“ “, LEFTBND-1)||LINE.I
  ’ISREDIT LINE_AFTER ‘zdest’ = DATALINE (Line)’
End
line1 = zdest + 1
’ISREDIT CURSOR = ‘line1 0
Exit

/*******************************************************************************/
WHEREAMI:; RETURN SIGL
/*******************************************************************************/
/* Draw SELECT */
/*******************************************************************************/
DrawSelect:
  Line.0 = 0
  Line = "SELECT"
Do I = 1 To SQLDA.SQLD
  If I > 1 Then Line = Line ‘,’
  ColName = ‘"SQLDA.I.SQLNAME”’
  Null = SQLDA.I.SQLTYPE//2
  If Length(Line)+Length(ColName)+LENGTH(" ,") > BndSize THEN
    Do
      L = Line.0 + 1; Line.0 = L
      Line.L = Line
      Line = “ “
    End
    Line = Line ColName
  End I
If Line ^= “” Then
  Do
    L = Line.0 + 1; Line.0 = L
    Line.L = Line
    Line = “ “
  End
  L = Line.0 + 1; Line.0 = L
  Line.L = “FROM” TABLE
  Return
/*******************************************************************************/
/* Draw INSERT */
/*******************************************************************************/
DrawInsert:
  Line.0 = 0
  Line = "INSERT INTO” TABLE “("
Do I = 1 To SQLDA.SQLD
  If I > 1 Then Line = Line ‘,’
  ColName = ‘"SQLDA.I.SQLNAME”’
  If Length(Line)+Length(ColName) > BndSize THEN
    Do
      L = Line.0 + 1; Line.0 = L
      Line.L = Line
      Line = “ “
    End
    Line = Line ColName
  End I
If I = SQLDA.SQLD Then Line = Line ‘)’
End I
If Line ^= “” Then
  Do
    L = Line.0 + 1; Line.0 = L
    Line.L = Line
    Line = “ “
  End
L = Line.0 + 1; Line.0 = L
Line.L = “ VALUES ("
L = Line.0 + 1; Line.0 = L
Line.L = “-
".- ENTER VALUES BELOW      COLUMN NAME      DATA TYPE"
Do I = 1 To SQLDA.SQLD
  If SQLDA.SQLD > 1 & I < SQLDA.SQLD Then
Line = "                         , --"
Else
    Line = "                         ) --"
Line = Line Left(SQLDA.I.SQLNAME,18)
Type  = SQLDA.I.SQLTYPE
Null  = Type//2
If Null Then Type = Type - 1
Len   = SQLDA.I.SQLLEN
Prcsn = SQLDA.I.SQLLEN.SQLPRECISION
Scale = SQLDA.I.SQLLEN.SQLSCALE
Select
    When (Type = CHTYPE   ,
          |Type = VCHTYPE  ,
          |Type = LVCHTYPE ,
          |Type = GRTYP    ,
          |Type = VGRTYP   ,
          |Type = LVGRTYP  ) THEN
        Type = SQLTYPES.Type("STRIP(LEN)")"
When (Type = FLOTYPE  ) THEN
        Type = SQLTYPES.Type("STRIP((LEN\*4)-11) ")"
When (Type = DCTYPE   ) THEN
        Type = SQLTYPES.Type("STRIP(PRCSN)","STRIP(SCALE)")"
Otherwise
        Type = SQLTYPES.Type
End
Line = Line Type
If Null = 0 Then
    Line = Line "NOT NULL"
L = Line.0 + 1; Line.0 = L
Line.L = Line
End I
Return

/**********************************************************************/
/* Draw UPDATE */
/**********************************************************************/
DrawUpdate:
    Line.0 = 1
    Line.1 = "UPDATE" TABLE "SET"
L = Line.0 + 1; Line.0 = L
Line.L =",
    "-- COLUMN NAME           ENTER VALUES BELOW      DATA TYPE"
Do I = 1 To SQLDA.SQLD
    If I = 1 Then
        Line = "  ">
    Else
        Line = " ,
    Line = Line Left('"SQLDA.I.SQLNAME"=',21)
    Line = Line Left(" ",20)
    Type  = SQLDA.I.SQLTYPE
    Null  = Type//2
If Null Then Type = Type - 1
Len   = SQLDA.I.SQLLEN
Prcsn = SQLDA.I.SQLLEN.SQLPRECISION
Scale = SQLDA.I.SQLLEN.SQLSCALE
Select
    When (Type = CHTYPE   ,
          |Type = VCHTYPE  ,
          |Type = LVCHTYPE ,
          |Type = GRTYP    ,
          |Type = VGRTYP   ,
          |Type = LVGRTYP  ) THEN
        Type = SQLTYPES.Type("STRIP(LEN)"")"
When (Type = FLOTYPE  ) THEN
        Type = SQLTYPES.Type("STRIP((LEN\*4)-11) ")"
When (Type = DCTYPE   ) THEN
        Type = SQLTYPES.Type("STRIP(PRCSN)","STRIP(SCALE)")"
Otherwise
        Type = SQLTYPES.Type
End
Line = Line "--" Type
If Null = 0 Then
    Line = Line "NOT NULL"
L = Line.0 + 1; Line.0 = L
Line.L = Line
End I
L = Line.0 + 1; Line.0 = L
Line.L = "WHERE"
Return
DrawLoad:
Line.0 = 1
Line.1 = "LOAD DATA INDDN SYSREC INTO TABLE" TABLE
Position = 1
Do I = 1 To SQLDA.SQLD
  If I = 1 Then
    Line = "("
  Else
    Line = " ,"
  End
  Line = Line Left('"SQLDA.I.SQLNAME"',20)
  Type = SQLDA.I.SQLTYPE
  Null = Type//2
  If Null Then Type = Type - 1
  Len = SQLDA.I.SQLLEN
  Prcsn = SQLDA.I.SQLLEN.SQLPRECISION
  Scale = SQLDA.I.SQLLEN.SQLSCALE
  Select
    When (Type = CHTYPE ,
      |Type = GRTYP ) THEN
      Type = SQLTYPES.Type("STRIP(LEN)")
    When (Type = FLOTYPE ) THEN
      Type = SQLTYPES.Type("STRIP((LEN*4)-11)"")
    When (Type = DCTYPE ) THEN
      Do
        Type = SQLTYPES.Type "EXTERNAL"
        Type = Type("STRIP(PRCSN)","STRIP(SCALE)"")
        Len = (PRCSN+2)%2
      End
    End
    When (Type = DATYPE ,
      |Type = TITYPE ,
      |Type = TSTYPE ) THEN
      Type = SQLTYPES.Type "EXTERNAL"
    Otherwise
      Type = SQLTYPES.Type
    End
    If   (Type = GRTYP ,
      |Type = VGRTYP ,
      |Type = LVGRTYP ) THEN
      Len = Len * 2
    If   (Type = VCHTYPE ,
      |Type = LVCHTYPE ,
      |Type = VGRTYP ,
      |Type = LVGRTYP ) THEN
      Len = Len + 2
  End
  Line = Line Type
  L = Line.0 + 1; Line.0 = L
  Line.L = Line
  If Null = 1 Then
    Do
      Line = " 
      Line = Line Left('',20)
      Line = Line " NULLIF("RIGHT(POSITION,5)")='?'"
      L = Line.0 + 1; Line.0 = L
      Line.L = Line
    End
  End I
  Position = Position + Len + 1
End I
L = Line.0 + 1; Line.0 = L
Line.L = ")"
Return
/* Draw LOAD */

Display SQLCA:
"ISREDIT LINE_AFTER "zdest" = MSGLINE 'SQLSTATE="SQLSTATE"'" 
"ISREDIT LINE_AFTER "zdest" = MSGLINE 'SQLWARN ="SQLWARN.0"," 
  | SQLWARN.1"," 
  | SQLWARN.2"," 
  | SQLWARN.3"," 
  | SQLWARN.4"," 
  | SQLWARN.5"," 
  | SQLWARN.6"," 
  | SQLWARN.7"," 
  | SQLWARN.8"," 
  | SQLWARN.9"," 

Chapter 4. Embedded SQL programming 735
Example of how an indicator variable is used in a REXX program

The way that you use indicator variables for input host variables in REXX programs is slightly different than the way that you use indicator variables in other languages. When you want to pass a null value to a Db2 column, in addition to putting a negative value in an indicator variable, you also need to put a valid value in the corresponding host variable.

For example, the following statements set a value in the WORKDEPT column in table EMP to null:

```sql
SQLSTMT="UPDATE EMP",
 "SET WORKDEPT = ?",
 HVWORKDEPT='000'
INDWORKDEPT=-1
"EXECSQL PREPARE S100 FROM :SQLSTMT"
"EXECSQL EXECUTE S100 USING :HVWORKDEPT :INDWORKDEPT"
```

In the following program, the phone number for employee Haas is selected into variable HVPhone. After the SELECT statement executes, if no phone number for employee Haas is found, indicator variable INDPhone contains -1.

```rexx
'SUBCOM DSNREXX'
IF RC THEN ,
 S_RC = RXSUBCOM("ADD","DSNREXX","DSNREXX")
ADDRESS DSNREXX
 'CONNECT' 'DSN'
SQLSTMT = ,
 "SELECT PHONENO FROM DSN8C10.EMP WHERE LASTNAME='HAAS'"
"EXECSQL DECLARE C1 CURSOR FOR S1"
"EXECSQL PREPARE S1 FROM :SQLSTMT"
Say "SQLCODE from PREPARE is "SQLCODE
"EXECSQL OPEN C1"
Say "SQLCODE from OPEN is "SQLCODE
"EXECSQL FETCH C1 INTO :HVPhone :INDPhone"
Say "SQLCODE from FETCH is "SQLCODE
If INDPhone < 0 Then ,
Say 'Phone number for Haas is null.'
"EXECSQL CLOSE C1"
Say "SQLCODE from CLOSE is "SQLCODE
S_RC = RXSUBCOM("DELETE","DSNREXX","DSNREXX")
```

Example REXX programs for LOB data

Db2 programs in REXX can use LOB host variables and file reference variables.

**Example of using simple LOB host variables in a REXX program**

```rexx
/* REXX exec to use a LOB in a host var */
ssid = "VA1A";
Address TSO "SUBCOM DSNREXX";
if rc then s_rc = RXSUBCOM("ADD","DSNREXX","DSNREXX")
  say "rc from rxsubcom add=" rc
Address DSNREXX;
 "CONNECT" ssid;
if sqlcode \= 0 then do;
  say "CONNECT to" ssid "failed."
  call sqlca
```
exit 8;
end;

stmt = "DROP TABLE REXXCLOB";
Address DSNREXX, "EXECSQL EXECUTE IMMEDIATE :STMT";
say "RC/SQLCODE after DROP is" rc"/sqlcode;
if rc <> 0 then call sqlca;
stmt = "CREATE TABLE REXXCLOB (" ||
"C1 CLOB(2M))"
Address DSNREXX, "EXECSQL EXECUTE IMMEDIATE :STMT";
say "RC/SQLCODE after CREATE is" rc"/sqlcode;
if rc <> 0 then call sqlca;
/* Insert into the CLOB table */
data = "THIS IS A SHORT CLOB, BUT IT IS A CLOB"
stmt = "INSERT INTO REXXCLOB (C1) VALUES(?)";
Address DSNREXX "EXECSQL PREPARE S1 FROM :STMT"
say "RC/SQLCODE after PREPARE is" rc"/sqlcode;
if rc <> 0 then call sqlca;
mydata = copies('Z',75000);
say "length of :mydata="length(mydata);
Address DSNREXX "EXECSQL EXECUTE S1 USING :MYDATA"
say "RC/SQLCODE after EXECUTE is" rc"/sqlcode;
if rc <> 0 then call sqlca;
/*
var1 = copies(' ',2048000);
say "length of :var1="length(var1);
*/
stmt = "SELECT C1, LENGTH(C1) FROM REXXCLOB"
Address DSNREXX "EXECSQL PREPARE S1 FROM :STMT"
say "RC/SQLCODE after PREPARE (SELECT C1) is" rc"/sqlcode;
if rc <> 0 then call sqlca;
Address DSNREXX "EXECSQL DECLARE C1 CURSOR FOR S1"
Address DSNREXX "EXECSQL DECLARE C1 CURSOR FOR S1"
say "RC/SQLCODE after DECLARE is" rc"/sqlcode;
if rc <> 0 then call sqlca;
Address DSNREXX "EXECSQL OPEN C1"
say "RC/SQLCODE after OPEN is" rc"/sqlcode;
if rc <> 0 then call sqlca;
Address DSNREXX "EXECSQL FETCH C1 INTO :VAR1, :VAR2"
say "RC/SQLCODE after FETCH is" rc"/sqlcode;
if rc <> 0 then call sqlca;
say "length(var1)="length(var1);
say "var2=" var2;
Address DSNREXX "EXECSQL CLOSE C1"
say "RC/SQLCODE after CLOSE is" rc"/sqlcode;
if rc <> 0 then call sqlca;
/**
* Disconnect from the DB2 system.
*disconnect
*/
s_rc = RXSUBCOM("DELETE","DSNREXX","DSNREXX")
exit 0;
sqlca_error:
call sqlca
exit 16
Example of using LOB data with an SQLDA in a REXX program

```rexx
/* REXX EXEC TO INSERT A LOB USING SQLDA */
Address TSO "SUBCOM DSNREXX" ;
if rc then s_rc = RXSUBCOM("ADD","DSNREXX","DSNREXX") ;
say "rc from rxsubcom add= " rc
ssid = "VA1A" ;
Address DSNREXX "CONNECT" ssid ;
if sqlcode \= 0 then do ;
  say "CONNECT to" ssid "failed.";
  call sqlca
  exit 8 ;
end ;
stmt = "DROP TABLE REXXCLOB" ;
Address DSNREXX "EXECSQL EXECUTE IMMEDIATE :STMT" ;
say "RC/SQLCODE after DROP is" rc"/"sqlcode ;
if rc <> 0 then call sqlca ;
stmt = "CREATE TABLE REXXCLOB (" || ,
  "C1 CLOB(1M))" ;
Address DSNREXX "EXECSQL EXECUTE IMMEDIATE :STMT" ;
say "RC/SQLCODE after CREATE is" rc"/"sqlcode ;
if rc <> 0 then call sqlca ;
/* Insert into the CLOB table */
stmt = "INSERT INTO REXXCLOB (C1) VALUES(?) " ;
Address DSNREXX "EXECSQL PREPARE S1 INTO :D1 FROM :STMT" ;
say "RC/SQLCODE after PREPARE is" rc"/"sqlcode ;
if rc <> 0 then call sqlca ;
mydata = copies('A',1048560) ;   /* ~1M */
d1.sqlld = 1 ;
d1.sqltype = 408 ;
d1.sqllongl= length(mydata) ;
d1.sqldata = mydata ;
say "length of mydata is" length(mydata) ;
Address DSNREXX "EXECSQL EXECUTE S1 USING DESCRIPTOR :D1" ;
say "RC/SQLCODE after EXECUTE S1, USING D1 is" rc"/"sqlcode ;
if rc <> 0 then call sqlca ;
```
stmt = "SELECT C1, LENGTH(C1) AS LENGTH FROM REXXCLOB";
Address DSNREXX "EXECSQL PREPARE S1 INTO :OUTDA FROM :STMT"

say "RC/SQLCODE after PREPARE (SELECT C1) is" rc"/"sqlcode;
if rc <> 0 then call sqlca;

say "After PREPARE INTO, SQLDA looks like:";
say " outda.sqld=>"outda.sqld"<";
do i = 1 to outda.sqld;
say " "
say " outda."i".sqlname=>"outda.i.sqlname"<";
say " outda."i".sqltype=>"outda.i.sqltype"<";
end ;
say "
Address DSNREXX "EXECSQL DECLARE C1 CURSOR FOR S1";

say "RC/SQLCODE after DECLARE is" rc"/"sqlcode;
if rc <> 0 then call sqlca;

Address DSNREXX "EXECSQL OPEN C1"

say "RC/SQLCODE after OPEN is" rc"/"sqlcode;
if rc <> 0 then call sqlca;

Do forever;
Address DSNREXX "EXECSQL FETCH C1 INTO DESCRIPTOR :OUTDA"

say "RC/SQLCODE after FETCH is" rc"/"sqlcode;
if rc <> 0 then call sqlca;

if sqlcode = 100 then leave ; /* do forever */
say "outda.sqld=>"outda.sqld"<";

do i = 1 to outda.sqld;
say i": sqlname =>"outda.i.sqlname"<";
say i": sqltype =>"outda.i.sqltype"<";
say i": sqllen =>"outda.i.sqllen"<";
say i": sqllongl=>"outda.i.sqllongl"<";
say i": length =>"length(outda.i.sqldata)"<";
if length(outda.i.sqldata) > 62 then
   say i": sqldata =>"substr(outda.i.sqldata,1,62)"<"...
else
   say i": sqldata =>"outda.i.sqldata"<";
say ' ';
end ; /* do forever */
Address DSNREXX "EXECSQL CLOSE C1"

say "RC/SQLCODE after CLOSE is" rc"/"sqlcode;
if rc <> 0 then call sqlca;

/**********************************************************************
|  Disconnect from the DB2 system.                                     |
|**********************************************************************/
Address DSNREXX "DISCONNECT"

say "RC after DISCONNECT is" rc

s_rc = RXSUBCOM("DELETE","DSNREXX","DSNREXX")
exit 0 ;

sqlca_error:
call sqlca
exit 16

/**********************************************************************
| Error handling routine for bad SQL codes - just report and end.      |
|**********************************************************************/
SQLCA:
*/
say "       SQLCODE  = >"SQLCODE"<"
say "       SQLSTATE = >"SQLSTATE"<"
say "       SQLError = >"SQLError"<"
say "       SQLERRMC = >"SQLERRMC"<"
say "       SQLERRP  = >"SQLERRP"<"
say "       SQLERRD.1= >"SQLERRD.1"<";
say "       SQLERRD.2= >"SQLERRD.2"<";
say "       SQLERRD.3= >"SQLERRD.3"<";
say "       SQLERRD.4= >"SQLERRD.4"<";
say "       SQLERRD.5= >"SQLERRD.5"<";
say "       SQLERRD.6= >"SQLERRD.6"<";
say "       SQLWARN.0= >"SQLWARN.0"<"
say "       SQLWARN.1= >"SQLWARN.1"<";
*/
Example of using LOB File Reference Variables in a REXX program

/* REXX EXEC TO USE A CLOB FILE REFERENCE VARIABLE */

ssid = "VA1A" ;
Address TSO "SUBCOM DSNREXX" ;
if rc then s_rc = RXSUBCOM("ADD","DSNREXX","DSNREXX") ;
say "rc from rxsubcom add=" rc

Address DSNREXX ;
"CONNECT" ssid ;
if sqlcode \= 0 then do ;
say "CONNECT to" ssid "failed.";
call sqlca_error
exit 8 ;
end ;
stmt = "DROP TABLE REXXFRV" ;
Address DSNREXX ,
"EXECSQL EXECUTE IMMEDIATE :STMT" ;
say "RC/SQLCODE after DROP is" rc"/"sqlcode ;
if rc <> 0 & sqlcode <> -204 then call sqlca_error ;
stmt = "CREATE TABLE REXXFRV (" || ,
"C1 CLOB(2M))" ;
Address DSNREXX ,
"EXECSQL EXECUTE IMMEDIATE :STMT" ;
say "RC/SQLCODE after CREATE is" rc"/"sqlcode ;
if rc <> 0 then call sqlca_error ;

/* Write the CLOB to the preallocated file */
lines = 1500; /* enough 80 byte lines to make 2,000,000 bytes */
data.1 = "THIS IS A SHORT CLOB, BUT IT IS A CLOB 01" ;
data.2 = "THIS IS A SHORT CLOB, BUT IT IS A CLOB 02" ;
data.3 = "THIS IS A SHORT CLOB, BUT IT IS A CLOB 03" ;
data.4 = "THIS IS A SHORT CLOB, BUT IT IS A CLOB 04" ;
data.5 = "THIS IS A SHORT CLOB, BUT IT IS A CLOB 05" ;
data.6 = "THIS IS A SHORT CLOB, BUT IT IS A CLOB 06" ;
data.7 = "THIS IS A SHORT CLOB, BUT IT IS A CLOB 07" ;
data.8 = "THIS IS A SHORT CLOB, BUT IT IS A CLOB 08" ;
data.9 = "THIS IS A SHORT CLOB, BUT IT IS A CLOB 09" ;
data.10= "THIS IS A SHORT CLOB, BUT IT IS A CLOB 10" ;
data.0 = 10 ;
say 'data. stem initialized' ;
Do i = 1 to data.0 ;
data.i = left(data.i,131) ;
end ;
say 'data. stem padded to 131' ;
Do i = 1 to lines ;
Address MVS "EXECIO" data.0 "DISKW FRVFILE (stem data." ;
if rc <> 0 then do ;
say 'rc from execio='rc ;
signal bad_write ;
end ;
end ;
/* Close the file */
Address MVS "EXECIO 0 DISKW FRVFILE (FINIS"

/* The file now has to be freed. Otherwise, a
 SQLCODE -452, reason 12 at location 210 will be
 issued. */
Address TSO "FREE FI(FRVFILE)"
if rc <> 0 then signal bad_free;
stmt = "INSERT INTO REXXFRV (C1) VALUES(?) ";
Address DSNREXX "EXECSQL PREPARE S1 FROM :STMT";
say "RC/SQLCODE after PREPARE is" rc"/sqlcode;
if rc <> 0 then call sqlca_error;

/* Build the special SQLDA used by REXX for working with
 LOBs. */
mysqlda.sqlind = 0;  /* not null */
/* Note for a file reference variable, there is
 no SQLDATA value. Just use SQLDATA as part of the stem
 for .name and .fileoption, which are required for FRVs. */
mysqlda.1.sqltype = 920  /* clob file ref var */
mysqlda.1.sqldata.name = "SYSADM.FRV";  /* file name */
/* There are 4 fileoptions that can be set, and you can
 specify the value via text or a number. Here are the
 allowable values:
 SQL_FILE_READ    or 2
 SQL_FILE_CREATE  or 8
 SQL_FILE_OVERWRITE or 16
 SQL_FILE_APPEND  or 32
 */
mysqlda.1.sqlllen = length(mysqlda.1.sqldata.name);

Address DSNREXX "EXECSQL EXECUTE S1 USING DESCRIPTOR :MYSQLDA";
say "RC/SQLCODE after EXECUTE is" rc"/sqlcode;
if rc <> 0 then call sqlca_error;
stmt = "SELECT C1, LENGTH(C1) FROM REXXFRV";
Address DSNREXX "EXECSQL PREPARE S1 FROM :STMT";
say "RC/SQLCODE after PREPARE (SELECT C1) is" rc"/sqlcode;
if rc <> 0 then call sqlca_error;
Address DSNREXX "EXECSQL DECLARE C1 CURSOR FOR S1";
say "RC/SQLCODE after DECLARE is" rc"/sqlcode;
if rc <> 0 then call sqlca_error;
Address DSNREXX "EXECSQL OPEN C1";
say "RC/SQLCODE after OPEN is" rc"/sqlcode;
if rc <> 0 then call sqlca_error;
Address DSNREXX "EXECSQL FETCH C1 INTO :VAR1, :VAR2";
say "RC/SQLCODE after FETCH is" rc"/sqlcode;
if rc <> 0 then call sqlca;
say "length(var1)="length(var1) ;
say "var1=";
say var1;
Address DSNREXX "EXECSQL CLOSE C1";
say "RC/SQLCODE after CLOSE is" rc"/"sqlcode;
if rc <> 0 then call sqlca;
/**********************************************************************
* Disconnect from the DB2 system.                                    *
***********************************************************************/
"DISCONNECT";
say "RC after DISCONNECT is" rc
s_rc = RXSUBCOM("DELETE","DSNREXX","DSNREXX")
exit 0;

sqlca_error:
call sqlca
if sqlcode > 0 then return;
exit 8;
/*********************************************************************/
/* Error handling routine for bad SQL codes - just report and end. */
/*********************************************************************/
SQLCA:
*********************************************************************/
SAY "Error. SQLCODE  = >"SQLCODE"<" 
SAY "     SQLSTATE = >"SQLSTATE"<" 
SAY "     SQLERRMC = >"SQLERRMC"<" 
SAY "     SQLERRP  = >"SQLERRP"<" 
SAY "     SQLERRD.1= >"SQLERRD.1"<" 
SAY "     SQLERRD.2= >"SQLERRD.2"<" 
SAY "     SQLERRD.3= >"SQLERRD.3"<" 
SAY "     SQLERRD.4= >"SQLERRD.4"<" 
SAY "     SQLERRD.5= >"SQLERRD.5"<" 
SAY "     SQLERRD.6= >"SQLERRD.6"<" 
SAY "     SQLWARN.0= >"SQLWARN.0"<" 
SAY "     SQLWARN.1= >"SQLWARN.1"<" 
SAY "     SQLWARN.2= >"SQLWARN.2"<" 
SAY "     SQLWARN.3= >"SQLWARN.3"<" 
SAY "     SQLWARN.4= >"SQLWARN.4"<" 
SAY "     SQLWARN.5= >"SQLWARN.5"<" 
SAY "     SQLWARN.6= >"SQLWARN.6"<" 
SAY "     SQLWARN.7= >"SQLWARN.7"<" 
SAY "     SQLWARN.8= >"SQLWARN.8"<" 
SAY "     SQLWARN.9= >"SQLWARN.9"<"
SAY "     SQLWARN.10= >"SQLWARN.10"<"
return;

Defining the SQL communications area, SQLSTATE, and SQLCODE in REXX

When Db2 prepares a REXX program that contains SQL statements, Db2 automatically includes an SQLCA in the program.

About this task

The REXX SQLCA differs from the SQLCA for other languages. The REXX SQLCA consists of a set of separate variables, rather than a structure.

The SQLCA has the following forms:
- A set of simple variables
- A set of compound variables that begin with the stem SQLCA

The simple variables is the default form of the SQLCA. Using CALL SQLEXEC results in the compound stem variables. Otherwise, the attachment command used determines the form of the SQLCA. If you use the ADDRESS DSNREXX 'CONNECT' ssld syntax to connect to Db2, the SQLCA variables are a set of simple variables. If you use the CALL SQLDBS 'ATTACH TO' syntax to connect to Db2, the SQLCA variables are compound variables that begin with the stem SQLCA.

Switching forms of the SQLCA within an application is not recommended.
Related tasks
Checking the execution of SQL statements
After executing an SQL statement, your program should check for any errors before you commit the data and handle the errors that they represent.

Checking the execution of SQL statements by using the SQLCA
One way to check whether an SQL statement executed successfully is to use the SQL communication area (SQLCA). This area is set apart for communication with Db2.

Checking the execution of SQL statements by using SQLCODE and SQLSTATE
Whenever an SQL statement executes, the SQLCODE and SQLSTATE fields of the SQLCA receive a return code.

Defining the items that your program can use to check whether an SQL statement executed successfully
If your program contains SQL statements, the program should define some infrastructure so that it can check whether the statements executed successfully. You can either include an SQL communications area (SQLCA), which contains SQLCODE and SQLSTATE variables, or declare individual SQLCODE and SQLSTATE host variables.

Defining SQL descriptor areas (SQLDA) in REXX
If your program includes certain SQL statements, you must define at least one SQL descriptor area (SQLDA). Depending on the context in which it is used, the SQLDA stores information about prepared SQL statements or host variables. This information can then be read by either the application program or Db2.

Procedure
Code the SQLDA declarations directly in your program.

Each SQLDA consists of a set of REXX variables with a common stem. The stem must be a REXX variable name that contains no periods and is the same as the value of descriptor-name that you specify when you use the SQLDA in an SQL statement.

Restrictions:
• You must place SQLDA declarations before the first SQL statement that references the data descriptor, unless you use the TWOPASS SQL processing option.
• You cannot use the SQL INCLUDE statement for the SQLDA, because it is not supported in COBOL.

Related tasks
Defining SQL descriptor areas (SQLDA)
If your program includes certain SQL statements, you must define at least one SQL descriptor area (SQLDA). Depending on the context in which it is used, the SQLDA stores information about prepared SQL statements or host variables. This information can then be read by either the application program or Db2.

Related reference
SQL descriptor area (SQLDA) (Db2 SQL)

Equivalent SQL and REXX data types
All REXX data is string data. Therefore, when a REXX program assigns input data to a column, Db2 converts the data from a string type to the column type. When a REXX program assigns column data to an output variable, Db2 converts the data from the column type to a string type.

When you assign input data to a Db2 table column, you can either let Db2 determine the type that your input data represents, or you can use an SQLDA to tell Db2 the intended type of the input data.

When a REXX program assigns data to a column, it can either let Db2 determine the data type or use an SQLDA to specify the intended data type. If the program lets Db2 assign a data type for the input data, Db2 bases its choice on the input string format.
The following table shows the SQL data types that Db2 assigns to input data and the corresponding formats for that data. The two SQLTYPE values that are listed for each data type are the value for a column that does not accept null values and the value for a column that accepts null values.

<table>
<thead>
<tr>
<th>SQL data type assigned by Db2</th>
<th>SQLTYPE for data type</th>
<th>REXX input data format</th>
</tr>
</thead>
<tbody>
<tr>
<td>INTEGER</td>
<td>496/497</td>
<td>A string of numerics that does not contain a decimal point or exponent identifier. The first character can be a plus (+) or minus (-) sign. The number that is represented must be between -2,147,483,648 and 2,147,483,647, inclusive.</td>
</tr>
<tr>
<td>BIGINT</td>
<td>492/493</td>
<td>A string of numbers that does not contain a decimal point or an exponent identifier. The first character can be a plus (+) or minus (-) sign. The number that is represented must be between -9,223,372,036,854,775,808 and -2,147,483,648, inclusive, or between 2,147,483,648 and 9,223,372,036,854,775,807.</td>
</tr>
</tbody>
</table>
| DECIMAL(p,s)                 | 484/485               | One of the following formats:  
  • A string of numerics that contains a decimal point but no exponent identifier. p represents the precision and s represents the scale of the decimal number that the string represents. The first character can be a plus (+) or minus (-) sign.  
  • A string of numerics that does not contain a decimal point or an exponent identifier. The first character can be a plus (+) or minus (-) sign. The number that is represented is less than -9,223,372,036,854,775,808 or greater than 9,223,372,036,854,775,807. |
| FLOAT                        | 480/481               | A string that represents a number in scientific notation. The string consists of a series of numerics followed by an exponent identifier (an E or e followed by an optional plus (+) or minus (-) sign and a series of numerics). The string can begin with a plus (+) or minus (-) sign. |
| VARCHAR(n)                   | 448/449               | One of the following formats:  
  • A string of length n, enclosed in single or double quotation marks.  
  • The character X or x, followed by a string enclosed in single or double quotation marks. The string within the quotation marks has a length of 2n bytes and is the hexadecimal representation of a string of n characters.  
  • A string of length n that does not have a numeric or graphic format, and does not satisfy either of the previous conditions. |
| VARGRAPHIC(n)                | 464/465               | One of the following formats:  
  • The character G, g, N, or n, followed by a string enclosed in single or double quotation marks. The string within the quotation marks begins with a shift-out character (X'0E') and ends with a shift-in character (X'0F'). Between the shift-out character and shift-in character are n double-byte characters.  
  • The characters GX, Gx, gX, or gx, followed by a string enclosed in single or double quotation marks. The string within the quotation marks has a length of 4n bytes and is the hexadecimal representation of a string of n double-byte characters. |
For example, when Db2 executes the following statements to update the MIDINIT column of the EMP table, Db2 must determine a data type for HVMIDINIT:

```
SQLSTMT="UPDATE EMP",
"SET MIDINIT = ?",
"WHERE EMPNO = '000200'"
"EXEC SQL PREPARE S100 FROM :SQLSTMT"
HVMIDINIT='H'
"EXEC SQL EXECUTE S100 USING",
":"HVMIDINIT"
```

Because the data that is assigned to HVMIDINIT has a format that fits a character data type, Db2 REXX Language Support assigns a VARCHAR type to the input data.

If you do not assign a value to a host variable before you assign the host variable to a column, Db2 returns an error code.

Related concepts
Compatibility of SQL and language data types
The host variable data types that are used in SQL statements must be compatible with the data types of the columns with which you intend to use them.

Accessing the Db2 REXX language support application programming interfaces

Db2 REXX Language Support includes several application programming interfaces that enable your REXX program to connect to a Db2 subsystem and execute SQL statements.

About this task
Db2 REXX Language Support includes the following application programming interfaces:

**DSNREXX CONNECT**

Identifies the REXX task as a connected user of the specified Db2 subsystem. The DSNREXX plan resources are allocated by establishing an allied thread.

You should not confuse the DSNREXX CONNECT command with the Db2 SQL CONNECT statement.

You must execute the DSNREXX CONNECT command before your REXX program can execute SQL statements. Do not use the DSNREXX CONNECT command from a stored procedure.

A currently connected REXX task must be disconnected before switching to a different Db2 subsystem.

The syntax of the DSNREXX CONNECT command is:

```
ADDRESS DSNREXX
'CONNECT'
'subsystem-ID'
'REXX-variable'
```

Notes:
1. CALL SQLDBS 'ATTACH TO' ssid is an alternative to ADDRESS DSNREXX 'CONNECT' ssid.
2. The REXX-variable or 'subsystem-ID' string may also be a single member name in a data sharing group or the group attachment name.

The following example illustrates how to establish remote connections through the DSNREXX interface.

```
/* REXX */
/* Sample to connect to remote subsystems */
/* Connect to the local subsystem */
ADDRESS DSNREXX 'CONNECT' 'DB01'
```
/* Now connect to multiple remote subsystems */
ADDRESS DSNREXX 'EXECSQL CONNECT TO REMOTESYS1'
.
.
ADDRESS DSNREXX 'EXECSQL CONNECT TO REMOTESYS2'
.
.

**DSNREXX EXECSQL**

Executes SQL statements in REXX programs.

The syntax of the DSNREXX EXECSQL command is:

```
ADDRESS DSNREXX 'EXECSQL' "SQL-statement"
```

**Notes:**

1. CALL 'SQLEXEC' "SQL-statement" is an alternative to ADDRESS DSNREXX 'EXECSQL' "SQL-statement".
2. 'EXECSQL' and "SQL-statement" can be enclosed in either single or double quotation marks.

**DSNREXX DISCONNECT**

Deallocates the DSNREXX plan and removes the REXX task as a connected user of Db2.

You should execute the DSNREXX DISCONNECT command to release resources that are held by Db2. Otherwise resources are not released until the REXX task terminates.

Do not use the DSNREXX DISCONNECT command from a stored procedure.

The syntax of the DSNREXX DISCONNECT command is:

```
ADDRESS DSNREXX 'DISCONNECT'
```

**Note:** CALL SQLDBS 'DETACH' is an alternative to ADDRESS DSNREXX 'DISCONNECT'.

These application programming interfaces are available through the DSNREXX host command environment. To make DSNREXX available to the application, invoke the RXSUBCOM function. The syntax is:

```
RXSUBCOM ('ADD','DELETE','DSNREXX' , 'DSNREXX')
```

The ADD function adds DSNREXX to the REXX host command environment table. The DELETE function deletes DSNREXX from the REXX host command environment table.

The following example illustrates REXX code that makes DSNREXX available to an application.

```
'SUBCOM DSNREXX' /* HOST CMD ENV AVAILABLE? */
IF RC THEN /* IF NOT, MAKE IT AVAILABLE */
  S_RC = RXSUBCOM(ADD,'DSNREXX','DSNREXX')
ADDRESS DSNREXX /* ADD HOST CMD ENVIRONMENT */
  /* SEND ALL COMMANDS OTHER Than REXX instructions TO */
  /* DSNREXX */
```
Ensuring that Db2 correctly interprets character input data in REXX programs

Db2 REXX Language Support might incorrectly interpret character literals as graphic or numeric literals unless you mark them correctly.

Procedure

Precede and follow character literals with a double quotation mark, followed by a single quotation mark, followed by another double quotation mark ("'"').

For example, Specify the string the string '100' as "'100'".

Enclosing the string in apostrophes is not adequate, because REXX removes the apostrophes when it assigns a literal to a variable. For example, suppose that you want to pass the value in a host variable called stringvar to Db2. The value that you want to pass is the string '100'. First, you assign the string to the host variable by issuing the following REXX command:

```
stringvar = '100'
```

After the command executes, stringvar contains the characters 100 (without the apostrophes). Db2 REXX Language Support then passes the numeric value 100 to Db2, which is not what you intended.

However, suppose that you write the following command:

```
stringvar = ""'100''"
```

In this case, REXX assigns the string '100' to stringvar, including the single quotation marks. Db2 REXX Language Support then passes the string ‘100’ to Db2, which is the result that you want.

Passing the data type of an input data type to Db2 for REXX programs

In certain situations, you should tell Db2 the data type to use for input data in a REXX program. For example, if you are assigning or comparing input data to columns of type SMALLINT, CHAR, or GRAPHIC, you should tell Db2 to use those data types.

About this task

Db2 does not assign data types of SMALLINT, CHAR, or GRAPHIC to input data. If you assign or compare this data to columns of type SMALLINT, CHAR, or GRAPHIC, Db2 must do more work than if the data types of the input data and columns match.

Procedure

Use an SQLDA.
Examples

Example: Specifying CHAR as an input data type
Suppose that you want to tell Db2 that the data with which you update the MIDINIT column of the EMP table is of type CHAR, rather than VARCHAR. You need to set up an SQLDA that contains a description of a CHAR column, and then prepare and execute the UPDATE statement using that SQLDA, as shown in the following example.

```
INSQLDA.SQLD = 1                /* SQLDA contains one variable */
INSQLDA.1.SQLTYPE = 453         /* Type of the variable is CHAR, */
                         /* and the value can be null */
INSQLDA.1.SQLLEN  = 1           /* Length of the variable is 1 */
INSQLDA.1.SQLDATA = 'H'         /* Value in variable is H */
INSQLDA.1.SQLIND  = 0           /* Input variable is not null */
SQLSTMT="UPDATE EMP",
   "SET MIDINIT = ?",
   "WHERE EMPNO = '000200'"
"EXECSQL PREPARE S100 FROM :SQLSTMT"
"EXECSQL EXECUTE S100 USING DESCRIPTOR :INSQLDA"
```

Example: specifying the input data type as DECIMAL with precision and scale
Suppose that you want to tell Db2 that the data is of type DECIMAL with precision and nonzero scale. You need to set up an SQLDA that contains a description of a DECIMAL column, as shown in the following example.

```
INSQLDA.SQLD = 1                          /* SQLDA contains one variable */
INSQLDA.1.SQLTYPE = 484                   /* Type of variable is DECIMAL */
INSQLDA.1.SQLLEN.SQLPRECISION = 18        /* Precision of variable is 18 */
INSQLDA.1.SQLLEN.SQLSCALE = 8             /* Scale of variable is 8 */
INSQLDA.1.SQLDATA = 9876543210.87654321   /* Value in variable */
```

Related reference
SQL descriptor area (SQLDA) (Db2 SQL)
The REXX SQLDA (Db2 SQL)

Setting the isolation level of SQL statements in a REXX program

Isolation levels specify the locking behavior for SQL statements. You can set the isolation level for SQL statements in your REXX program to repeatable read (RR), read stability (RS), cursor stability (CS), or uncommitted read (UR).

**Procedure**

Execute the SET CURRENT PACKAGESET statement to select one of the following Db2 REXX Language Support packages with the isolation level that you need.

<table>
<thead>
<tr>
<th>Package name*a</th>
<th>Isolation level</th>
</tr>
</thead>
<tbody>
<tr>
<td>DSNREXRR</td>
<td>Repeatable read (RR)</td>
</tr>
<tr>
<td>DSNREXRS</td>
<td>Read stability (RS)</td>
</tr>
<tr>
<td>DSNREXCS</td>
<td>Cursor stability (CS)</td>
</tr>
<tr>
<td>DSNREXUR</td>
<td>Uncommitted read (UR)</td>
</tr>
</tbody>
</table>

*Note:*

a. These packages enable your program to access Db2 and are bound when you install Db2 REXX Language Support.

For example, to change the isolation level to cursor stability, execute the following SQL statement:

```
"EXECSQL SET CURRENT PACKAGESET='DSNREXCS'"
```
Retrieving data from Db2 tables in REXX programs

All output data in REXX programs is string data. Although, you can determine the data type that the data represents from its format and from the data type of the column from which the data was retrieved.

About this task

The following table gives the format for each type of output data.

<table>
<thead>
<tr>
<th>SQL data type</th>
<th>REXX output data format</th>
</tr>
</thead>
<tbody>
<tr>
<td>SMALLINT</td>
<td>A string of numerics that does not contain leading zeroes, a decimal point, or an exponent identifier. If the string represents a negative number, it begins with a minus (-) sign. The numeric value is between -9223372036854775808 and 9223372036854775807, inclusive.</td>
</tr>
<tr>
<td>INTEGER</td>
<td></td>
</tr>
<tr>
<td>BIGINT</td>
<td></td>
</tr>
<tr>
<td>DECIMAL(p,s)</td>
<td>A string of numerics with one of the following formats:</td>
</tr>
<tr>
<td></td>
<td>• Contains a decimal point but not an exponent identifier. The string is padded with zeroes to match the scale of the corresponding table column. If the value represents a negative number, it begins with a minus (-) sign.</td>
</tr>
<tr>
<td></td>
<td>• Does not contain a decimal point or an exponent identifier. The numeric value is less than -9223372036854775808 or greater than 9223372036854775807. If the value is negative, it begins with a minus (-) sign.</td>
</tr>
<tr>
<td>FLOAT(n)</td>
<td>A string that represents a number in scientific notation. The string consists of a numeric, a decimal point, a series of numerics, and an exponent identifier. The exponent identifier is an E followed by a minus (-) sign and a series of numerics if the number is between -1 and 1. Otherwise, the exponent identifier is an E followed by a series of numerics. If the string represents a negative number, it begins with a minus (-) sign.</td>
</tr>
<tr>
<td>REAL</td>
<td></td>
</tr>
<tr>
<td>DOUBLE</td>
<td></td>
</tr>
<tr>
<td>DECFLOAT</td>
<td>REXX emulates the DECFLOAT data type with DOUBLE, so support for DECFLOAT is limited to the REXX support for DOUBLE. The following special values are not supported:</td>
</tr>
<tr>
<td></td>
<td>• INFINITY</td>
</tr>
<tr>
<td></td>
<td>• SNAN</td>
</tr>
<tr>
<td></td>
<td>• NAN</td>
</tr>
<tr>
<td>CHAR(n)</td>
<td>A character string or LOB value of length n bytes. The string is not enclosed in single or double quotation marks.</td>
</tr>
<tr>
<td>VARCHAR(n)</td>
<td></td>
</tr>
<tr>
<td>CLOB(n)</td>
<td></td>
</tr>
<tr>
<td>BLOB(n)</td>
<td></td>
</tr>
<tr>
<td>GRAPHIC(n)</td>
<td>A string of length 2*n bytes. Each pair of bytes represents a double-byte character. This string does not contain a leading G, is not enclosed in quotation marks, and does not contain shift-out or shift-in characters.</td>
</tr>
<tr>
<td>VARGRAPHIC(n)</td>
<td></td>
</tr>
<tr>
<td>DBCLOB(n)</td>
<td></td>
</tr>
</tbody>
</table>

Because you cannot use the SELECT INTO statement in a REXX procedure, to retrieve data from a Db2 table you must prepare a SELECT statement, open a cursor for the prepared statement, and then fetch...
rows into host variables or an SQLDA using the cursor. The following example demonstrates how you can retrieve data from a Db2 table using an SQLDA:

```sql
SQLSTMT= ,
'SELECT EMPNO, FIRSTNME, MIDINIT, LASTNAME,' ,
'  WORKDEPT, PHONENO, HIREDATE, JOB,' ,
'  EDLEVEL, SEX, BIRTHDATE, SALARY,' ,
'  BONUS, COMM' ,
'  FROM EMP'
EXEC SQL DECLARE C1 CURSOR FOR S1
EXEC SQL PREPARE S1 INTO :OUTSQLDA FROM :SQLSTMT
EXEC SQL OPEN C1
Do Until(SQLCODE ¬= 0)
  EXEC SQL FETCH C1 USING DESCRIPTOR :OUTSQLDA
  If SQLCODE = 0 Then Do
    Line = ''
    Do I = 1 To OUTSQLDA.SQLD
      Line = Line OUTSQLDA.I.SQLDATA
    End I
    Say Line
  End
End
```

### Cursors and statement names in REXX

In REXX applications that contain SQL statements, you must use a predefined set of names for cursors or prepared statements. The following names are valid for cursors and prepared statements in REXX applications:

- **c1 to c100**
  - Cursor names for DECLARE CURSOR, OPEN, CLOSE, and FETCH statements. By default, c1 to c100 are defined with the WITH RETURN clause, and c51 to c100 are defined with the WITH HOLD clause. You can use the ATTRIBUTES clause of the PREPARE statement to override these attributes or add additional attributes. For example, you might want to add attributes to make your cursor scrollable.

- **c101 to c200**
  - Cursor names for ALLOCATE, DESCRIBE, FETCH, and CLOSE statements that are used to retrieve result sets in a program that calls a stored procedure.

- **s1 to s100**
  - Prepared statement names for DECLARE STATEMENT, PREPARE, DESCRIBE, and EXECUTE statements.

Use only the predefined names for cursors and statements. When you associate a cursor name with a statement name in a DECLARE CURSOR statement, the cursor name and the statement must have the same number. For example, if you declare cursor c1, you need to declare it for statement s1:

```sql
EXEC SQL 'DECLARE C1 CURSOR FOR S1'
```

Do not use any of the predefined names as host variables names.

### Handling SQL error codes in REXX applications

REXX applications can request more information about SQL error codes by using the DSNTIAR subroutine or issuing a GET DIAGNOSTICS statement.

**Procedure**

Db2 does not support the SQL WHENEVER statement in a REXXX program. To handle SQL errors and warnings, use the following methods:

- To test for SQL errors or warnings, test the SQLCODE or SQLSTATE value and the SQLWARN. values after each EXECSQL call. This method does not detect errors in the REXX interface to Db2.
- To test for SQL errors or warnings or errors or warnings from the REXX interface to Db2, test the REXX RC variable after each EXECSQL call. The following table lists the values of the RC variable.
Table 122. REXX return codes after SQL statements

<table>
<thead>
<tr>
<th>Return code</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>No SQL warning or error occurred.</td>
</tr>
<tr>
<td>+1</td>
<td>An SQL warning occurred.</td>
</tr>
<tr>
<td>-1</td>
<td>An SQL error occurred.</td>
</tr>
<tr>
<td>-3</td>
<td>The first token after ADDRESS DSNREXX is in error. For a description of the tokens allowed, see “Accessing the Db2 REXX language support application programming interfaces” on page 745.</td>
</tr>
</tbody>
</table>

You can also use the REXX SIGNAL ON ERROR and SIGNAL ON FAILURE keyword instructions to detect negative values of the RC variable and transfer control to an error routine.

Related tasks
Handling SQL error codes
Application programs can request more information about SQL error codes from Db2.

Related reference
GET DIAGNOSTICS (Db2 SQL)
Chapter 5. Calling a stored procedure from your application

To run a stored procedure, you can either call it from a client program or invoke it from the command line processor.

Before you begin

Before you call a stored procedure, ensure that you have all of the following authorizations that are required to run the stored procedure:

• Authorization to execute the stored procedure that is referenced in the CALL statement.
  The authorizations that you need depend on whether the form of the CALL statement is CALL procedure-name or CALL :host-variable.
• Authorization to execute any triggers or user-defined functions that the stored procedure invokes.
• Authorization to execute the stored procedure package and any packages under the stored procedure package.

For example, if the stored procedure invokes any user-defined functions, you need authorization to execute the packages for those user-defined functions.

About this task

An application program that calls a stored procedure can perform one or more of the following actions:

• Call more than one stored procedure.
• Call a single stored procedure more than once at the same or at different levels of nesting. However, do not assume that the variables for the stored procedures persist between calls.

If a stored procedure runs as a main program, before each call, Language Environment reinitializes the storage that is used by the stored procedure. Program variables for the stored procedure do not persist between calls.

If a stored procedure runs as a subprogram, Language Environment does not initialize the storage between calls. Program variables for the stored procedure can persist between calls. However, you should not assume that your program variables are available from one stored procedure call to another call for the following reasons:

– Stored procedures from other users can run in an instance of Language Environment between two executions of your stored procedure.
– Consecutive executions of a stored procedure might run in different stored procedure address spaces.
– The z/OS operator might refresh Language Environment between two executions of your stored procedure.

• Call a local or remote stored procedure.

If both the client and server application environments support two-phase commit, the coordinator controls updates between the application, the server, and the stored procedures. If either side does not support two-phase commit, updates fail.

• Mix CALL statements with other SQL statements.
• Use any of the Db2 attachment facilities.

Db2 runs stored procedures under the Db2 thread of the calling application, which means that the stored procedures are part of the caller’s unit of work.
**JDBC and ODBC applications:** These instructions do not apply to JDBC and ODBC applications. Instead, see the following information for how to call stored procedures from those applications:

- For ODBC applications, see Stored procedure calls in a Db2 ODBC application (Db2 Programming for ODBC).
- For JDBC applications, see Calling stored procedures in JDBC applications (Db2 Application Programming for Java)

**Procedure**

To call a stored procedure from your application:

1. Assign values to the IN and INOUT parameters.
2. Optional: To improve application performance, initialize the length of LOB output parameters to zero.
3. If the stored procedure exists at a remote location, perform the following actions:
   a) Assign values to the OUT parameters.
      
      When you call a stored procedure at a remote location, the local Db2 server cannot determine whether the parameters are input (IN) or output (OUT or INOUT) parameters. Therefore, you must initialize the values of all output parameters before you call a stored procedure at a remote location.
   
   b) Optional: Issue an explicit CONNECT statement to connect to the remote server.
      
      If you do not issue this statement explicitly, you can implicitly connect to the server by using a three-part name to identify the stored procedure in the next step.
      
      The advantage of issuing an explicit CONNECT statement is that your CALL statement, which is described in the next step, is portable to other operating systems. The advantage of implicitly connecting is that you do not need to issue this extra CONNECT statement.

   **Requirement:** When deciding whether to implicitly or explicitly connect to the remote server, consider the requirement for programs that execute the ASSOCIATE LOCATORS or DESCRIBE PROCEDURE statements. You must use the same form of the procedure name on the CALL statement and on the ASSOCIATE LOCATORS or DESCRIBE PROCEDURE statement.

4. Invoke the stored procedure with the SQL CALL statement. Make sure that you pass parameter data types that are compatible.

   If the stored procedure exists on a remote server and you did not issue an explicit CONNECT statement, specify a three-part name to identify the stored procedure, and implicitly connect to the server where the stored procedure is located.

   For native SQL procedures, the active version of the stored procedure is invoked by default. Optionally, you can specify a version of the stored procedure other than the active version.

   To allow null values for parameters, use indicator variables.

5. Optional: Retrieve the status of the procedure.
6. Process any output, including the OUT and INOUT parameters.
7. If the stored procedure returns multiple result sets, retrieve those result sets.

   **Recommendation:** Close the result sets after you retrieve them, and issue frequent commits to prevent Db2 storage shortages and EDM POOL FULL conditions.

8. For PL/I applications, also perform the following actions:
   a) Include the run time option NOEXECOPS in your source code.
   b) Specify the compile-time option SYSTEM(MVS).
      
      These additional steps ensure that the linkage conventions work correctly on z/OS.

9. For C applications, include the following line in your source code:

   ```
   #pragma runopts(PLIST(OS))
   ```
This code ensures that the linkage conventions work correctly on z/OS.

This option is not applicable to other operating systems. If you plan to use a C stored procedure on other platforms besides z/OS, use one of the forms of conditional compilation, as shown in the following example, to include this option only when you compile on z/OS.

**Form 1**

```c
#ifdef MVS
#pragma runopts(PLIST(OS))
#endif
```

**Form 2**

```c
#ifndef WKSTN
#pragma runopts(PLIST(OS))
#endif
```

10. Prepare the application as you would any other application by precompiling, compiling, and link-editing the application and binding the DBRM.

If the application calls a remote stored procedure, perform the following additional steps when you bind the DBRM:

- Bind the DBRM into a package at the local Db2 server. Use the bind option DBPROTOCOL(DRDA). If the stored procedure name cannot be resolved until run time, also specify the bind option VALIDATE(RUN). The stored procedure name might not be resolved at run time if you use a variable for the stored procedure name or if the stored procedure exists on a remote server.
- Bind the DBRM into a package at the remote Db2 server. If your client program accesses multiple servers, bind the program at each server.
- Bind all packages into a plan at the local Db2 server. Use the bind option DBPROTOCOL(DRDA).

11. Ensure that stored procedure completed successfully.

If a stored procedure abnormally terminates, Db2 performs the following actions:

- The calling program receives an SQL error as notification that the stored procedure failed.
- Db2 places the calling program's unit of work in a must-rollback state.
- Db2 stops the stored procedure, and subsequent calls fail, in either of the following conditions:
  - The number of abnormal terminations equals the STOP AFTER n FAILURES value for the stored procedure.
  - The number of abnormal terminations equals the default MAX ABEND COUNT value for the subsystem.
- The stored procedure does not handle the abend condition, and Db2 refreshes the environment for Language Environment to recover the storage that the application uses. In most cases, the environment does not need to restart.
- A data set is allocated in the DD statement CEEDUMP in the JCL procedure that starts the stored procedures address space. In this case, Language Environment writes a small diagnostic dump to this data set. Use the information in the dump to debug the stored procedure.
- In a data sharing environment, the stored procedure is placed in STOPABN status only on the member where the abends occurred. A calling program can invoke the stored procedure from other members of the data sharing group. The status on all other members is STARTED.

**Examples**

**Example 1: Simple CALL statement**

The following example shows a simple CALL statement that you might use to invoke stored procedure A:

```sql
EXEC SQL CALL A (:EMP, :PRJ, :ACT, :EMT, :EMS, :EME, :TYPE, :CODE);
```
In this example, :EMP, :PRJ, :ACT, :EMT, :EMS, :EME, :TYPE, and :CODE are host variables that you have declared earlier in your application program.

**Example 2: Using a host structure for multiple parameter values**

Instead of passing each parameter separately, as shown in the example of a simple CALL statement, you can pass them together as a host structure. For example, assume that you defined the following host structure in your application:

```c
struct {
    char EMP[7];
    char PRJ[7];
    short ACT;
    short EMT;
    char EMS[11];
    char EME[11];
} empstruc;
```

You can then issue the following CALL statement to invoke stored procedure A:

```sql
EXEC SQL CALL A (:empstruc, :TYPE, :CODE);
```

**Example 3: Calling a remote stored procedure**

- The following example shows how to explicitly connect to LOCA and then issue a CALL statement:

```sql
EXEC SQL CONNECT TO LOCA;
EXEC SQL CALL SCHEMAA.A (:EMP, :PRJ, :ACT, :EMT, :EMS, :EME,
                         :TYPE, :CODE);
```

- The following example shows how to implicitly connect to LOCA by specifying the three-part name for stored procedure A in the CALL statement:

```sql
EXEC SQL CALL LOCA.SCHEMAA.A (:EMP, :PRJ, :ACT, :EMT, :EMS,
                                :EME, :TYPE, :CODE);
```

**Example 4: Passing parameters that can have null values**

The preceding examples assume that none of the input parameters can have null values. The following example shows how to allow for null values for the parameters by passing indicator variables in the parameter list:

```sql
EXEC SQL CALL A (:EMP :IEMP, :PRJ :IPRJ, :ACT :IACT,
                 :EMT :IEMT, :EMS :IEMS, :EME :IEME,
                 :TYPE :ITYPE, :CODE :ICODE);
```

In this example, :IEMP, :IPRJ, :IACT, :IEMT, :IEMS, :IEME, :ITYPE, and :ICODE are indicator variables for the parameters.

**Example 5: Passing string constants and null values**

The following example CALL statement passes integer and character string constants, a null value, and several host variables:

```sql
EXEC SQL CALL A ('000130', 'IF1000', 90, 1.0, NULL, '2009-10-01',
                 :TYPE, :CODE);
```

**Example 6: of using a host variable for the stored procedure name**

The following example CALL statement uses a host variable for the name of the stored procedure:

```sql
EXEC SQL CALL :procnm (:EMP, :PRJ, :ACT, :EMT, :EMS, :EME,
                       :TYPE, :CODE);
```

Assume that the stored procedure name is A. The host variable `procnm` is a character variable of length 255 or less that contains the value 'A'. Use this technique if you do not know in advance the name of the stored procedure, but you do know the parameter list convention.

**Example 7: Using an SQLDA to pass parameters in a single structure**

The following example CALL statement shows how to pass parameters in a single structure, the SQLDA, rather than as separate host variables:
EXEC SQL CALL A USING DESCRIPTOR :sqlda;

sqlda is the name of an SQLDA.

One advantage of using an SQLDA is that you can change the encoding scheme of the stored procedure parameter values. For example, if the subsystem on which the stored procedure runs has an EBCDIC encoding scheme, and you want to retrieve data in ASCII CCSID 437, you can specify the CCSIDs for the output parameters in the SQLVAR fields of the SQLDA.

This technique for overriding the CCSIDs of parameters is the same as the technique for overriding the CCSIDs of variables. This technique involves including dynamic SQL for varying-list SELECT statements in your program. When you use this technique, the defined encoding scheme of the parameter must be different from the encoding scheme that you specify in the SQLDA. Otherwise, no conversion occurs.

The defined encoding scheme for the parameter is the encoding scheme that you specify in the CREATE PROCEDURE statement. If you do not specify an encoding scheme in this statement, the defined encoding scheme for the parameter is the default encoding scheme for the subsystem.

Example 8: Reusable CALL statement

Because the following example CALL statement uses a host variable name for the stored procedure and an SQLDA for the parameter list, it can be reused to call different stored procedures with different parameter lists:

EXEC SQL CALL :procnm USING DESCRIPTOR :sqlda;

Your client program must assign a stored procedure name to the host variable procnm and load the SQLDA with the parameter information before issuing the SQL CALL statement.

Related concepts

Stored procedure parameters

You can pass information between a stored procedure and the calling application program by using parameters. Applications pass the required parameters in the SQL CALL statement. Optionally, the application can also include an indicator variable with each parameter to allow for null values or to pass large output parameter values.

Related tasks

Including dynamic SQL for varying-list SELECT statements in your program

A varying-list SELECT statement returns rows that contain an unknown number of values of unknown type. When you use this type of statement, you do not know in advance exactly what kinds of host variables you need to declare for storing the results.

Preparing an application to run on Db2 for z/OS

To prepare and run applications that contain embedded static SQL statements or dynamic SQL statements, you must precompile, compile, link-edit, and bind them.

Managing authorization for stored procedures (Managing Security)

Temporarily overriding the active version of a native SQL procedure

If you want a particular call to a native SQL procedure to use a version other than the active version, you can temporarily override the active version. Such an override might be helpful when you are testing a new version of a native SQL procedure.

Related reference

Statements (Db2 SQL)

Procedures that are supplied with Db2 (Db2 SQL)
Passing large output parameters to stored procedures by using indicator variables

If any output parameters occupy a large amount of storage, passing the entire storage area to a stored procedure can degrade performance.

About this task

In the calling program, you can specify indicator variables for large output parameters to pass only a 2-byte area to the stored procedure, but receive the entire output data area from the stored procedure. When Db2 processes the CALL statement, it inspects the parameters before moving any data. If an output parameter has a NULL indicator, Db2 determines that it does not need to copy the associated data area to the Db2 address space, which avoids the need for acquisition of extra buffers or cross-memory moves.

You can use the following procedure regardless of whether the linkage convention for the stored procedure is GENERAL, GENERAL WITH NULLS, or SQL.

Procedure

To pass large output parameters to stored procedures by using indicator variables:

1. Declare an indicator variable for every large output parameter in the stored procedure.
   
   If you are using the GENERAL WITH NULLS or SQL linkage convention, you must declare indicator variables for all of your parameters. In this case, you do not need to declare another indicator variable.

2. Assign a negative value to each indicator variable that is associated with a large output variable.

3. Include the indicator variables in the CALL statement.

Example

For example, suppose that a stored procedure that is defined with the GENERAL linkage convention takes one integer input parameter and one character output parameter of length 6000. You do not want to pass the 6000 byte storage area to the stored procedure. The following example PL/I program passes only 2 bytes to the stored procedure for the output variable and receives all 6000 bytes from the stored procedure:

```
DCL INTVAR BIN FIXED(31);   /* This is the input variable */
DCL BIGVAR(6000);           /* This is the output variable */
DCL I1 BIN FIXED(15);       /* This is an indicator variable */
⋮
I1 = -1;                    /* Setting I1 to -1 causes only */
/* a two byte area representing */
/* I1 to be passed to the */
/* stored procedure, instead of */
/* the 6000 byte area for BIGVAR*/
EXEC SQL CALL PROCX(:INTVAR, :BIGVAR INDICATOR :I1);
```

Related reference

Linkage conventions for external stored procedures
The linkage convention for a stored procedure can be either GENERAL, GENERAL WITH NULLS, or SQL. These linkage conventions apply to only external stored procedures.

Data types for calling stored procedures

The data types that are available for calling applications are the same as the data types that are used when retrieving or updating stored procedures.

The format of the parameters that you pass in the CALL statement in an application must be compatible with the data types of the parameters in the CREATE PROCEDURE statement.
For languages other than REXX

For all data types except LOBs, ROWIDs, locators, and VARCHARs (for C language), see the tables listed in the following table for the host data types that are compatible with the data types in the stored procedure definition.

Table 123. Listing of tables of compatible data types

<table>
<thead>
<tr>
<th>Language</th>
<th>Compatible data types table</th>
</tr>
</thead>
<tbody>
<tr>
<td>Assembler</td>
<td>“Equivalent SQL and assembler data types” on page 564</td>
</tr>
<tr>
<td>C</td>
<td>“Equivalent SQL and C data types” on page 611</td>
</tr>
<tr>
<td>COBOL</td>
<td>“Equivalent SQL and COBOL data types” on page 677</td>
</tr>
<tr>
<td>PL/I</td>
<td>“Equivalent SQL and PL/I data types” on page 720</td>
</tr>
</tbody>
</table>

Calling a stored procedure from a REXX procedure

The format of the parameters that you pass in the CALL statement in a REXX procedure must be compatible with the data types of the parameters in the CREATE PROCEDURE statement.

The following table lists each SQL data type that you can specify for the parameters in the CREATE PROCEDURE statement and the corresponding format for a REXX parameter that represents that data type.

Table 124. Parameter formats for a CALL statement in a REXX procedure

<table>
<thead>
<tr>
<th>SQL data type</th>
<th>REXX format</th>
</tr>
</thead>
<tbody>
<tr>
<td>SMALLINT</td>
<td>A string of numerics that does not contain a decimal point or exponent</td>
</tr>
<tr>
<td>INTEGER</td>
<td>identifier. The first character can be a plus or minus sign. This format also</td>
</tr>
<tr>
<td>BIGINT</td>
<td>applies to indicator variables that are passed as parameters.</td>
</tr>
<tr>
<td>DECIMAL(p,s)</td>
<td>A string of numerics that has a decimal point but no exponent identifier.</td>
</tr>
<tr>
<td>NUMERIC(p,s)</td>
<td>The first character can be a plus or minus sign.</td>
</tr>
<tr>
<td>REAL</td>
<td>A string that represents a number in scientific notation. The string consists</td>
</tr>
<tr>
<td>FLOAT(n)</td>
<td>of a series of numerics followed by an exponent identifier (an E or e</td>
</tr>
<tr>
<td>DOUBLE</td>
<td>followed by an optional plus or minus sign and a series of numerics).</td>
</tr>
<tr>
<td>DECFLOAT</td>
<td>A string of numerics that does not contain a decimal point or exponent</td>
</tr>
<tr>
<td>CHARACTER(n)</td>
<td>identifier. The first character can be a plus or minus sign.</td>
</tr>
<tr>
<td>VARCHAR(n)</td>
<td>A string of length $n$, enclosed in single quotation marks.</td>
</tr>
<tr>
<td>VARCHAR(n) FOR BIT DATA</td>
<td></td>
</tr>
<tr>
<td>GRAPHIC(n)</td>
<td>The character G followed by a string enclosed in single quotation marks.</td>
</tr>
<tr>
<td>VARGRAPHIC(n)</td>
<td>The string within the quotation marks begins with a shift-out character ($X'OE'$) and</td>
</tr>
<tr>
<td></td>
<td>ends with a shift-in character ($X'0F'$). Between the shift-out character and</td>
</tr>
<tr>
<td></td>
<td>shift-in character are $n$ double-byte characters.</td>
</tr>
<tr>
<td>BINARY</td>
<td>Recommendation: Pass BINARY and VARBINARY values by using the SQLDA.</td>
</tr>
<tr>
<td>VARBINARY</td>
<td>If you specify an SQLDA when you call the stored procedure, set the SQLTYPE</td>
</tr>
<tr>
<td></td>
<td>in the SQLDA. SQLDATA is a string of characters.</td>
</tr>
<tr>
<td></td>
<td>If you use host variables, the REXX format of BINARY and VARBINARY data is</td>
</tr>
<tr>
<td></td>
<td>BX followed by a string that is enclosed in a single quotation mark.</td>
</tr>
</tbody>
</table>

Chapter 5. Calling a stored procedure from your application
<table>
<thead>
<tr>
<th>SQL data type</th>
<th>REXX format</th>
</tr>
</thead>
<tbody>
<tr>
<td>DATE</td>
<td>A string of length 10, enclosed in single quotation marks. The format of the string depends on the value of field DATE FORMAT that you specify when you install Db2.</td>
</tr>
<tr>
<td>TIME</td>
<td>A string of length 8, enclosed in single quotation marks. The format of the string depends on the value of field TIME FORMAT that you specify when you install Db2.</td>
</tr>
<tr>
<td>TIMESTAMP</td>
<td>A string of length 19 to 32, enclosed in single quotation marks. The string has the format yyyy-mm-dd.hh.mm.ss or yyyy-mm-dd-hh.mm.ss.nnnnnnnnnnn, where the number of fractional second digits can range 0 - 12.</td>
</tr>
<tr>
<td>TIMESTAMP WITH TIME ZONE</td>
<td>A string of length 148 to 161, enclosed in single quotation marks. The string has the format yyyymm-dd-mm.ss.nnnnnnnnnnn±th:tm or yyyymm-dd-mm.ss.nnnnnnnnnnn±th:tm, where the number of fractional second digits can range 0 - 12.</td>
</tr>
<tr>
<td>XML</td>
<td>No equivalent.</td>
</tr>
</tbody>
</table>

The following figure demonstrates how a REXX procedure calls the stored procedure in “REXX stored procedures” on page 287. The REXX procedure performs the following actions:

- Connects to the Db2 subsystem that was specified by the REXX procedure invoker.
- Calls the stored procedure to execute a Db2 command that was specified by the REXX procedure invoker.
- Retrieves rows from a result set that contains the command output messages.

```rexx
/* REXX */
PARSE ARG SSID COMMAND         /* Get the SSID to connect to */
/* and the DB2 command to be */
/* executed */

/* Set up the host command environment for SQL calls. */
"SUBCOM DSNREXX" /* Host cmd env available? */
IF RC THEN /* No--make one */
   S_RC = RXSUBCOM('ADD','DSNREXX','DSNREXX')
ENDIF /* Connect to the DB2 subsystem. */
ADDRESS DSNREXX "CONNECT" SSID
IF SQLCODE = 0 THEN CALL SQLCA
PROC = 'COMMAND'
RESULTSIZE = 32703
RESULT = LEFT(' ',RESULTSIZE,' ')

/* Call the stored procedure that executes the DB2 command. */
/* The input variable (COMMAND) contains the DB2 command. */
/* The output variable (RESULT) will contain the return area */
/* from the IFI COMMAND call after the stored procedure executes. */
ADDRESS DSNREXX "EXECSQL",
"CALL" PROC "(:COMMAND, :RESULT)"
IF SQLCODE < 0 THEN CALL SQLCA
SAY 'RETCODE ='RETCODE
SAY 'SQLCODE ='SQLCODE
SAY 'SQLERRMC ='SQLERRMC
SAY 'SQLERRP ='SQLERRP
SAY 'SQLERRD.='SQLERRD.1',
   | SQLERRD.2',
   | SQLERRD.3',
   | SQLERRD.4',
   | SQLERRD.5',
   | SQLERRD.6
SAY 'SQLWARN.='SQLWARN.0',
   | SQLWARN.1',
```
Related concepts
REXX stored procedures
A REXX stored procedure is similar to any other REXX procedure and follows the same rules as stored procedures in other languages. A REXX stored procedure receives input parameters, executes REXX commands, optionally executes SQL statements, and returns at most one output parameter. However, a few differences exist.

Preparing a client program that calls a remote stored procedure

If you call a remote stored procedure from an embedded SQL application, you need to do a few extra steps when you prepare the client program. You do not need to do any extra steps when you prepare the stored procedure.

Before you begin
For an ODBC or CLI application, ensure that the Db2 packages and plan that are associated with the ODBC driver are bound to Db2. These packages and plan must be bound before you can run your application.

Procedure
To prepare a client program that calls a remote stored procedure:

1. Precompile, compile, and link-edit the client program on the local Db2 subsystem.
2. Bind the resulting DBRM into a package at the local Db2 subsystem by using the BIND PACKAGE command with the option DBPROTOCOL(DRDA).
3. Bind the same DBRM, the one for the client program, into a package at the remote location by using the BIND PACKAGE command and specifying a location name. If your client program needs to access multiple servers, bind the program at each server.
   For example, suppose that you want a client program to call a stored procedure at location LOCA. You precompile the program to produce DBRM A. Then you can use the following command to bind DBRM A into package collection COLLA at location LOCA:

   ```
   BIND PACKAGE (LOCA.COLLA) MEMBER(A)
   ```

4. Bind all packages into a plan on the local Db2 subsystem. Specify the bind option DBPROTOCOL(DRDA).
5. Bind any stored procedures that run under Db2 ODBC on a remote Db2 database server as a package at the remote site.
   Those procedures do not need to be bound into the Db2 ODBC plan.

Related tasks
Binding DBRMs to create packages (Db2 Programming for ODBC)

Related reference
BIND PACKAGE (DSN) (Db2 Commands)
How Db2 determines which stored procedure to run

A procedure is uniquely identified by its name and its qualifying schema name. You can tell Db2 exactly which stored procedure to run by qualifying it with its schema name when you call it. Otherwise, Db2 determines which stored procedure to run.

However, if you do not qualify the stored procedure name, Db2 uses the following method to determine which stored procedure to run:

1. Db2 searches the list of schema names from the PATH bind option or the CURRENT PATH special register from left to right until it finds a schema name for which a stored procedure definition exists with the name in the CALL statement.

   Db2 uses schema names from the PATH bind option for CALL statements of the following form:

   ```sql
   CALL procedure-name
   ```

   Db2 uses schema names from the CURRENT PATH special register for CALL statements of the following form:

   ```sql
   CALL host-variable
   ```

2. When Db2 finds a stored procedure definition, Db2 executes that stored procedure if the following conditions are true:
   - The caller is authorized to execute the stored procedure.
   - The stored procedure has the same number of parameters as in the CALL statement.

   If both conditions are not true, Db2 continues to go through the list of schemas until it finds a stored procedure that meets both conditions or reaches the end of the list.

3. If Db2 cannot find a suitable stored procedure, it returns an SQL error code for the CALL statement.

Calling different versions of a stored procedure from a single application

You can call different versions of a stored procedure from the same application program, even though those versions all have the same load module name.

Procedure

To call different versions of a stored procedure from a single application:

1. When you define each version of the stored procedure, use the same stored procedure name but different schema names, different COLLID values, and different WLM environments.

2. In the program that invokes the stored procedure, specify the unqualified stored procedure name in the CALL statement.

3. Use the SQL path to indicate which version of the stored procedure that the client program should call. You can choose the SQL path in several ways:

   - If the client program is not an ODBC or JDBC application, use one of the following methods:
     - Use the CALL `procedure-name` form of the CALL statement. When you bind plans or packages for the program that calls the stored procedure, bind one plan or package for each version of the stored procedure that you want to call. In the PATH bind option for each plan or package, specify the schema name of the stored procedure that you want to call.
     - Use the CALL `host-variable` form of the CALL statement. In the client program, use the SET PATH statement to specify the schema name of the stored procedure that you want to call.
   
   - If the client program is an ODBC or JDBC application, choose one of the following methods:
     - Use the SET PATH statement to specify the schema name of the stored procedure that you want to call.
When you bind the stored procedure packages, specify a different collection for each stored procedure package. Use the COLLID value that you specified when defining the stored procedure to Db2.

4. When you run the client program, specify the plan or package with the PATH value that matches the schema name of the stored procedure that you want to call.

Results

For example, suppose that you want to write one program, PROGY, that calls one of two versions of a stored procedure named PROCX. The load module for both stored procedures is named SUMMOD. Each version of SUMMOD is in a different load library. The stored procedures run in different WLM environments, and the startup JCL for each WLM environment includes a STEPLIB concatenation that specifies the correct load library for the stored procedure module.

First, define the two stored procedures in different schemas and different WLM environments:

```sql
CREATE PROCEDURE TEST.PROCX(IN V1 INTEGER, OUT V2 CHAR(9))
LANGUAGE C
EXTERNAL NAME SUMMOD
WLM ENVIRONMENT TESTENV;
```

```sql
CREATE PROCEDURE PROD.PROCX(IN V1 INTEGER, OUT V2 CHAR(9))
LANGUAGE C
EXTERNAL NAME SUMMOD
WLM ENVIRONMENT PRODENV;
```

When you write CALL statements for PROCX in program PROGY, use the unqualified form of the stored procedure name:

```sql
CALL PROCX(V1,V2);
```

Bind two plans for PROGY. In one BIND statement, specify PATH(TEST). In the other BIND statement, specify PATH(PROD).

To call TEST.PROCX, execute PROGY with the plan that you bound with PATH(TEST). To call PROD.PROCX, execute PROGY with the plan that you bound with PATH(PROD).

Invoking multiple instances of a stored procedure

Your application program can issue multiple CALL statements to the same local or remote stored procedure. Assume that your stored procedure returns result sets and the calling application leaves those result sets open before the next call to that same stored procedure. In that case, each CALL statement invokes a unique instance of the stored procedure.

About this task

When you invoke multiple instances of a stored procedure, each instance runs serially within the same Db2 thread and opens its own result sets. These multiple calls invoke multiple instances of any packages that are invoked while running the stored procedure. These instances are invoked at either the same or different level of nesting under one Db2 connection or thread.

For local stored procedures that issue remote SQL, instances of the applications are created at the remote server site. These instances are created regardless of whether result sets exist or are left open between calls.

If you call too many instances of a stored procedure or if you open too many cursors, Db2 storage shortages and EDM POOL FULL conditions might occur. If the stored procedure issues remote SQL statements to another Db2 server, these conditions can occur at both the Db2 client and at the Db2 server.
Procedure

To invoke multiple instances of a stored procedure:

1. To optimize storage usage and prevent storage shortages, ensure that you specify appropriate values for the following two subsystem parameters:

   **MAX_ST_PROC**
   - Controls the maximum number of stored procedure instances that you can call within the same thread.

   **MAX_NUM_CUR**
   - Controls the maximum number of cursors that can be opened by the same thread.

   When either of the values from these subsystem parameters is exceeded while an application is running, the CALL statement or the OPEN statement receives SQLCODE -904.

2. In your application, issue CALL statements to the stored procedure.

3. In the calling application for the stored procedure, close the result sets and issue frequent commits. Even read-only applications should perform these actions.

   Applications that fail to close result sets or issue an adequate number of commits might terminate abnormally with Db2 storage shortage and EDM POOL FULL conditions.

Related reference

- MAX OPEN CURSORS field (MAX_NUM_CUR subsystem parameter) (Db2 Installation and Migration)
- MAX STORED PROCS field (MAX_ST_PROC subsystem parameter) (Db2 Installation and Migration)
- CALL (Db2 SQL)

### Designating the active version of a native SQL procedure

When a native SQL procedure is called, Db2 uses the version that is designated as the active version.

#### About this task

When you create a native SQL procedure, that first version is by default the active version. If you create additional versions of a stored procedure, you can designate another version to be the active version.

**Exception:** If an existing active version is still being used by a process, the new active version is not used until the next call to that procedure.

#### Procedure

To designate the active version of a native SQL procedure, issue an ALTER PROCEDURE statement with the following items:

- The name of the native SQL procedure for which you want to change the active version.
- The ACTIVATE VERSION clause with the name of the version that you want to be active.

When the ALTER statement is committed, the new version of the procedure becomes the active version and is used by the next call for that procedure.

#### Example

The following ALTER PROCEDURE statement makes version V2 of the UPDATE_BALANCE procedure the active version.

```sql
ALTER PROCEDURE UPDATE_BALANCE
ACTIVATE VERSION V2;
```
Temporarily overriding the active version of a native SQL procedure

If you want a particular call to a native SQL procedure to use a version other than the active version, you can temporarily override the active version. Such an override might be helpful when you are testing a new version of a native SQL procedure.

About this task

Recommendation: If you want all calls to a native SQL procedure to use a particular version, do not temporarily override the active version in every call. Instead, make that version the active version. Otherwise, performance might be slower.

Procedure

To temporarily override the active version of a native SQL procedure, specify the following statements in your program:

1. The SET CURRENT ROUTINE VERSION statement with the name of the version of the procedure that you want to use. If the specified version does not exist, the active version is used.
2. The CALL statement with the name of the procedure.

Example

The following CALL statement invokes version V1 of the UPDATE_BALANCE procedure, regardless of what the current active version of that procedure is.

```
SET CURRENT ROUTINE VERSION = V1;
SET procname = 'UPDATE_BALANCE';
CALL :procname USING DESCRIPTOR :x;
```

Specifying the number of stored procedures that can run concurrently

Multiple stored procedures can run concurrently, each under its own z/OS task control block (TCB). The z/OS Workload Manager (WLM) manages how many concurrent stored procedures can run in an address space. The number of concurrent stored procedures in an address space cannot exceed the value of the NUMTCB field that was specified on the DSNTIPX installation panel, during Db2 installation.

Procedure

You can override that value in the following ways:

- Edit the JCL procedures that start stored procedures address spaces, and modify the value of the NUMTCB parameter.
- Specify the following parameter in the Start Parameters field of the Create An Application Environment panel when you set up a WLM application environment:

```
NUMTCB=number-of-TCBs
```

Special cases:

- For REXX stored procedures, you must set the NUMTCB parameter to 1.
- Stored procedures that invoke utilities can invoke only one utility at a time in a single address space. Consequently, the value of the NUMTCB parameter is forced to 1 for those procedures.

Related concepts

Installation step 21: Configure Db2 for running stored procedures and user-defined functions (Db2 Installation and Migration)
Migration step 23: Configure Db2 for running stored procedures and user-defined functions (optional) (Db2 Installation and Migration)

Related tasks
Maximizing the number of procedures or functions that run in an address space (Db2 Performance)

Retrieving the procedure status

When an SQL procedure returns control to the calling program, it also returns the procedure status. The status is an integer value that indicates the success of the procedure.

About this task

Db2 sets the status to 0 or -1 depending on the value of the SQLCODE. Alternatively, an SQL procedure can set the integer status value by using the RETURN statement. In this case, Db2 sets the SQLCODE in the SQLCA to 0.

Procedure

To retrieve the procedure status, perform one of the following actions in the calling program:

- Issue the GET DIAGNOSTICS statement with the DB2_RETURN_STATUS item. The specified host variable in the GET DIAGNOSTICS statement is set to one of the following values:
  - 0
    - This value indicates that the procedure returned with an SQLCODE that is greater or equal to zero. You can access the value directly from the SQLCA by retrieving the value of SQLERRD(1). For C applications, retrieve SQLERRD[0].
  - -1
    - This value indicates that the procedure returned with an SQLCODE that is less than zero. In this case, the SQLERRD(1) value in the SQLCA is not set. Db2 returns -1 only.
  - n
    - Any value other than 0 or -1 is the return value that was explicitly set in the procedure with the RETURN statement.

For example, the following SQL code creates an SQL procedure that is named TESTIT, which calls another SQL procedure that is named TRYIT. The TRYIT procedure returns a status value. The TESTIT procedure retrieves that value with the DB2_RETURN_STATUS item of the GET DIAGNOSTICS statement.

```
CREATE PROCEDURE TESTIT ()
LANGUAGE SQL
A1:BEGINDECLARE RETVAL INTEGER DEFAULT 0;
    ...CALL TRYIT;
    GET DIAGNOSTICS RETVAL = DB2_RETURN_STATUS;
    IF RETVAL <> 0 THEN
        ...LEAVE A1;
    ELSE
        ...END IF;
    END A1
```

- Retrieve the value of SQLERRD(1) in the SQLCA. For C applications, retrieve SQLERRD[0]. This field contains the integer value that was set by the RETURN statement in the SQL procedure. This method is not applicable if the status was set by Db2.

Related concepts
SQL communication area (SQLCA) (Db2 SQL)

Related reference
GET DIAGNOSTICS (Db2 SQL)
Writing a program to receive the result sets from a stored procedure

You can write a program to receive results set from a stored procedure for either a fixed number of result sets, for which you know the contents, or a variable number of result sets, for which you do not know the contents.

About this task

A program for a fixed number of result sets is simpler to write than a program for a variable number of result sets. However, if you write a program for a variable number of result sets, you do not need to make modifications to the program if the stored procedure changes.

If your program calls an SQL procedure that returns result sets, you must write the program for a fixed number of result sets.

In the following steps, you do not need to connect to the remote location when you execute these statements:

- DESCRIBE PROCEDURE
- ASSOCIATE LOCATORS
- ALLOCATE CURSOR
- DESCRIBE CURSOR
- FETCH
- CLOSE

Procedure

To write a program to receive the result sets from a stored procedure:

1. Declare a locator variable for each result set that is to be returned.

   If you do not know how many result sets are to be returned, declare enough result set locators for the maximum number of result sets that might be returned.

2. Call the stored procedure and check the SQL return code.

   If the SQLCODE from the CALL statement is +466, the stored procedure has returned result sets.

3. Determine how many result sets the stored procedure is returning.

   If you already know how many result sets the stored procedure returns, skip this step.

   Use the SQL statement DESCRIBE PROCEDURE to determine the number of result sets. DESCRIBE PROCEDURE places information about the result sets in an SQLDA. Make this SQLDA large enough to hold the maximum number of result sets that the stored procedure might return. When the DESCRIBE PROCEDURE statement completes, the fields in the SQLDA contain the following values:

   - SQLD contains the number of result sets that are returned by the stored procedure.
   - Each SQLVAR entry gives the following information about a result set:
     - The SQLNAME field contains the name of the SQL cursor that is used by the stored procedure to return the result set.
     - The SQLIND field contains the value -1, which indicates that no estimate of the number of rows in the result set is available.
     - The SQLDATA field contains the value of the result set locator, which is the address of the result set.

4. Link result set locators to result sets by performing one of the following actions:

   - Use the ASSOCIATE LOCATORS statement. You must embed this statement in an application or SQL procedure. The ASSOCIATE LOCATORS statement assigns values to the result set locator variables.
If you specify more locators than the number of result sets that are returned, Db2 ignores the extra locators.

• If you executed the DESCRIBE PROCEDURE statement previously, the result set locator values are in the SQLDATA fields of the SQLDA. You can copy the values from the SQLDATA fields to the result set locators manually, or you can execute the ASSOCIATE LOCATORS statement to do it for you.

The stored procedure name that you specify in an ASSOCIATE LOCATORS statement or DESCRIBE PROCEDURE statement must match the stored procedure name in the CALL statement as follows:

• If the name is unqualified in the CALL statement, do not qualify it.
• If the name is qualified with a schema name in the CALL statement, qualify it with the schema name.
• If the name is qualified with a location name and schema name in the CALL statement, qualify it with a location name and schema name.

5. Allocate cursors for fetching rows from the result sets.

Use the SQL statement ALLOCATE CURSOR to link each result set with a cursor. Execute one ALLOCATE CURSOR statement for each result set. The cursor names can differ from the cursor names in the stored procedure.

To use the ALLOCATE CURSOR statement, you must embed it in an application or SQL procedure.

6. Determine the contents of the result sets.

If you already know the format of the result set, skip this step.

Use the SQL statement DESCRIBE CURSOR to determine the format of a result set and put this information in an SQLDA. For each result set, you need an SQLDA that is big enough to hold descriptions of all columns in the result set.

You can use DESCRIBE CURSOR for only those cursors for which you executed ALLOCATE CURSOR previously.

After you execute DESCRIBE CURSOR, if the cursor for the result set is declared WITH HOLD, the high-order bit of byte 8 of field SQLDAID in the SQLDA is set to 1.

7. Fetch rows from the result sets into host variables by using the cursors that you allocated with the ALLOCATE CURSOR statements.

Fetching rows from a result set is the same as fetching rows from a table.

If you executed the DESCRIBE CURSOR statement, perform the following steps before you fetch the rows:

a. Allocate storage for host variables and indicator variables. Use the contents of the SQLDA from the DESCRIBE CURSOR statement to determine how much storage you need for each host variable.

b. Put the address of the storage for each host variable in the appropriate SQLDATA field of the SQLDA.

c. Put the address of the storage for each indicator variable in the appropriate SQLIND field of the SQLDA.

Example

The following examples show C language code that accomplishes each of these steps. Coding for other languages is similar.

The following example demonstrates how to receive result sets when you know how many result sets are returned and what is in each result set.

```sql
EXEC SQL BEGIN DECLARE SECTION;
static volatile SQL TYPE IS RESULT_SET_LOCATOR *loc1, *loc2;
EXEC SQL END DECLARE SECTION;
```
The following example demonstrates how to receive result sets when you do not know how many result sets are returned or what is in each result set.

```
EXEC SQL BEGIN DECLARE SECTION;
static volatile SQL TYPE IS RESULT_SET_LOCATOR *loc1, *loc2, *loc3;
EXEC SQL END DECLARE SECTION;
```

```
EXEC SQL CALL P2(:parm1, :parm2, ...);
if(SQLCODE==+466)
{
    EXEC SQL DESCRIBE PROCEDURE P2 INTO :proc_da;
    ...
    if (proc_da->number = 3)
    {
        EXEC SQL ASSOCIATE LOCATORS (:loc1, :loc2, :loc3) WITH PROCEDURE P2;
        ...
    }
    else
    {
        EXEC SQL ASSOCIATE LOCATORS (:loc1, :loc2) WITH PROCEDURE P2;
        ...
    }
}
```
EXEC SQL ALLOCATE C2 CURSOR FOR RESULT SET :loc2;
EXEC SQL ALLOCATE C3 CURSOR FOR RESULT SET :loc3;

The following example demonstrates how you can use an SQL procedure to receive result sets. The logic assumes that no handler exists to intercept the +466 SQLCODE, such as DECLARE CONTINUE HANDLER FOR SQLWARNING .... Such a handler causes SQLCODE to be reset to zero. Then the test for IF SQLCODE = 466 is never true and the statements in the IF body are never executed.

DECLARE RESULT1 RESULT_SET_LOCATOR VARYING;
DECLARE RESULT2 RESULT_SET_LOCATOR VARYING;
DECLARE AT_END, VAR1, VAR2 INT DEFAULT 0;
DECLARE SQLCODE INTEGER DEFAULT 0;
DECLARE CONTINUE HANDLER FOR NOT FOUND SET AT_END = 99;
SET TOTAL1 = 0;
SET TOTAL2 = 0;
CALL TARGETPROCEDURE();
IF SQLCODE = 466 THEN
  ASSOCIATE RESULT SET LOCATORS(RESULT1,RESULT2)
  WITH PROCEDURE SPDG3091;
  ALLOCATE RSCUR1 CURSOR FOR RESULT1;
  ALLOCATE RSCUR2 CURSOR FOR RESULT2;
  WHILE AT_END = 0 DO
    FETCH RSCUR1 INTO VAR1;
    SET TOTAL1 = TOTAL1 + VAR1;
    SET VAR1 = 0;  /* Reset so the last value fetched is not added after AT_END */
  END WHILE;
  SET AT_END = 0;  /* Reset for next loop */
  WHILE AT_END = 0 DO
    FETCH RSCUR2 INTO VAR2;
    SET TOTAL2 = TOTAL2 + VAR2;
    SET VAR2 = 0;  /* Reset so the last value fetched is not added after AT_END */
  END WHILE;
END IF;

Related concepts
Example programs that call stored procedures
Examples can be used as models when you write applications that call stored procedures. In addition, prefix.SDSNSAMP contains sample jobs DSNTEJ6P and DSNTEJ6S and programs DSN8EP1 and DSN8EP2, which you can run.

**Related reference**
- ALLOCATE CURSOR (Db2 SQL)
- ASSOCIATE LOCATORS (Db2 SQL)
- CALL (Db2 SQL)
- DESCRIBE CURSOR (Db2 SQL)
- DESCRIBE PROCEDURE (Db2 SQL)
- SQL descriptor area (SQLDA) (Db2 SQL)
Chapter 6. Coding methods for distributed data

You can access distributed data by using three-part table names or explicit connect statements.

**Introductory concepts**

- Distributed data (Introduction to Db2 for z/OS)
- Effects of distributed data on programming (Introduction to Db2 for z/OS)
- Distributed data access (Introduction to Db2 for z/OS)

Three-part table names are described in “Accessing distributed data by using three-part table names” on page 773. Explicit connect statements are described in “Accessing distributed data by using explicit CONNECT statements” on page 775.

These two methods of coding applications for distributed access are illustrated by the following example.

Spiffy Computer has a main project table that supplies information about all projects that are currently active throughout the company. Spiffy has several branches in various locations around the world, each a Db2 location that maintains a copy of the project table named DSN8C10.PROJ. The main branch location occasionally inserts data into all copies of the table. The application that makes the inserts uses a table of location names. For each row that is inserted, the application executes an INSERT statement in DSN8C10.PROJ for each location.

**Copying a table from a remote location**

To copy a table from one location to another, you can either write your own application program or use the Db2 DataPropagator product.

**Related concepts**
- Monitoring Db2 in distributed environments (Db2 Performance)

**Related tasks**
- Improving performance for applications that access distributed data (Db2 Performance)

### Accessing distributed data by using three-part table names

You can use three-part table names to access data at a remote location through DRDA access.

When you use three-part table names, you must create copies of the package that you used at the local site at all possible remote locations that could be accessed by the three-part table name references. You must also explicitly or generically specify remote packages in the PKLIST of the PLAN that is used by the application.

**Recommendation:** Always use an alias, which resolves to a three-part table name, rather than specifying a specific three-part table name in an SQL statement. Using an alias will permit you to physically move the location of the table as needed. By using an alias, you can drop and re-create the alias by specifying the table’s new remote location and then rebind the packages of the application.

In a three-part table name, the first part denotes the location. The local Db2 makes and breaks an implicit connection to a remote server as needed.

When a three-part name is parsed and forwarded to a remote location, any special register settings are automatically propagated to remote server. This allows the SQL statements to process the same way no matter at what site a statement is run.

**Example**

The following example assumes that all systems involved implement two-phase commit. This example suggests updating several systems in a loop and ending the unit of work by committing only when the loop is complete. Updates are coordinated across the entire set of systems.

© Copyright IBM Corp. 1983, 2022
Spiffy's application uses a location name to construct a three-part table name in an INSERT statement. It then prepares the statement and executes it dynamically. The values to be inserted are transmitted to the remote location and substituted for the parameter markers in the INSERT statement.

The following overview shows how the application uses aliases for three-part names:

<table>
<thead>
<tr>
<th>Read in the alias values</th>
</tr>
</thead>
<tbody>
<tr>
<td>Do for all locations</td>
</tr>
<tr>
<td>Read location name</td>
</tr>
<tr>
<td>Set up statement to prepare</td>
</tr>
<tr>
<td>Prepare statement</td>
</tr>
<tr>
<td>Execute statement</td>
</tr>
<tr>
<td>End loop</td>
</tr>
<tr>
<td>Commit</td>
</tr>
</tbody>
</table>

After the application obtains the next alias of a remote table to be inserted, for example, REGION1PROJ (which is the DSN8C10.PROJ table at location SAN_JOSE), it creates the following character string:

```
INSERT INTO REGION1PROJ VALUES (?,?,?,?,?,?,?,?)
```

The alias is created as follows:

```
CREATE ALIAS REGION1PROJ FOR SAN_JOSE.DSN8C10.PROJ
```

The application assigns the character string to the variable INSERTX and then executes these statements:

```
EXEC SQL
  PREPARE STMT1 FROM :INSERTX;
EXEC SQL
  EXECUTE STMT1 USING :PROJNO, :PROJNAME, :DEPTNO, :RESPEMP,
                      :PRSTAFF, :PRSTDATE, :PRENDATE, :MAJPROJ;
```

The host variables for Spiffy's project table match the declaration for the sample project table.

To keep the data consistent at all locations, the application commits the work only when the loop has executed for all locations. Either every location has committed the INSERT or, if a failure has prevented any location from inserting, all other locations have rolled back the INSERT. (If a failure occurs during the commit process, the entire unit of work can be indoubt.)

**Three-part names and multiple servers**

**Recommendation:** Always use an asterisk (*) for the location name in a pklist. Never use the explicit location name unless you are sure that no other location could ever be accessed.

The following steps are recommended:

1. Bind the DBRM into a package at the local Db2.
2. Bind package copy at the first target site of the alias.
3. Bind package copy at the target site.

**Related concepts**
Considerations for binding packages at a remote location
When you bind packages at a remote location, you need to understand how the behavior of the remote packages differs from the behavior of local packages.

**Aliases (Db2 SQL)**
**Synonyms (deprecated) (Db2 SQL)**

**Related tasks**
Including dynamic SQL in your program
Dynamic SQL is prepared and executed while the program is running.

**Related reference**
Project table (DSN8C10.PROJ) (Introduction to Db2 for z/OS)
Accessing remote declared temporary tables by using three-part table names

You can access a remote declared temporary table by using a three-part name. However, if you combine explicit CONNECT statements and three-part names in your application, a reference to a remote declared temporary table must be a forward reference.

In a CREATE GLOBAL TEMPORARY TABLE or DECLARE GLOBAL TEMPORARY TABLE statement, you cannot specify an alias that resolves to a three-part name object at a remote location. You also cannot specify a three-part name object even if the location of the three-part name refers to the location where the object is being created or declared.

Example

You can perform the following series of actions, which includes a forward reference to a declared temporary table:

```sql
EXEC SQL CONNECT TO CHICAGO; /* Connect to the remote site */
EXEC SQL DECLARE GLOBAL TEMPORARY TABLE T1 /* Define the temporary table */
                    (CHARCOL CHAR(6) NOT NULL) /* at the remote site */
                     ON COMMIT DROP TABLE;
EXEC SQL CONNECT RESET; /* Connect back to local site */
EXEC SQL INSERT INTO CHICAGO.SESSION.T1 (VALUES 'ABCDEF'); /* Access the temporary table */
                /* at the remote site (forward reference) */
```

However, you cannot perform the following series of actions, which includes a backward reference to the declared temporary table:

```sql
EXEC SQL DECLARE GLOBAL TEMPORARY TABLE T1 /* Define the temporary table */
                    (CHARCOL CHAR(6) NOT NULL) /* at the local site (ATLANTA) */
                     ON COMMIT DROP TABLE;
EXEC SQL CONNECT TO CHICAGO; /* Connect to the remote site */
EXEC SQL INSERT INTO ATLANTA.SESSION.T1 (VALUES 'ABCDEF'); /* Cannot access temp table */
                /* from the remote site (backward reference) */
```

Example using an alias

You can perform the following series of actions, which includes a forward reference to a declared temporary table using an alias. First you need to declare the alias at the requester. The name you give the alias must resolve to match the real name.

```sql
CREATE APPLT1 FOR CHICAGO.SESSION.T1
```

The CONNECT and DECLARE statements refer to the real declared temp table.

```sql
EXEC SQL CONNECT TO CHICAGO;
EXEC SQL DECLARE GLOBAL TEMPORARY TABLE T1
                    (CHARCOL CHAR(6) NOT NULL)
                     ON COMMIT DROP TABLE;
EXEC SQL CONNECT RESET;
EXEC SQL INSERT INTO APPLT1 VALUES ('ABCDEF');
```

Accessing distributed data by using explicit CONNECT statements

When you use explicit CONNECT statements to access distributed data, the application program explicitly connects to each new server.

About this task

You must bind the DBRMs for the SQL statements to be executed at the server to packages that reside at that server.
The following example assumes that all systems involved implement two-phase commit. This example suggests updating several systems in a loop and ending the unit of work by committing only when the loop is complete. Updates are coordinated across the entire set of systems.

In this example, Spiffy's application executes CONNECT for each server in turn, and the server executes INSERT. In this case, the tables to be updated each have the same name, although each table is defined at a different server. The application executes the statements in a loop, with one iteration for each server.

The application connects to each new server by means of a host variable in the CONNECT statement. CONNECT changes the special register CURRENT SERVER to show the location of the new server. The values to insert in the table are transmitted to a location as input host variables.

The following overview shows how the application uses explicit CONNECTs:

<table>
<thead>
<tr>
<th>Read input values</th>
<th>Do for all locations</th>
<th>Read location name</th>
<th>Connect to location</th>
<th>Execute insert statement</th>
<th>End loop</th>
<th>Commit</th>
<th>Release all</th>
</tr>
</thead>
</table>

For example, the application inserts a new location name into the variable LOCATION_NAME and executes the following statements:

```
EXEC SQL
  CONNECT TO :LOCATION_NAME;
EXEC SQL
  INSERT INTO DSN8C10.PROJ VALUES (:PROJNO, :PROJNAME, :DEPTNO, :RESPEMP,
                                :PRSTAFF, :PRSTDATE, :PRENDATE, :MAJPROJ);
```

To keep the data consistent at all locations, the application commits the work only when the loop has executed for all locations. Either every location has committed the INSERT or, if a failure has prevented any location from inserting, all other locations have rolled back the INSERT. (If a failure occurs during the commit process, the entire unit of work can be indoubt.)

The host variables for Spiffy's project table match the declaration for the sample project table. LOCATION_NAME is a character-string variable of length 16.

Related reference
Project table (DSN8C10.PROJ) (Introduction to Db2 for z/OS)

Specifying a location alias name for multiple sites

You can override the location name that an application uses to access a server.

About this task
Db2 uses the DBALIAS value in the SYSIBM.LOCATIONS table to override the location name that an application uses to access a server.

For example, suppose that an employee database is deployed across two sites and that both sites make themselves known as location name EMPLOYEE. To access each site, insert a row for each site into SYSIBM.LOCATIONS with the location names SVL_EMPLOYEE and SJ_EMPLOYEE. Both rows contain EMPLOYEE as the DBALIAS value. When an application issues a CONNECT TO SVL_EMPLOYEE statement, Db2 searches the SYSIBM.LOCATIONS table to retrieve the location and network attributes of the database server. Because the DBALIAS value is not blank, Db2 uses the alias EMPLOYEE, and not the location name, to access the database.

If the application uses fully qualified object names in its SQL statements, Db2 sends the statements to the remote server without modification. For example, suppose that the application issues the statement SELECT * FROM SVL_EMPLOYEE.authid.table with the fully-qualified object name. However, Db2 accesses the remote server by using the EMPLOYEE alias. The remote server must identify itself as both SVL_EMPLOYEE and EMPLOYEE; otherwise, it rejects the SQL statement with a message indicating
that the database is not found. If the remote server is Db2, the location SVL_EMPLOYEE might be defined
as a location alias for EMPLOYEE. Db2 z/OS servers are defined with this alias by using the DDF ALIAS
statement of the DSNJU003 change log inventory utility. Db2 locally executes any SQL statements that
contain fully qualified object names if the high-level qualifier is the location name or any of its alias
names.

Related reference
LOCATIONS catalog table (Db2 SQL)
DSNJU003 (change log inventory) (Db2 Utilities)

Releasing connections

When you connect to remote locations explicitly, you must also terminate those connections explicitly.

About this task

To break the connections, you can use the RELEASE statement. The RELEASE statement differs from the
CONNECT statement in the following ways:

• While the CONNECT statement makes an immediate connection, the RELEASE statement does not
immediately break a connection. The RELEASE statement labels connections for release at the next
commit point. A connection that has been labeled for release is in the release-pending state and can still
be used before the next commit point.

• While the CONNECT statement connects to exactly one remote system, you can use the RELEASE
statement to specify a single connection or a set of connections for release at the next commit point.

Example

By using the RELEASE statement, you can place any of the following connections in the release-pending
state:

• A specific connection that the next unit of work does not use:

  EXEC SQL RELEASE SPIFFY1;

• The current SQL connection, whatever its location name:

  EXEC SQL RELEASE CURRENT;

• All connections except the local connection:

  EXEC SQL RELEASE ALL;

Transmitting mixed data

Mixed data is data that contains both character and graphic data.

About this task

If you transmit mixed data between your local system and a remote system, put the data in varying-length
character strings instead of fixed-length character strings.

Converting mixed data: When ASCII MIXED data or Unicode MIXED data is converted to EBCDIC MIXED,
the converted string is longer than the source string. An error occurs if that conversion is performed on
a fixed-length input host variable. The remedy is to use a varying-length string variable with a maximum
length that is sufficient to contain the expansion.
Identifying the server at run time

You can request the location name of the system to which you are connected.

About this task

The special register CURRENT SERVER contains the location name of the system you are connected to. You can assign that name to a host variable with a statement like this:

```sql
EXEC SQL SET :CS = CURRENT SERVER;
```

SQL limitations at dissimilar servers

When you execute SQL statements on a remote server that is running another Db2 family product, certain limitations exist. Generally, a program that uses DRDA access can use SQL statements and clauses that are supported by a remote server, even if they are not supported by the local server.

The following examples suggest what to expect from dissimilar servers:

- They support SELECT, INSERT, UPDATE, DELETE, DECLARE CURSOR, and FETCH, but details vary.

  **Example:** Db2 for Linux, UNIX, and Windows and DB2 for i support a form of INSERT that allows for multiple rows of input data. In this case, the VALUES clause is followed by multiple lists in parentheses. Each list represents the values to be inserted for a row of data. Db2 for z/OS does not support this form of INSERT.

- Data definition statements vary more widely.

  **Example:** Db2 for z/OS supports ROWID columns; Db2 for Linux, UNIX, and Windows does not support ROWID columns. Any data definition statements that use ROWID columns cannot run across all platforms.

- Statements can have different limits.

  **Example:** A query in Db2 for z/OS can have 750 columns; for other systems, the maximum is higher. But a query using 750 or fewer columns could execute in all systems.

- Some statements are not sent to the server but are processed completely by the requester. You cannot use those statements in a remote package even though the server supports them.

- In general, if a statement to be executed at a remote server contains host variables, a Db2 requester assumes them to be input host variables unless it supports the syntax of the statement and can determine otherwise. If the assumption is not valid, the server rejects the statement.

Related reference

[Characteristics of SQL statements in Db2 for z/OS (Db2 SQL)](https://www.ibm.com/support/knowledgecenter/SSDTHD_12.1.0/com.ibm.db2.luw.sql.sql05/stat01.sql001 exporting-db2-server-name)

Support for executing long SQL statements in a distributed environment

A distributed application can send prepared SQL statements exceed 32 KB in size. If the statements exceed 32 KB in size, the server must support these long statements.

If a distributed application assigns an SQL statement to a DBCLOB (UTF-16) variable and sends the prepared statement to a remote server, the remote Db2 server converts it to UTF-8. If the remote server does not support UTF-8, the requester converts the statement to the system EBCDIC CCSID before sending it to the remote server.
Distributed queries against ASCII or Unicode tables

When you perform a distributed query, the server determines the encoding scheme of the result table. When a distributed query against an ASCII or Unicode table arrives at the Db2 for z/OS server, the server indicates in the reply message that the columns of the result table contain ASCII or Unicode data, rather than EBCDIC data. The reply message also includes the CCSIDs of the data to be returned. The CCSID of data from a column is the CCSID that was in effect when the column was defined.

The encoding scheme in which Db2 returns data depends on two factors:

- The encoding scheme of the requesting system.
  - If the requester is ASCII or Unicode, the returned data is ASCII or Unicode. If the requester is EBCDIC, the returned data is EBCDIC, even though it is stored at the server as ASCII or Unicode. However, if the SELECT statement that is used to retrieve the data contains an ORDER BY clause, the data displays in ASCII or Unicode order.

- Whether the application program overrides the CCSID for the returned data. The ways to do this are as follows:
  - For static SQL
    You can bind a plan or package with the ENCODING bind option to control the CCSIDs for all static data in that plan or package. For example, if you specify ENCODING(UNICODE) when you bind a package at a remote Db2 for z/OS system, the data that is returned in host variables from the remote system is encoded in the default Unicode CCSID for that system.
  - For static or dynamic SQL
    An application program can specify overriding CCSIDs for individual host variables in DECLARE VARIABLE statements. An application program that uses an SQLDA can specify an overriding CCSID for the returned data in the SQLDA. When the application program executes a FETCH statement, you receive the data in the CCSID that is specified in the SQLDA.

Related tasks
Setting the CCSID for host variables
All Db2 string data, other than binary data, has an encoding scheme and a coded character set ID (CCSID) associated with it. You can associate an encoding scheme and a CCSID with individual host variables. Any data in those host variable is then associated with that encoding scheme and CCSID.

Related reference
BIND and REBIND options for packages, plans, and services (Db2 Commands)

Restrictions when using scrollable cursors to access distributed data

The restrictions that exist for scrollable cursors depend on what the requester and the server support. If a Db2 for z/OS server processes an OPEN cursor statement for a scrollable cursor, and the OPEN cursor statement comes from a requester that does not support scrollable cursors, the Db2 for z/OS server returns an SQL error. However, if a stored procedure at the server uses a scrollable cursor to return a result set, the down-level requester can access data through that cursor. The Db2 for z/OS server converts the scrollable result set cursor to a non-scrollable cursor. The requester can retrieve the data using sequential FETCH statements.
Restrictions when using rowset-positioned cursors to access distributed data

The restrictions that exist for row-positioned cursors depend on what the requester and the server support.

If a Db2 for z/OS server processes an OPEN cursor statement for a rowset-positioned cursor, and the OPEN cursor statement comes from a requester that does not support rowset-positioned cursors, the Db2 for z/OS server returns an SQL error. However, if a stored procedure at the server uses a rowset-positioned cursor to return a result set, the down-level requester can access data through that cursor by using row-positioned FETCH statements.

IBM MQ with Db2

IBM MQ is a message handling system that enables applications to communicate in a distributed environment across different operating systems and networks.

IBM MQ handles the communication from one program to another by using application programming interfaces (APIs). You can use any of the following APIs to interact with the IBM MQ message handling system:

• Message Queue Interface (MQI)
• IBM MQ classes for Java
• IBM MQ classes for Java Message Service (JMS)

Db2 provides its own application programming interface to the WebSphere® MQ message handling system through a set of external user-defined functions, which are called Db2 MQ functions. You can use these functions in SQL statements to combine Db2 database access with IBM MQ message handling. The Db2 MQ functions use the MQI.

Related reference
The Message Queue Interface overview

IBM MQ messages

IBM MQ uses messages to pass information between applications.

Messages consist of the following parts:

• The message attributes, which identify the message and its properties.
• The message data, which is the application data that is carried in the message.

Related concepts
Db2 MQ functions and Db2 MQ XML stored procedures
You can use the Db2 MQ functions and stored procedures to send messages to a message queue or to receive messages from the message queue.

IBM MQ message handling

Conceptually, the IBM MQ message handling system takes a piece of information (the message) and sends it to its destination. MQ guarantees delivery, despite any network disruptions that might occur.

In IBM MQ, a destination is called a message queue, and a queue resides in a queue manager. Applications can put messages on queues or get messages from them.

Db2 communicates with the WebSphere message handling system through a set of external user-defined functions, which are called Db2 MQ functions. These functions use the MQI.

When you send a message, you must specify the following three components:

message data
Defines what is sent from one program to another.
service
Defines where the message is going to or coming from. The parameters for managing a queue are defined in the service, which is typically defined by a system administrator. The complexity of the parameters in the service is hidden from the application program.

policy
Defines how the message is handled. Policies control such items as:

- The attributes of the message, for example, the priority.
- Options for send and receive operations, for example, whether an operation is part of a unit of work.

The default service and policy are set as part of defining the WebSphere MQ configuration for a particular installation of Db2. (This action is typically performed by a system administrator.) Db2 provides the default service Db2.DEFAULT.SERVICE and the default policy Db2.DEFAULT.POLICY.

Related tasks
Additional steps for enabling IBM MQ user-defined functions (Db2 Installation and Migration)

Related reference
IBM MQ home

IBM MQ message handling with the MQI
One way to send and receive IBM MQ messages from Db2 applications is to use the Db2 MQ functions that use MQI.

These MQI-based functions use the services and policies that are defined in two Db2 tables, SYSIBM.MQSERVICE_TABLE and SYSIBM.MQPOLICY_TABLE. These tables are user-managed and are typically created and maintained by a system administrator. Each table contains a row for the default service and policy that are provided by Db2.

The application program does not need to know the details of the services and policies that are defined in these tables. The application needs only specify which service and policy to use for each message that it sends and receives. The application specifies this information when it calls a Db2 MQ function.

Related concepts
Db2 MQ functions and Db2 MQ XML stored procedures
You can use the Db2 MQ functions and stored procedures to send messages to a message queue or to receive messages from the message queue.

Related reference
Db2 MQ tables
The Db2 MQ tables contain service and policy definitions that are used by the Message Queue Interface (MQI) based Db2 MQ functions. You must populate the Db2 MQ tables before you can use these MQI-based functions.

Db2 MQI services
A service describes a destination to which an application sends messages or from which an application receives messages. Db2 Message Queue Interface (MQI) services are defined in the Db2 table SYSIBM.MQSERVICE_TABLE.

The MQI-based Db2 MQ functions use the services that are defined in the Db2 table SYSIBM.MQSERVICE_TABLE. This table is user-managed and is typically created and maintained by a system administrator. This table contains a row for each defined service, including your customized services and the default service that is provided by Db2.

The application program does not need to know the details of the defined services. When an application program calls an MQI-based Db2 MQ function, the program selects a service from SYSIBM.MQSERVICE_TABLE by specifying it as a parameter.

Related concepts
Db2 MQ functions and Db2 MQ XML stored procedures
You can use the Db2 MQ functions and stored procedures to send messages to a message queue or to receive messages from the message queue.

**IBM MQ message handling**
Conceptually, the IBM MQ message handling system takes a piece of information (the message) and sends it to its destination. MQ guarantees delivery, despite any network disruptions that might occur.

**Related reference**

Db2 MQ tables
The Db2 MQ tables contain service and policy definitions that are used by the Message Queue Interface (MQI) based Db2 MQ functions. You must populate the Db2 MQ tables before you can use these MQI-based functions.

Db2 MQI policies
A policy controls how the MQ messages are handled. Db2 Message Queue Interface (MQI) policies are defined in the Db2 table SYSIBM.MQPOLICY_TABLE.

The MQI-based Db2 MQ functions use the policies that are defined in the Db2 table SYSIBM.MQPOLICY_TABLE. This table is user-managed and is typically created and maintained by a system administrator. This table contains a row for each defined policy, including your customized policies and the default policy that is provided by Db2.

The application program does not need to know the details of the defined policies. When an application program calls an MQI-based Db2 MQ function, the program selects a policy from SYSIBM.MQPOLICY_TABLE by specifying it as a parameter.

**Related concepts**

Db2 MQ functions and Db2 MQ XML stored procedures
You can use the Db2 MQ functions and stored procedures to send messages to a message queue or to receive messages from the message queue.

**IBM MQ message handling**
Conceptually, the IBM MQ message handling system takes a piece of information (the message) and sends it to its destination. MQ guarantees delivery, despite any network disruptions that might occur.

**Related reference**

Db2 MQ tables
The Db2 MQ tables contain service and policy definitions that are used by the Message Queue Interface (MQI) based Db2 MQ functions. You must populate the Db2 MQ tables before you can use these MQI-based functions.

**Db2 MQ functions and Db2 MQ XML stored procedures**

You can use the Db2 MQ functions and stored procedures to send messages to a message queue or to receive messages from the message queue.

The Db2 MQ functions support the following types of operations:

- **Send and forget**, where no reply is needed.
- **Read or receive**, where one or all messages are either read without removing them from the queue, or received and removed from the queue.
- **Request and response**, where a sending application needs a response to a request.
- **Publish and subscribe**, where messages are assigned to specific publisher services and are sent to queues. Applications that subscribe to the corresponding subscriber service can monitor specific messages.

You can use the Db2 MQ functions and stored procedures to send messages to a message queue or to receive messages from the message queue.

You can send a request to a message queue and receive a response, and you can also publish messages to the IBM MQ publisher and subscribe to messages that have been published with specific topics. The Db2 MQ XML functions and stored procedures enable you to query XML documents and then publish the results to a message queue.
The Db2 MQ functions include scalar functions, table functions, and XML-specific functions. For each of these functions, you can call a version that uses the MQI. The function signatures are the same. However, the qualifying schema names are different. To call an MQI-based function, specify the schema name DB2MQ.

**Requirement:** Before you can call the version of these functions that uses MQI, you need to populate the Db2 MQ tables.

The following table describes the Db2 MQ scalar functions.

### Table 125. Db2 MQ scalar functions

<table>
<thead>
<tr>
<th>Scalar function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MQREAD (receive-service, service-policy)</td>
<td>MQREAD returns a message in a VARCHAR variable from the MQ location specified by receive-service, using the policy defined in service-policy. This operation does not remove the message from the head of the queue but instead returns it. If no messages are available to be returned, a null value is returned.</td>
</tr>
<tr>
<td>MQREADCLOB (receive-service, service-policy)</td>
<td>MQREADCLOB returns a message in a CLOB variable from the MQ location specified by receive-service, using the policy defined in service-policy. This operation does not remove the message from the head of the queue but instead returns it. If no messages are available to be returned, a null value is returned.</td>
</tr>
<tr>
<td>MQRECEIVE (receive-service, service-policy, correlation-id)</td>
<td>MQRECEIVE returns a message in a VARCHAR variable from the MQ location specified by receive-service, using the policy defined in service-policy. This operation removes the message from the queue. If correlation-id is specified, the first message with a matching correlation identifier is returned; if correlation-id is not specified, the message at the beginning of queue is returned. If no messages are available to be returned, a null value is returned.</td>
</tr>
<tr>
<td>MQRECEIVECLOB (receive-service, service-policy, correlation-id)</td>
<td>MQRECEIVECLOB returns a message in a CLOB variable from the MQ location specified by receive-service, using the policy defined in service-policy. This operation removes the message from the queue. If correlation-id is specified, the first message with a matching correlation identifier is returned; if correlation-id is not specified, the message at the head of queue is returned. If no messages are available to be returned, a null value is returned.</td>
</tr>
<tr>
<td>MQSEND (send-service, service-policy, msg-data, correlation-id)</td>
<td>MQSEND sends the data in a VARCHAR or CLOB variable msg-data to the MQ location specified by send-service, using the policy defined in service-policy. An optional user-defined message correlation identifier can be specified by correlation-id. The return value is 1 if successful or 0 if not successful.</td>
</tr>
</tbody>
</table>

**Notes:**

1. You can send or receive messages in VARCHAR variables or CLOB variables. The maximum length for a message in a VARCHAR variable is 32 KB. The maximum length for a message in a CLOB variable is 2 MB.

The following table describes the MQ table functions that Db2 can use.
### Table 126. Db2 MQ table functions

<table>
<thead>
<tr>
<th>Table function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MQREADALL (receive-service, service-policy, num-rows)</td>
<td>MQREADALL returns a table that contains the messages and message metadata in VARCHAR variables from the MQ location specified by receive-service, using the policy defined in service-policy. This operation does not remove the messages from the queue. If num-rows is specified, a maximum of num-rows messages is returned; if num-rows is not specified, all available messages are returned.</td>
</tr>
<tr>
<td>MQREADALLCLOB (receive-service, service-policy, num-rows)</td>
<td>MQREADALLCLOB returns a table that contains the messages and message metadata in CLOB variables from the MQ location specified by receive-service, using the policy defined in service-policy. This operation does not remove the messages from the queue. If num-rows is specified, a maximum of num-rows messages is returned; if num-rows is not specified, all available messages are returned.</td>
</tr>
<tr>
<td>MQRECEIVEALL (receive-service, service-policy, correlation-id, num-rows)</td>
<td>MQRECEIVEALL returns a table that contains the messages and message metadata in VARCHAR variables from the MQ location specified by receive-service, using the policy defined in service-policy. This operation removes the messages from the queue. If correlation-id is specified, only those messages with a matching correlation identifier are returned; if correlation-id is not specified, all available messages are returned. If num-rows is specified, a maximum of num-rows messages is returned; if num-rows is not specified, all available messages are returned.</td>
</tr>
<tr>
<td>MQRECEIVEALLCLOB (receive-service, service-policy, correlation-id, num-rows)</td>
<td>MQRECEIVEALLCLOB returns a table that contains the messages and message metadata in CLOB variables from the MQ location specified by receive-service, using the policy defined in service-policy. This operation removes the messages from the queue. If correlation-id is specified, only those messages with a matching correlation identifier are returned; if correlation-id is not specified, all available messages are returned. If num-rows is specified, a maximum of num-rows messages is returned; if num-rows is not specified, all available messages are returned.</td>
</tr>
</tbody>
</table>

**Notes:**

1. You can send or receive messages in VARCHAR variables or CLOB variables. The maximum length for a message in a VARCHAR variable is 32 KB. The maximum length for a message in a CLOB variable is 2 MB.
2. The first column of the result table of a Db2 MQ table function contains the message.

**Related tasks**

Additional steps for enabling IBM MQ user-defined functions (Db2 Installation and Migration)

**Related reference**

Procedures that are supplied with Db2 (Db2 SQL)
MQREADALL (Db2 SQL)
MQREADALLCLOB (Db2 SQL)
MQRECEIVEALL (Db2 SQL)
MQRECEIVEALLCLOB (Db2 SQL)
The Message Queue Interface overview
Generating XML documents from existing tables and sending them to an MQ message queue

You can send data from a Db2 table to the MQ message queue. First put the data in an XML document and then send that document to the message queue.

Procedure

To generate XML documents from existing tables and send them to an MQ message queue:
1. Compose an XML document by using the Db2 XML publishing functions.
2. Cast the XML document to type VARCHAR or CLOB.
3. Send the document to an MQ message queue by using the appropriate Db2 MQ function.

Related concepts

Db2 MQ functions and Db2 MQ XML stored procedures
You can use the Db2 MQ functions and stored procedures to send messages to a message queue or to receive messages from the message queue.

Functions for constructing XML values (Db2 Programming for XML)

Shredding XML documents from an MQ message queue

When you retrieve XML data from an MQ message queue, you can shred that data into Db2 tables for easy retrievability.

About this task

Procedure

To shred XML documents from an MQ message queue:
1. Retrieve the XML document from an MQ message queue by using the appropriate MQ function.
2. Shred the retrieved message to Db2 tables by using the XML decomposition stored procedure (XDBDECOMPXML).

Related concepts

Db2 MQ functions and Db2 MQ XML stored procedures
You can use the Db2 MQ functions and stored procedures to send messages to a message queue or to receive messages from the message queue.

Db2 MQ tables

The Db2 MQ tables contain service and policy definitions that are used by the Message Queue Interface (MQI) based Db2 MQ functions. You must populate the Db2 MQ tables before you can use these MQI-based functions.

The Db2 MQ tables are SYSIBM.MQSERVICE_TABLE and SYSIBM.MQPOLICY_TABLE. These tables are user-managed. You need to create them during the installation or migration process. Installation job DSNTIJRT creates these tables with one default row in each table.

If you previously used the AMI-based Db2 MQ functions, you used AMI configuration files instead of these tables. To use the MQI-based Db2 MQ functions, you need to move the data from those configuration files to the Db2 tables SYSIBM.MQSERVICE_TABLE and SYSIBM.MQPOLICY_TABLE.

The following table describes the columns for SYSIBM.MQSERVICE_TABLE.
Table 127. SYSIBM.MQSERVICE_TABLE column descriptions

<table>
<thead>
<tr>
<th>Column name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SERVICENAME</td>
<td>This column contains the service name, which is an optional input parameter of the MQ functions. This column is the primary key for the SYSIBM.MQSERVICE_TABLE table.</td>
</tr>
<tr>
<td>QUEUEMANAGER</td>
<td>This column contains the name of the queue manager where the MQ functions are to establish a connection.</td>
</tr>
<tr>
<td>INPUTQUEUE</td>
<td>This column contains the name of the queue from which the MQ functions are to send and retrieve messages.</td>
</tr>
<tr>
<td>CODEDCHARSETID</td>
<td>This column contains the character set identifier for character data in the messages that are sent and received by the MQ functions. This column corresponds to the CodedCharSetId field in the message descriptor structure (MQMD). MQ functions use the value in this column to set the CodedCharSetId field. The default value for this column is 0, which sets the CodedCharSetId field of the MQMD to the value MQCCSI_Q_MGR.</td>
</tr>
<tr>
<td>ENCODING</td>
<td>This column contains the encoding value for the numeric data in the messages that are sent and received by the MQ functions. This column corresponds to the Encoding field in the message descriptor structure (MQMD). MQ functions use the value in this column to set the Encoding field. The default value for this column is 0, which sets the Encoding field in the MQMD to the value MQENC_NATIVE.</td>
</tr>
<tr>
<td>DESCRIPTION</td>
<td>This column contains the description of the service.</td>
</tr>
</tbody>
</table>

The following table describes the columns for SYSIBM.MQPOLICY_TABLE.

Table 128. SYSIBM.MQPOLICY_TABLE column descriptions

<table>
<thead>
<tr>
<th>Column name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>POLICYNAME</td>
<td>This column contains the policy name, which is an optional input parameter of the MQ functions. This column is the primary key for the SYSIBM.MQPOLICY_TABLE table.</td>
</tr>
</tbody>
</table>
### Table 128. SYSIBM.MQPOLICY_TABLE column descriptions (continued)

<table>
<thead>
<tr>
<th>Column name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SEND_PRIORITY</td>
<td>This column contains the priority of the message. This column corresponds to the Priority field in the message descriptor structure (MQMD). MQ functions use the value in this column to set the Priority field. The default value for this column is -1, which sets the Priority field in the MQMD to the value MQQPRI_PRIORITY_AS_Q_DEF.</td>
</tr>
</tbody>
</table>
| SEND_PERSISTENCE     | This column indicates whether the message persists despite any system failures or instances of restarting the queue manager. This column corresponds to the Persistence field in the message descriptor structure (MQMD). MQ functions use the value in this column to set the Persistence field. This column can have the following values:  
  - Q: Sets the Persistence field in the MQMD to the value MQPER_PERSISTENCE_AS_Q_DEF. This value is the default.  
  - Y: Sets the Persistence field in the MQMD to the value MQPER_PERSISTENT.  
  - N: Sets the Persistence field in the MQMD to the value MQPER_NOT_PERSISTENT. |
| SEND_EXPIRY          | This column contains the message expiration time, in tenths of a second. This column corresponds to the Expiry field in the message descriptor structure (MQMD). MQ functions use the value in this column to set the Expiry field. The default value is -1, which sets the Expiry field to the value MQEI_UNLIMITED. |
| SEND_RETRY_COUNT     | This column contains the number of times that the MQ function is to try to send a message if the procedure fails. The default value is 5. |
| SEND_RETRY_INTERVAL  | This column contains the interval, in milliseconds, between each attempt to send a message. The default value is 1000. |
### Table 128. SYSIBM.MQPOLICY_TABLE column descriptions (continued)

<table>
<thead>
<tr>
<th>Column name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SEND_NEW_CORRELID</td>
<td>This column specifies how the correlation identifier is to be set if a correlation identifier is not passed as an input parameter in the MQ function. The correlation identifier is set in the CorrelId field in the message descriptor structure (MQMD). This column can have one of the following values:</td>
</tr>
<tr>
<td></td>
<td><strong>N</strong> Sets the CorrelId field in the MQMD to binary zeros. This value is the default.</td>
</tr>
<tr>
<td></td>
<td><strong>Y</strong> Specifies that the queue manager is to generate a new correlation identifier and set the CorrelId field in the MQMD to that value. This 'Y' value is equivalent to setting the MQPMO_NEW_CORREL_ID option in the Options field in the put message options structure (MQPMO).</td>
</tr>
<tr>
<td>SEND_RESPONSE_MSGID</td>
<td>This column specifies how the MsgId field in the message descriptor structure (MQMD) is to be set for report and reply messages. This column corresponds to the Report field in the MQMD. MQ functions use the value in this column to set the Report field. This column can have one of the following values:</td>
</tr>
<tr>
<td></td>
<td><strong>N</strong> Sets the MQRO_NEW_MSG_ID option in the Report field in the MQMD. This value is the default.</td>
</tr>
<tr>
<td></td>
<td><strong>P</strong> Sets the MQRO_PASS_MSG_ID option in the Report field in the MQMD.</td>
</tr>
<tr>
<td>SEND_RESPONSE_CORRELID</td>
<td>This column specifies how the CorrelID field in the message descriptor structure (MQMD) is to be set for report and reply messages. This column corresponds to the Report field in the MQMD. MQ functions use the value in this column to set the Report field. This column can have one of the following values:</td>
</tr>
<tr>
<td></td>
<td><strong>C</strong> Sets the MQRO_COPY_MSG_ID_TO_CORREL_ID option in the Report field in the MQMD. This value is the default.</td>
</tr>
<tr>
<td></td>
<td><strong>P</strong> Sets the MQRO_PASS_CORREL_ID option in the Report field in the MQMD.</td>
</tr>
<tr>
<td>Column name</td>
<td>Description</td>
</tr>
<tr>
<td>--------------------------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>SEND_EXCEPTION_ACTION</td>
<td>This column specifies what to do with the original message when it cannot be delivered to the destination queue.</td>
</tr>
<tr>
<td></td>
<td>This column corresponds to the Report field in the message descriptor structure (MQMD). MQ functions use the value in this column to set the Report field.</td>
</tr>
<tr>
<td></td>
<td>This column can have one of the following values:</td>
</tr>
<tr>
<td>Q</td>
<td>Sets the MQRO_DEAD_LETTER_Q option in the Report field in the MQMD. This value is the default.</td>
</tr>
<tr>
<td>D</td>
<td>Sets the MQRO_DISCARD_MSG option in the Report field in the MQMD.</td>
</tr>
<tr>
<td>P</td>
<td>Sets the MQRO_PASS_DISCARD_AND_EXPIRY option in the Report field in the MQMD.</td>
</tr>
<tr>
<td>SEND_REPORT_EXCEPTION</td>
<td>This column specifies whether an exception report message is to be generated when a message cannot be delivered to the specified destination queue and if so, what that report message should contain.</td>
</tr>
<tr>
<td></td>
<td>This column corresponds to the Report field in the message descriptor structure (MQMD). MQ functions use the value in this column to set the Report field.</td>
</tr>
<tr>
<td></td>
<td>This column can have one of the following values:</td>
</tr>
<tr>
<td>N</td>
<td>Specifies that an exception report message is not to be generated. No options in the Report field are set. This value is the default.</td>
</tr>
<tr>
<td>E</td>
<td>Sets the MQRO_EXCEPTION option in the Report field in the MQMD.</td>
</tr>
<tr>
<td>D</td>
<td>Sets the MQRO_EXCEPTION_WITH_DATA option in the Report field in the MQMD.</td>
</tr>
<tr>
<td>F</td>
<td>Sets the MQRO_EXCEPTION_WITH_FULL_DATA option in the Report field in the MQMD.</td>
</tr>
</tbody>
</table>
Table 128. SYSIBM.MQPOLICY_TABLE column descriptions (continued)

<table>
<thead>
<tr>
<th>Column name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SEND_REPORT_COA</td>
<td>This column specifies whether the queue manager is to send a confirm-on-arrival (COA) report message when the message is placed in the destination queue, and if so, what that COA message is to contain.</td>
</tr>
<tr>
<td></td>
<td>This column corresponds to the Report field in the message descriptor structure (MQMD). MQ functions use the value in this column to set the Report field.</td>
</tr>
<tr>
<td></td>
<td>This column can have one of the following values:</td>
</tr>
<tr>
<td>N</td>
<td>Specifies that a COA message is not to be sent. No options in the Report field are set. This value is the default.</td>
</tr>
<tr>
<td>C</td>
<td>Sets the MQRO_COA option in the Report field in the MQMD.</td>
</tr>
<tr>
<td>D</td>
<td>Sets the MQRO_COA_WITH_DATA option in the Report field in the MQMD.</td>
</tr>
<tr>
<td>F</td>
<td>Sets the MQRO_COA_WITH_FULL_DATA option in the Report field in the MQMD.</td>
</tr>
<tr>
<td>SEND_REPORT_COD</td>
<td>This column specifies whether the queue manager is to send a confirm-on-delivery (COD) report message when an application retrieves and deletes a message from the destination queue, and if so, what that COD message is to contain.</td>
</tr>
<tr>
<td></td>
<td>This column corresponds to the Report field in the message descriptor structure (MQMD). MQ functions use the value in this column to set the Report field.</td>
</tr>
<tr>
<td></td>
<td>This column can have one of the following values:</td>
</tr>
<tr>
<td>N</td>
<td>Specifies that a COD message is not to be sent. No options in the Report field are set. This value is the default.</td>
</tr>
<tr>
<td>C</td>
<td>Sets the MQRO_COD option in the Report field in the MQMD.</td>
</tr>
<tr>
<td>D</td>
<td>Sets the MQRO_COD_WITH_DATA option in the Report field in the MQMD.</td>
</tr>
<tr>
<td>F</td>
<td>Sets the MQRO_COD_WITH_FULL_DATA option in the Report field in the MQMD.</td>
</tr>
</tbody>
</table>
### Table 128. SYSIBM.MQPOLICY_TABLE column descriptions (continued)

<table>
<thead>
<tr>
<th>Column name</th>
<th>Description</th>
</tr>
</thead>
</table>
| SEND_REPORT_EXPIRY        | This column specifies whether the queue manager is to send an expiration report message if a message is discarded before it is delivered to an application, and if so, what that message is to contain. This column corresponds to the Report field in the message descriptor structure (MQMD). MQ functions use the value in this column to set the Report field. This column can have one of the following values:  
  **N**  
  Specifies that an expiration report message is not to be sent. No options in the Report field are set. This value is the default.  
  **C**  
  Sets the MQRO_EXPIRATION option in the Report field in the MQMD.  
  **D**  
  Sets the MQRO_EXPIRATION_WITH_DATA option in the Report field in the MQMD.  
  **F**  
  Sets the MQRO_EXPIRATION_WITH_FULL_DATA option in the Report field in the MQMD. |
| SEND_REPORT_ACTION        | This column specifies whether the receiving application sends a positive action notification (PAN), a negative action notification (NAN), or both. This column corresponds to the Report field in the message descriptor structure (MQMD). MQ functions use the value in this column to set the Report field. This column can have one of the following values:  
  **N**  
  Specifies that neither notification is to be sent. No options in the Report field are set. This value is the default.  
  **P**  
  Sets the MQRO_PAN option in the Report field in the MQMD.  
  **T**  
  Sets the MQRO_NAN option in the Report field in the MQMD.  
  **B**  
  Sets both the MQRO_PAN and MQRO_NAN options in the Report field in the MQMD. |
<table>
<thead>
<tr>
<th>Column name</th>
<th>Description</th>
</tr>
</thead>
</table>
| SEND_MSG_TYPE  | This column contains the type of message. This column corresponds to the MsqType field in the message descriptor structure (MQMD). MQ functions use the value in this column to set the MsqType field. This column can have one of the following values:  
|                | **DTG** Sets the MsqType field in the MQMD to MQMT_DATAGRAM. This value is the default.                                                   |
|                | **REQ** Sets the MsqType field in the MQMD to MQMT_REQUEST.                                                                                   |
|                | **RLY** Sets the MsqType field in the MQMD to MQMT_REPLY.                                                                                      |
|                | **RPT** Sets the MsqType field in the MQMD to MQMT_REPORT.                                                                                     |
| REPLY_TO_Q     | This column contains the name of the message queue to which the application that issued the MQGET call is to send reply and report messages.  
|                | This column corresponds to the ReplyToQ field in the message descriptor structure (MQMD). MQ functions use the value in this column to set the ReplyToQ field.  
|                | The default value for this column is SAME AS INPUT_Q, which sets the name to the queue name that is defined in the service that was used for sending the message. If no service was specified, the name is set to DB2MQ_DEFAULT_Q, which is the name of the input queue for the default service. |
| REPLY_TO_QMGR  | This column contains the name of the queue manager to which the reply and report messages are to be sent.  
|                | This column corresponds to the ReplyToQMgr field in the message descriptor structure (MQMD). MQ functions use the value in this column to set the ReplyToQMgr field.  
|                | The default value for this column is SAME AS INPUT_QMGR, which sets the name to the queue manager name that is defined in the service that was used for sending the message. If no service was specified, the name is set to the name of the queue manager for the default service. |
Table 128. SYSIBM.MQPOLICY_TABLE column descriptions (continued)

<table>
<thead>
<tr>
<th>Column name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>RCV_WAIT_INTERVAL</strong></td>
<td>This column contains the time, in milliseconds, that Db2 is to wait for messages to arrive in the queue.</td>
</tr>
<tr>
<td></td>
<td>This column corresponds to the WaitInterval field in the get message options structure (MQGMO). MQ functions use the value in this column to set the WaitInterval field.</td>
</tr>
<tr>
<td></td>
<td>The default is 10.</td>
</tr>
<tr>
<td><strong>RCV_CONVERT</strong></td>
<td>This column indicates whether to convert the application data in the message to conform to the CodedCharSetId and Encoding values of the specified MQ service.</td>
</tr>
<tr>
<td></td>
<td>This column corresponds to the Options field in the get message options structure (MQGMO). MQ functions use the value in this column to set the Options field.</td>
</tr>
<tr>
<td></td>
<td>This column can have one of the following values:</td>
</tr>
<tr>
<td><strong>Y</strong></td>
<td>Sets the MQGMO_CONVERT option in the Options field in the MQGMO. This value is the default.</td>
</tr>
<tr>
<td><strong>N</strong></td>
<td>Specifies that no data is to be converted.</td>
</tr>
<tr>
<td><strong>RCV_ACCEPT_TRUNC_MSG</strong></td>
<td>This column specifies the behavior of the MQ function when oversized messages are retrieved.</td>
</tr>
<tr>
<td></td>
<td>This column corresponds to the Options field in the get message options structure (MQGMO). MQ functions use the value in this column to set the Options field.</td>
</tr>
<tr>
<td></td>
<td>This column can have one of the following values:</td>
</tr>
<tr>
<td><strong>Y</strong></td>
<td>Sets the MQGMO_ACCEPT_TRUNCATED_MSG option in the Options field in the MQGMO. This value is the default.</td>
</tr>
<tr>
<td><strong>N</strong></td>
<td>Specifies that no messages are to be truncated. If the message is too large to fit in the buffer, the MQ function terminates with an error.</td>
</tr>
<tr>
<td><strong>Recommendation:</strong></td>
<td>Set this column to Y. In this case, if the message buffer is too small to hold the complete message, the MQ function can fill the buffer with as much of the message as the buffer can hold.</td>
</tr>
</tbody>
</table>

Chapter 6. Coding methods for distributed data 793
<table>
<thead>
<tr>
<th>Column name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>REV_OPEN_SHARED</td>
<td>This column specifies the input queue mode when messages are retrieved.</td>
</tr>
<tr>
<td></td>
<td>This column corresponds to the Options parameter for an MQOPEN call. MQ functions use the</td>
</tr>
<tr>
<td></td>
<td>value in this column to set the Options parameter.</td>
</tr>
<tr>
<td></td>
<td>This column can have one of the following values:</td>
</tr>
<tr>
<td></td>
<td>S  Sets the MQOO_INPUT_SHARED option. This value is the default.</td>
</tr>
<tr>
<td></td>
<td>E  Sets the MQ option MQOO_INPUT_EXCLUSIVE option.</td>
</tr>
<tr>
<td></td>
<td>D  Sets the MQ option MQOO_INPUT_AS_Q_DEF option.</td>
</tr>
<tr>
<td>SYNCPOINT</td>
<td>This column indicates whether the MQ function is to operate within the protocol for a normal</td>
</tr>
<tr>
<td></td>
<td>unit of work.</td>
</tr>
<tr>
<td></td>
<td>This column can have one of the following values:</td>
</tr>
<tr>
<td></td>
<td>Y  Specifies that the MQ function is to operate within the protocol for a normal unit of work.</td>
</tr>
<tr>
<td></td>
<td>N  Specifies that the MQ function is to operate outside the protocol for a normal unit of work.</td>
</tr>
<tr>
<td>DESC</td>
<td>This column contains the description of the policy.</td>
</tr>
</tbody>
</table>

**Related reference**

Core WLM environments for Db2-supplied routines (Db2 Installation and Migration)
Developing applications reference

**Basic messaging with IBM MQ**

The most basic form of messaging with the Db2 MQ functions occurs when all database applications connect to the same Db2 database server. Clients can be local to the database server or distributed in a network environment.

In a simple scenario, client A invokes the MQSEND function to send a user-defined string to the location that is defined by the default service. Db2 executes the MQ functions that perform this operation on the database server. At some later time, client B invokes the MQRECEIVE function to remove the message at the head of the queue that is defined by the default service, and return it to the client. Db2 executes the MQ functions that perform this operation on the database server.

Database clients can use simple messaging in a number of ways:

- Data collection
Information is received in the form of messages from one or more sources. An information source can be any application. The data is received from queues and stored in database tables for additional processing.

- **Workload distribution**

  Work requests are posted to a queue that is shared by multiple instances of the same application. When an application instance is ready to perform some work, it receives a message that contains a work request from the head of the queue. Multiple instances of the application can share the workload that is represented by a single queue of pooled requests.

- **Application signaling**

  In a situation where several processes collaborate, messages are often used to coordinate their efforts. These messages might contain commands or requests for work that is to be performed. For more information about this technique, see “Application to application connectivity with IBM MQ” on page 797.

The following scenario extends basic messaging to incorporate remote messaging. Assume that machine A sends a message to machine B.

1. The Db2 client executes an MQSEND function call, specifying a target service that has been defined to be a remote queue on machine B.
2. The MQ functions perform the work to send the message. The WebSphere MQ server on machine A accepts the message and guarantees that it will deliver it to the destination that is defined by the service and the current MQ configuration of machine A. The server determines that the destination is a queue on machine B. The server then attempts to deliver the message to the WebSphere MQ server on machine B, trying again as needed.
3. The IBM MQ server on machine B accepts the message from the server on machine A and places it in the destination queue on machine B.
4. A IBM MQ client on machine B requests the message at the head of the queue.

### Sending messages with IBM MQ

When you send messages with IBM MQ, you choose what data to send, where to send it and when to send it. This type of messaging is called **send and forget**; the sender sends a message and relies on IBM MQ to ensure that the message reaches its destination.

#### Procedure

To send messages with IBM MQ, use MQSEND.

Message content can be any combination of SQL statements, expressions, functions, and user-specified data. Because this MQSEND function uses two-phase commit, the COMMIT statement ensures that the message is added to the MQ queue.

#### Examples

If you send more than one column of information, separate the columns with the characters `" | ' ' | "`.  
MQSEND (LASTNAME " | ' ' | " FIRSTNAME)

The following examples use the DB2MQ schema for two-phase commit, with the default service Db2.DEFAULTSERVICE and the default policy Db2.DEFAULT.POLICY.

The following SQL SELECT statement sends a message that consists of the string "Testing msg":

```sql
SELECT DB2MQ.MQSEND ('Testing msg')
FROM SYSIBM.SYSDUMMY1;
COMMIT;
```
The MQSEND function is invoked once because SYSIBM.SYSDUMMY1 has only one row. Because this MQSEND function uses two-phase commit, the COMMIT statement ensures that the message is added to the queue.

When you use single-phase commit, you do not need to use a COMMIT statement. For example:

```sql
SELECT DB2MQ.MQSEND ('Testing msg')
FROM SYSIBM.SYSDUMMY1;
```

The MQ operation causes the message to be added to the queue.

Assume that you have an EMPLOYEE table, with VARCHAR columns LASTNAME, FIRSTNAME, and DEPARTMENT. To send a message that contains this information for each employee in DEPARTMENT 5LGA, issue the following SQL SELECT statement:

```sql
SELECT DB2MQ.MQSEND (LASTNAME || ' ' || FIRSTNAME || ' ' || DEPARTMENT)
FROM EMPLOYEE WHERE DEPARTMENT = '5lGA';
COMMIT;
```

### Related reference

**MQSEND (Db2 SQL)**

### Retrieving messages with IBM MQ

With IBM MQ, programs can read or receive messages. Both reading and receiving operations return the message at the start of the queue. However, the reading operation does not remove the message from the queue, whereas the receiving operation does.

### About this task

A message that is retrieved using a receive operation can be retrieved only once, whereas a message that is retrieved using a read operation allows the same message to be retrieved many times.

### Examples

The following examples use the DB2MQ2N schema for two-phase commit, with the default service Db2.DEFAULT.SERVICE and the default policy Db2.DEFAULT.POLICY.

#### Example

The following SQL SELECT statement reads the message at the head of the queue that is specified by the default service and policy:

```sql
SELECT DB2MQ2N.MQREAD()
FROM SYSIBM.SYSDUMMY1;
```

The MQREAD function is invoked once because SYSIBM.SYSDUMMY1 has only one row. The SELECT statement returns a VARCHAR(4000) string. If no messages are available to be read, a null value is returned. Because MQREAD does not change the queue, you do not need to use a COMMIT statement.

#### Example

The following SQL SELECT statement causes the contents of a queue to be materialized as a Db2 table:

```sql
SELECT T.*
FROM TABLE(DB2MQ2N.MQREADALL()) T;
```

The result table T of the table function consists of all the messages in the queue, which is defined by the default service, and the metadata about those messages. The first column of the materialized result table is the message itself, and the remaining columns contain the metadata. The SELECT
statement returns both the messages and the metadata. To return only the messages, issue the following statement:

```sql
SELECT T.MSG
FROM TABLE(DB2MQ2N.MQREADALL()) T;
```

The result table T of the table function consists of all the messages in the queue, which is defined by the default service, and the metadata about those messages. This SELECT statement returns only the messages.

**Example**

The following SQL SELECT statement receives (removes) the message at the head of the queue:

```sql
SELECT DB2MQ2N.MQRECEIVE()
FROM SYSIBM.SYSDUMMY1;
COMMIT;
```

The MQRECEIVE function is invoked once because SYSIBM.SYSDUMMY1 has only one row. The SELECT statement returns a VARCHAR(4000) string. Because this MQRECEIVE function uses two-phase commit, the COMMIT statement ensures that the message is removed from the queue. If no messages are available to be retrieved, a null value is returned, and the queue does not change.

**Example**

Assume that you have a MESSAGES table with a single VARCHAR(2000) column. The following SQL INSERT statement inserts all of the messages from the default service queue into the MESSAGES table in your Db2 database:

```sql
INSERT INTO MESSAGES
SELECT T.MSG
FROM TABLE(DB2MQ2N.MQRECEIVEALL()) T;
COMMIT;
```

The result table T of the table function consists of all the messages in the default service queue and the metadata about those messages. The SELECT statement returns only the messages. The INSERT statement stores the messages into a table in your database.

**Application to application connectivity with IBM MQ**

Application-to-application connectivity is typically used when putting together a diverse set of application subsystems. To facilitate application integration, WebSphere MQ provides the means to interconnect applications.

The *Request-and-reply* method is very common when interconnecting applications.

**Request-and-reply communication method**

The request-and-reply method enables one application to request the services of another application. One way to do this is for the requester to send a message to the service provider to request that some work be performed. When the work has been completed, the provider might decide to send results, or just a confirmation of completion, back to the requester. Unless the requester waits for a reply before continuing, IBM MQ must provide a way to associate the reply with its request.

IBM MQ provides a correlation identifier to correlate messages in an exchange between a requester and a provider. The requester marks a message with a known correlation identifier. The provider marks its reply with the same correlation identifier. To retrieve the associated reply, the requester provides that correlation identifier when receiving messages from the queue. The first message with a matching correlation identifier is returned to the requester.

The following examples use the DB2MQ schema for single-phase commit.
Examples of request-and-reply communication

Example

The following SQL SELECT statement sends a message consisting of the string "Msg with corr id" to the service MYSERVICE, using the policy MYPOLICY with correlation identifier CORRID1:

```
SELECT DB2MQ.MQSEND ('MYSERVICE', 'MYPOLICY', 'Msg with corr id', 'CORRID1')
FROM SYSIBM.SYSDUMMY1;
```

The MQSEND function is invoked once because SYSIBM.SYSDUMMY1 has only one row. Because this MQSEND uses single-phase commit, IBM MQ adds the message to the queue, and you do not need to use a COMMIT statement.

Example

The following SQL SELECT statement receives the first message that matches the identifier CORRID1 from the queue that is specified by the service MYSERVICE, using the policy MYPOLICY:

```
SELECT DB2MQ.MQRECEIVE ('MYSERVICE', 'MYPOLICY', 'CORRID1')
FROM SYSIBM.SYSDUMMY1;
```

The SELECT statement returns a VARCHAR(4000) string. If no messages are available with this correlation identifier, a null value is returned, and the queue does not change.

Asynchronous messaging in Db2 for z/OS

Programs can communicate with each other by sending data in messages rather than using constructs like synchronous remote procedure calls. With asynchronous messaging, the program that sends the message proceeds with its processing after sending the message, without waiting for a reply.

If the program needs information from the reply, the program suspends processing and waits for a reply message. If the messaging programs use an intermediate queue that holds messages, the requester program and the receiver program do not need to be running at the same time. The requester program places a request message on a queue and then exits. The receiver program retrieves the request from the queue and processes the request.

Asynchronous operations require that the service provider is capable of accepting requests from clients without notice. An asynchronous listener is a program that monitors message transporters, such as WebSphere MQ, and performs actions based on the message type. An asynchronous listener can use WebSphere MQ to receive all messages that are sent to an endpoint. An asynchronous listener can also register a subscription with a publish or subscribe infrastructure to restrict the messages that are received to messages that satisfy specified constraints.

Examples: The following examples show some common uses of asynchronous messaging:

Message accumulator

You can accumulate the messages that are sent asynchronously so that the listener checks for messages and stores those messages automatically in a database. This database, which acts as a message accumulator, can save all messages for a particular endpoint, such as an audit trail. The asynchronous listener can subscribe to a subset of messages, such as save only high value stock trades. The message accumulator stores entire messages, and does not provide for selection, transformation, or mapping of message contents to database structures. The message accumulator does not reply to messages.

Message event handler

The asynchronous event handler listens for messages and invokes the appropriate handler (such as a stored procedure) for the message endpoint. You can call any arbitrary stored procedure. The asynchronous listener lets you select, map, or reformat message contents for insertion into one or more database structures.

Asynchronous messaging has the following benefits:

- The client and database do not need to be available at the same time. If the client is available intermittently, or if the client fails between the time the request is issued and the response is sent,
it is still possible for the client to receive the reply. Or, if the client is on a mobile computer and becomes disconnected from the database, and if a response is sent, the client can still receive the reply.

- The content of the messages in the database contain information about when to process particular requests. The messages in the database use priorities and the request contents to determine how to schedule the requests.

- An asynchronous message listener can delegate a request to a different node. It can forward the request to a second computer to complete the processing. When the request is complete, the second computer returns a response directly to the endpoint that is specified in the message.

- An asynchronous listener can respond to a message from a supplied client, or from a user-defined application. The number of environments that can act as a database client is greatly expanded. Clients such as factory automation equipment, pervasive devices, or embedded controllers can communicate with Db2 either directly through IBM MQ or through some gateway that supports WebSphere MQ.

**MQListener in Db2 for z/OS**

Db2 for z/OS provides an asynchronous listener, MQListener. MQListener is a framework for tasks that read from IBM MQ queues and call Db2 stored procedures with messages as those messages arrive.

MQListener combines messaging with database operations. You can configure the MQListener daemon to listen to the IBM MQ message queues that you specify in a configuration database. MQListener reads the messages that arrive from the queue and calls Db2 stored procedures using the messages as input parameters. If the message requires a reply, MQListener creates a reply from the output that is generated by the stored procedure. The message retrieval order is fixed at the highest priority first, and then within each priority the first message received is the first message served.

MQListener runs as a single multi-threaded process on z/OS UNIX System Services. Each thread or task establishes a connection to its configured message queue for input. Each task also connects to a Db2 database on which to run the stored procedure. The information about the queue and the stored procedure is stored in a table in the configuration database. The combination of the queue and the stored procedure is a task.

MQListener tasks are grouped together into named configurations. By default, the configuration name is empty. If you do not specify the name of a configuration for a task, MQListener uses the configuration with an empty name.

**Transaction support**

There is support for both one-phase and two-phase commit environments. A one-phase commit environment is where DB interactions and MQ interactions are independent. A two-phase commit environment is where DB interactions and MQ interactions are combined in a single unit of work.

'`db2mqln1`' is the name of the executable for one phase and '`db2mqln2`' is the name of the executable for two phase.

**Logical ordering of messages**

The two-phase commit version of the MQListener stored procedure processes messages that are in a group in logical order. The single-phase commit version of the MQListener stored procedure processes messages that are in a group in physical order.

**Stored procedure interfaces**

The MQListener interface supports two stored procedure formats: with either two or three parameters.

The data type and length of the stored procedure parameters are determined when the MQListener is started. If you change the data type or length of the parameters, the change takes effect when you restart the MQListener. You can use the following commands to restart the MQListener:

```
mqlistener-command mqlistener-command admin
   -adminQueue adminqueue-name
```
Stored procedure interface with two parameters

This stored procedure interface for MQListener takes the incoming message as input and returns the reply, which might be NULL, as output. For example:

```
CREATE schema.proc(
    IN INMSG inMsgType,
    OUT OUTMSG outMsgType)
```

The data type for INMSG and the data type for OUTMSG can be VARCHAR, VARBINARY, CLOB, or BLOB, of any length, and are determined at startup. The input data type and output data type can be different data types. If an incoming message is a request and has a specified reply-to queue, the message in OUTMSG is sent to the specified queue. The incoming message can be one of the following message types:

- Datagram
- Datagram with report requested
- Request message with reply
- Request message with reply and report requested

Stored procedure interface with three parameters

This stored procedure interface for MQListener has parameters with the following information:

- An incoming message as input
- A reply, which might be NULL, as output
- A message header, which can be input or output

For example:

```
CREATE schema.proc(
    IN INMSG inMsgType,
    OUT OUTMSG outMsgType,
    INOUT MSGHEADER msgHeaderType)
```

The data type for INMSG and the data type for OUTMSG can be VARCHAR, VARBINARY, CLOB, or BLOB, of any length, and are determined at startup. The input data type and output data type can be different data types. If an incoming message is a request and has a specified reply-to queue, the message in OUTMSG is sent to the specified queue. The incoming message can be one of the following message types:

- Datagram
- Datagram with report requested
- Request message with reply
- Request message with reply and report requested

The data type for MSGHEADER can be VARBINARY or BLOB. The minimum length of MSGHEADER is 324, which is the size of the message queuing message descriptor (MQMD) structure for IBM MQ messages.

MQListener passes the message header to the stored procedure in the MSGHEADER parameter. The stored procedure can get the message descriptor properties from the MSGHEADER parameter. If the message is a request message, the stored procedure can specify the properties for the reply queue and reply queue manager in the MSGHEADER parameter. The output message in OUTMSG is sent to that specified queue.
**Configuring MQListener in Db2 for z/OS**

Before you can use MQListener, you must configure your database environment so that your applications can use messaging with database operations. You must also configure IBM MQ for MQListener.

**Before you begin**

Ensure that the person who runs the installation job has required authority to create the configuration table and to bind the DBRMs.

**About this task**

The following sample jobs for MQListener are located in the `prefix.SDSNSAMP` data set.

- **DSNTIJML**
  - A sample job for extracting library files and configuring MQListener in z/OS UNIX System Services.

- **DSNTEJML**
  - A sample job that runs scripts for configuring MQListener.

- **DSNTEJSP**
  - A sample job for extracting library tar files in z/OS UNIX System Services when applying PTFs for MQListener.

**Procedure**

To configure the environment for MQListener and to develop a simple application that receives a message, inserts the message in a table, and creates a simple response message, use these steps:

1. Configure MQListener to run in the Db2 environment, so that your applications can use messaging with database operations, by completing the following steps:
   a) In z/OS UNIX System Services, the default path for MQListener in is `/usr/lpp/db2c10/mql`. The `mqlsn.tar.Z` tar file for MQListener is located in this path. If MQListener is not installed in the default path, replace all occurrences of the default path in the samples DSNTEJML, DSNTEJSP and DSNTIJML with the path name where MQListener is installed before you run DSNTIJML.
   b) Customize and run installation job DSNTIJML. It completes the following actions:
      i) Extracts the necessary files and libraries to z/OS UNIX System Services under the path where MQListener is installed.
      ii) Creates the MQListener configuration table (SYSMQL LISTENERS) in the default database DSNDDB04.
      iii) Binds the DBRMs to the plan DB2MQLSN.
   c) Follow the instructions in the README file in the MQListener installation path to complete the configuration process.

2. Configure IBM MQ for MQListener.

You can run a simple MQListener application with a simple IBM MQ configuration. More complex applications might need a more complex configuration. Configure at least two kinds of IBM MQ entities: the queue manager and some local queues. Configure these entities for use in such instances as transaction management, deadletter queue, backout queue, and backout retry threshold.

   a) Create IBM MQ QueueManager. Define the IBM MQ subsystem to z/OS and then issue the following command from a z/OS console to start the queue manager, where `command-prefix` is the command prefix for the IBM MQ subsystem.

      ```shell
      command-prefix START QMGR
      ```

   b) Create Queues under IBM MQ QueueManager:
For example, in a simple MQListener application, you typically use the following IBM MQ queues:

**Deadletter queue**

The deadletter queue in IBM MQ holds messages that cannot be processed. MQListener uses this queue to hold replies that cannot be delivered, for example, because the queue to which the replies should be sent is full. A deadletter queue is useful in any MQ installation especially for recovering messages that are not sent.

**Backout queue**

For MQListener tasks that use two-phase commit, the backout queue serves a similar purpose as the deadletter queue. MQListener places the original request in the backout queue after the request is rolled back a specified number of times (called the backout threshold).

**Administration queue**

The administration queue is used for routing control messages, such as shutdown and restart, to MQListener. If you do not supply an administration queue, the only way to shut down MQListener is to issue a kill command.

**Application input and output queues**

The application uses input queues and output queues. The application receives messages from the input queue and sends replies and exceptions to the output queue.

Create your local queues by using CSQUTIL utility or by using IBM MQ operations and control panels from ISPF (csqorexx). The following is an example of the JCL that is used to create your local queues. In this example, MQND is the name of the queue manager:

```
//*
//* ADMIN_Q      : Admin queue
//* BACKOUT_Q    : Backout queue
//* IN_Q         : Input queue having a backout queue with threshold=3
//* REPLY_Q      : output queue or reply queue
//* DEADLETTER_Q : Dead letter queue
//*
//* DSNTECU   EXEC PGM=CSQUTIL,PARM='MQND'
//STEPLIB   DD   DSN=MQS.SCSQANLE,DISP=SHR
//          DD   DSN=MQS.SCSQAUTH,DISP=SHR
//SYSPRINT  DD   SYSOUT=*    
//SYSIN     DD *
COMMAND     DDNAME(CREATEQ)
/*
CREATEQ DD *
DEFINE QLOCAL('ADMIN_Q') REPLACE +
      DESCR('INPUT-OUTPUT') +
      PUT(ENABLED) +
      DEFPRTY(0) +
      DEFPSSIST(NO) +
      SHARE +
      DEFSOFT(SHARED) +
      GET(ENABLED)
DEFINE QLOCAL('BACKOUT_Q') REPLACE +
      DESCR('INPUT-OUTPUT') +
      PUT(ENABLED) +
      DEFPRTY(0) +
      DEFPSSIST(NO) +
      SHARE +
      DEFSOFT(SHARED) +
      GET(ENABLED)
DEFINE QLOCAL('REPLY_Q') REPLACE +
      DESCR('INPUT-OUTPUT') +
      PUT(ENABLED) +
      DEFPRTY(0) +
      DEFPSSIST(NO) +
      SHARE +
      DEFSOFT(SHARED) +
      GET(ENABLED)
DEFINE QLOCAL('IN_Q') REPLACE +
      DESCR('INPUT-OUTPUT') +
      PUT(ENABLED) +
      DEFPRTY(0) +
      DEFPSSIST(NO) +
      SHARE +
      DEFSOFT(SHARED) +
      GET(ENABLED) +
      BOQNAME('BACKOUT_Q') +
      BOTHRESH(3)
```
DEFINE QLOCAL('DEADLETTER_Q') REPLACE +
DESCR('INPUT-OUTPUT') +
PUT(ENABLED) +
DEFPRTY(9) +
DEFPSIST(NO) +
SHARE +
DEFSOPT(SHARED) +
GET(ENABLED)
/*
ALTER QMGR DEADQ ('DEADLETTER_Q') REPLACE

3. Configure MQListener tasks. For more information, see “Configuring MQListener tasks” on page 804.  
4. Create a stored procedure that MQListener uses to store messages in a table. See “Creating a sample  
stored procedure to use with MQListener” on page 807 for details.  
5. Run a simple MQListener application.  

Environment variables for logging and tracing MQListener  
Several environment variables control logging and tracing for MQListener. These variables are defined in  
the file .profile.  

MQLSNTRC  
When this environment variable is set to 1, MQListener writes function entry, data, and exit points to  
a unique HFS or zFS file. A unique trace file is generated whenever any of the MQListener commands  
are run. This trace file is used by IBM Support for debugging. Do not define this variable unless IBM  
Support requests that you do so.  

If you have enabled tracing, when the MQListener daemon is running, it writes to the trace file.  
Therefore, if you open the trace file while the MQListener daemon is running, you need to open it only  
in read mode.  

MQLSNLOG  
The log file contains diagnostic information about major events. This environment variable is set to the  
name of the file where all log information is written. All instances of the MQListener daemon running  
one or more tasks share the same file. For monitoring the MQListener daemon, this variable should  
always be set.  

When the MQListener daemon is running, it writes to the log file. Therefore, if you open the log file  
while the MQListener daemon is running, you need to open it only in read mode.  

MQLSNLWR  
When MQLSNLOG specifies an HFS log file, MQLSNLWR provides the capability to limit the HFS log file  
size. The syntax for an MQLSNLWR export command is:  

export MQLSNLWR=file-size,file-name  
The meanings of the variables are:  

file-size  
The maximum size of the MQListener log file, in megabytes.  

file-name  
The name of the HFS file into which MQListener saves a copy of the MQListener log file.  

When MQLSNLWR is specified, and the MQListener HFS log file reaches file-size, MQListener saves a  
copy of the log file named file-name, and reinitializes the HFS log file.  

Important: The ID under which MQListener runs must have write access to the HFS log file and to the  
copy of the HFS log file that is specified by file-name. If MQListener cannot open or write to the copy  
of the HFS log file, MQListener reinitializes the HFS log file, but does not create a copy.  

Refer to the README file for more details about these variables.
If you use MQListener, you must create the MQListener configuration table SYSMQL.LISTENERS by running installation job DSNTIJML.

The SYSMQL.LISTENERS table contains a row for each configuration that you create when you issue MQListener db2mqln1 or db2mqln2 configuration commands.

The following table describes each of the columns of the configuration table SYSMQL.LISTENERS.

<table>
<thead>
<tr>
<th>Column name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CONFIGURATIONNAME</td>
<td>The configuration name. The configuration name enables you to group several tasks into the same configuration. A single instance of MQListener can run all of the tasks that are defined within a configuration name.</td>
</tr>
<tr>
<td>QUEUEMANAGER</td>
<td>The name of the IBM MQ subsystem that contains the queues that are to be used.</td>
</tr>
<tr>
<td>INPUTQUEUE</td>
<td>The name of the queue in the WebSphere MQ subsystem that is to be monitored for incoming messages. The combination of the input queue and the queue manager are unique within a configuration</td>
</tr>
<tr>
<td>PROCNODE</td>
<td>Currently unused</td>
</tr>
<tr>
<td>PROCSCHEMA</td>
<td>The schema name of the stored procedure that will be called by MQListener</td>
</tr>
<tr>
<td>PROCNAME</td>
<td>The name of the stored procedure that will be called by MQListener</td>
</tr>
<tr>
<td>PROCTYPE</td>
<td>Currently unused</td>
</tr>
<tr>
<td>NUMINSTANCES</td>
<td>The number of duplicate instances of a single task that are to run in this configuration</td>
</tr>
<tr>
<td>WAITMILLIS</td>
<td>The time MQListener waits (in milliseconds) after processing the current message before it looks for the next message</td>
</tr>
<tr>
<td>MINQUEUEDEPTH</td>
<td>Currently unused</td>
</tr>
</tbody>
</table>

Configuring MQListener tasks
As part of configuring MQListener in Db2 for z/OS, you must configure at least one MQListener task.

About this task
Use MQListener command `db2mqln1` or `db2mqln2` to configure MQListener tasks. Issue the command from the z/OS UNIX System Services command line in any directory. Alternatively, you can put the command in a file, grant execute permission on the file, and use the BPXBATCH utility to invoke the command using JCL. Sample script files are provided and are located in the `/mqlistener-install-path/mqlsn/listener/script` directory in z/OS UNIX System Services. Sample JCL is also provided in member DSNTIJML of data set `prefix.SDSNSAMP`. When you run MQListener commands, configuration information is stored in the Db2 table SYSMQL.LISTENERS.

The command parameters are:

-`-adminQueue`  
The queue to which MQListener listens for administration commands. If `-adminQueue` is not specified, applications do not receive any administration commands through the message queue.
-adminQMgr
  The name of the IBM MQ subsystem that contains the queues that are to be used for administrative tasks. If -adminQMgr is not specified, the configured default queue manager is used.

-config
  A name that identifies a group of tasks that run together. If -config is not specified, the default configuration is run.

-queueManager
  The name of the IBM MQ subsystem that contains the queues that are to be used. If -queueManager is not specified, is not specified, the default queue manager is used.

-inputQueue
  The name of the queue in the IBM MQ subsystem that is to be monitored for incoming messages. The combination of the -inputQueue value and the -queueManager value must be unique within a configuration.

-numInstances
  The number of duplicate instances of a single task that are to run in a configuration.

-numMessagesCommit
  The number of messages that are received before MQListener issues a COMMIT. The default is 1. This option is supported only for db2mq1n2.

-procName
  The name of the stored procedure that MQListener calls when it detects that a message is received.

-procSchema
  The schema name of the stored procedure that MQListener calls when it detects that a message is received.

-ssID
  The subsystem where the MQListener daemon runs. Configuration information is stored in this subsystem.

-timeRestart
  If a stored procedure that is specified by -procSchema and -procName fails at MQListener startup time, the number of seconds that threads that are running with that stored procedure suspend before repeating the setup process. MQListener continues startup for threads that do not use that stored procedure. This value must be an integer between 0 and 7200. 0 is the default.

-restartDB2
  Whether MQListener automatically reconnects and resumes processing after Db2 is stopped and restarted.

    'Y'
    MQListener automatically reconnects and resumes processing after Db2 is stopped and restarted. 'N' is the default value.

    'N'
    MQListener does not automatically reconnect after Db2 is stopped and restarted. 'N' is the default value.

The syntax of the commands follows. In the command syntax, mqlistener-command is db2mq1n1 or db2mq1n2.

• To add an MQListener configuration, issue the mqlistener-command add command:

  mqlistener-command add
    -ssID subsystem-name
    -config configuration-name
    -queueManager queueManager-name
    -inputQueue inputQueue-name
    -procName stored-procedure-name
    -procSchema stored-procedure-schema name
    -numInstances number-of-instances
• To display information about the configuration, issue the following `mqlistener-command show` command:

```
mqlistener-command show
  -ssID subsystem-name
  -config configuration-name
```

To display information about all the configurations, issue the `mqlistener-command show` command:

```
mqlistener-command show
  -ssID subsystem-name
  -config all
```

• To remove the messaging tasks, issue the `mqlistener-command remove` command:

```
mqlistener-command remove
  -ssID subsystem-name
  -config configuration-name
  -queueManager queuemanager-name
  -inputQueue inputqueue-name
```

• To run the MQListener task, issue the `mqlistener-command run` command:

```
mqlistener-command run
  -ssID subsystem-name
  -config configuration-name
  -adminQueue adminqueue-name
  -adminQMgr adminqueuemanager-name
  -numMessagesCommit number-of-messages-before-commit
  -timeRestart number-of-seconds-to-suspend-before-restart
  -restartDB2 y-or-n
```

• To shutdown the MQListener daemon, issue the `mqlistener-command admin` command:

```
mqlistener-command admin
  -adminQueue adminqueue-name
  -adminQMgr adminqueuemanager-name
```

• To restart the MQListener daemon, issue the following command:

```
mqlistener-command mqlistener-command admin
  -adminQueue adminqueue-name
  -adminQMgr adminqueuemanager-name
  -adminCommand restart
```

• To get help with the command and the valid parameters, issue the `mqlistener-command help` command:

```
mqlistener-command help
```

• To get help for a particular parameter, issue the `mqlistener-command help` command, where `command` is a specific parameter:

```
mqlistener-command help command
```

**Restriction:**

• Use the same queue manager for the request queue and the reply queue.
• MQListener does not support logical messages that are composed of multiple physical messages. MQListener processes physical messages independently.
Creating a sample stored procedure to use with MQListener

You can create a sample stored procedure, APROC, that can be used by MQListener to store a message in a table. The stored procedure returns the string OK if the message is successfully inserted into the table.

About this task

This example assumes the following information about the environment:

• MQListener is installed and configured for subsystem DB2A.
• The IBM MQ subsystem that is defined is named CSQ1.
• The queue manager is running, and the following local queues are defined in the DB2A subsystem:
  - ADMIN_Q: The administration queue
  - BACKOUT_Q: The backout queue
  - DB2MQ_DEFAULT_Q: The input queue, which has a backout queue with a threshold of 3
  - REPLY_Q: The output queue or reply queue
  - DEADLETTER_Q: The dead letter queue
• The user who is running the MQListener daemon has the EXECUTE privilege on the DB2MQLSN plan.
• MQListener passes the message header (MQMD structure) to the stored procedure interface for MQListener.

Procedure

The following steps create Db2 objects that you can use with MQListener applications:

1. Create a table using SPUFI, DSNTEP2, or the command line processor in the subsystem where you want to run MQListener:

   ```
   CREATE TABLE PROCTABLE (MSG VARCHAR(25) CHECK (MSG NOT LIKE 'FAIL%'));
   ```

   The table contains a check constraint so that messages that start with the characters FAIL cannot be inserted into the table. The check constraint is used to demonstrate the behavior of MQListener when the stored procedure fails.

2. Create the following SQL procedure, and define it to the same Db2 subsystem.

   ```
   CREATE PROCEDURE TEST.APROC (
      IN PIN VARCHAR(25),
      OUT POUT VARCHAR(2),
      INOUT PMSGHEADER VARBINARY(500))
   VERSION V1
   LANGUAGE SQL
   PROCEDURE1: BEGIN
      DECLARE REPLYQ  VARBINARY(48);
      DECLARE REPLYQM VARBINARY(48);
      SET REPLYQ = VARBINARY(CONCAT('NEWREPLYQUEUE',X'00'));
      SET REPLYQM = VARBINARY(CONCAT('CSQ1',X'00'));
      SET PMSGHEADER = INSERT(PMSGHEADER,101,LENGTH(REPLYQ),REPLYQ);
      SET PMSGHEADER = INSERT(PMSGHEADER,149,LENGTH(REPLYQM),REPLYQM);
      INSERT INTO SYSADM.PROCTABLE VALUES (PIN);
      SET POUT = 'OK';
   END PROCEDURE1
   ```

3. Add the following configuration, named ACFG, to the configuration table by issuing this command:

   ```
   db2mqln2 add
   -ssID DB2A
   -config ACFG
   -queueManager CSQ1
   -inputQueue DB2MQ_DEFAULT_Q
   -procName APROC
   -procSchema TEST
   ```

4. Run the MQListener daemon for two-phase commit for configuration ACFG. To run MQListener with all of the tasks that are specified in the configuration, issue the following command:
5. Send a request to the input queue, ‘DB2MQ_DEFAULT_Q’, with the message ‘another sample message’.

6. Query table PROCTABLE to verify that the sample message was inserted:

   ```sql
   SELECT * FROM PROCTABLE;
   ```

7. Display the number of messages that remain on the input queue, to verify that the message has been removed. To do that issue the following command from a z/OS console:

   ```bash
   /-CSQ1 display queue('DB2MQ_DEFAULT_Q ') curdepth
   ```

8. Look at the ReplytoQ name that you specified, to verify that the string 'OK' is generated by the stored procedure.

**MQListener error processing**

MQListener reads from IBM MQ message queues and calls Db2 stored procedures with those messages. If any errors occur during this process and the message is to be sent to the deadletter queue, MQListener returns a reason code to the deadletter queue.

Specifically, MQListener performs the following actions:

- prefixes the message with an MQ dead letter header (MQDLH) structure
- sets the reason field in the MQDLH structure to the appropriate reason code
- sends the message to the deadletter queue

The following table describes the reason codes that the MQListener daemon returns.

<table>
<thead>
<tr>
<th>Reason code</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>900</td>
<td>The call to a stored procedure was successful but an error occurred during the Db2 commit process and either of the following conditions were true:</td>
</tr>
<tr>
<td></td>
<td>• No exception report was requested. ¹</td>
</tr>
<tr>
<td></td>
<td>• An exception report was requested, but could not be delivered.</td>
</tr>
<tr>
<td></td>
<td>This reason code applies only to one-phase commit environments.</td>
</tr>
<tr>
<td>901</td>
<td>The call to the specified stored procedure failed and the disposition of the MQ message is that an exception report be generated and the original message be sent the deadletter queue.</td>
</tr>
<tr>
<td>902</td>
<td>All of the following conditions occurred:</td>
</tr>
<tr>
<td></td>
<td>• The disposition of the MQ message is that an exception report is not to be generated. ¹</td>
</tr>
<tr>
<td></td>
<td>• The stored procedure was called unsuccessfully the number of times that is specified as the backout threshold.</td>
</tr>
<tr>
<td></td>
<td>• The name of the backout queue is the same as the deadletter queue.</td>
</tr>
<tr>
<td></td>
<td>This reason code applies only to two-phase commit environments.</td>
</tr>
<tr>
<td>MQRC_TRUNCATED_MSG__FAILED</td>
<td>The size of the MQ message is greater than the input parameter of the stored procedure that is to be invoked. In one-phase commit environments, this oversized message is sent to the dead letter queue. In two-phase commit environments, this oversized message is sent to the deadletter queue only when the message cannot be delivered to the backout queue.</td>
</tr>
</tbody>
</table>
Note:

1. To specify that the receiver application generate exception reports if errors occur, set the report field in the MQMD structure that was used when sending the message to one of the following values:
   - MQRO_EXCEPTION
   - MQRO_EXCEPTION_WITH_DATA
   - MQRO_EXCEPTION_WITH_FULL_DATA

Related reference
IBM MQ home

MQListener examples
The application receives a message, inserts the message into a table, and generates a simple response message.

To simulate a processing failure, the application includes a check constraint on the table that contains the message. The constraint prevents any string that begins with the characters 'fail' from being inserted into the table. If you attempt to insert a message that violates the check constraint, the example application returns an error message and re-queues the failing message to the backout queue.

In this example, the following assumptions are made:

- MQListener is installed and configured for subsystem DB7A.
- MQND is the name of IBM MQ subsystem that is defined. The Queue Manager is running, and the following local queues are defined in the DB7A subsystem:
  
  - ADMIN_Q : Admin queue
  - BACKOUT_Q : Backout queue
  - IN_Q : Input queue that has a backout queue with threshold = 3
  - REPLY_Q : Output queue or Reply queue
  - DEADLETTER_Q : Dead letter queue

- The person who is running the MQListener daemon has execute permission on the DB2MQLSN plan.

Before you run the MQListener daemon, add the following configuration, named ACFG, to the configuration table by issuing the following command:

```
db2mqln2 add
  -ssID DB7A
  -config ACFG
  -queueManager MQND
  -inputQueue IN_Q
  -procName APROC
  -procSchema TEST
```

Run the MQListener daemon for two-phase commit for configuration ACFG by issuing the following command:

```
db2mqln2 run
  -ssID DB7A
  -config ACFG
  -adminQueue ADMIN_Q
  -adminQMgr MQND
  -numMessagesCommit 1
  -timeRestart 60
```

The following examples show how to use MQListener to send a simple message and then inspect the results of the message in the IBM MQ queue manager and the database. The examples include queries to determine if the input queue contains a message or to determine if a record is placed in the table by the stored procedure.

MQListener example 1: Running a simple application:

1. Start with a clean database table by issuing the following SQL statement:
2. Send a datagram to the input queue, 'IN_Q', with the message as 'sample message'. Refer to WebSphere MQ sample CSQ4BCK1 to send a message to the queue. Specify the MsgType option for 'Message Descriptor' as 'MQMT_DATAGRAM'.

3. Query the table by using the following statement to verify that the sample message is inserted:

   ```sql
   select * from PROCTABLE
   ```

4. Display the number of messages that remain on the input queue to verify that the message has been removed. Issue the following command from a z/OS console:

   ```
   /-MQND display queue('In_Q') curdepth
   ```

**MQListener example 2: Sending requests to the input queue and inspecting the reply:**

1. Start with a clean database table by issuing the following SQL statement:

   ```sql
   delete from PROCTABLE
   ```

2. Send a request to the input queue, 'IN_Q', with the message as 'another sample message'. Refer to IBM MQ sample CSQ4BCK1 to send a message to the queue. Specify the MsgType option for 'Message Descriptor' as 'MQMT_REQUEST' and the queue name for ReplytoQ option.

3. Query the table by using the following statement to verify that the sample message is inserted:

   ```sql
   select * from PROCTABLE
   ```

4. Display the number of messages that remain on the input queue to verify that the message has been removed. Issue the following command from a z/OS console:

   ```
   /-MQND display queue('In_Q') curdepth
   ```

5. Look at the ReplytoQ name that you specified when you sent the request message for the reply by using the IBM MQ sample program CSQ4BCJ1. Verify that the string 'OK' is generated by the stored procedure.

**MQListener example 3: Testing an unsuccessful insert operation:** If you send a message that starts with the string 'fail', the constraint in the table definition is violated, and the stored procedure fails.

1. Start with a clean database table by issuing the following SQL statement:

   ```sql
   delete from PROCTABLE
   ```

2. Send a request to the input queue, 'IN_Q', with the message as 'failing sample message'. Refer to IBM MQ sample CSQ4BCK1 to send a message to the queue. Specify the MsgType option for 'Message Descriptor' as 'MQMT_REQUEST' and the queue name for ReplytoQ option.

3. Query the table by using the following statement to verify that the sample message is not inserted:

   ```sql
   select * from PROCTABLE
   ```

4. Display the number of messages that remain on the input queue to verify that the message has been removed. Issue the following command from a z/OS console:

   ```
   /-MQND display queue('In_Q') curdepth
   ```

5. Look at the Backout queue and find the original message by using the WebSphere MQ sample program CSQ4BCJ1.

   **Note:** In this example, if a request message with added options for 'exception report' is sent (the Report option is specified for 'Message Descriptor'), an exception report is sent to the reply queue and the original message is sent to the deadletter queue.
Chapter 7. Db2 as a web services consumer and provider

Web services are a set of resources and components that applications can use over HTTP. You can use Db2 as a web services provider and a web services consumer.

Db2 as a web services consumer

Db2 can act as a client for web services, which enables you to be a consumer of web services in your Db2 applications.

SOAP web services Simple Object Access Protocol (SOAP) is an XML protocol that consists of the following characteristics:
- An envelope that defines a framework for describing the contents of a message and how to process the message
- A set of encoding rules for expressing instances of application-defined data types
- A convention for representing SOAP requests and responses

A set of SOAP functions is provided by Db2 and is installed and configured when you install or migrate Db2.

REST web services The Representational State Transfer (REST) protocol provides access to web-based content directly from SQL statements through HTTP requests. A set of basic sample REST user-defined functions can be installed with Db2. These functions provide access to web-based content through the HTTP GET, POST, PUT, and DELETE methods.

Db2 as a web services provider

You can enable your Db2 data and applications as web services through the Web Services Object Runtime Framework (WORF). You can define a web service in Db2 by using a Document Access Definition Extension (DADX). In the DADX file, you can define web services based on SQL statements and stored procedures. Based on your definitions in the DADX file, WORF performs the following actions:
- Handles the connection to Db2 and the execution of the SQL and the stored procedure call
- Converts the result to a web service
- Handles the generation of any Web Services Definition Language (WSDL) and UDDI (Universal Description, Discovery, and Integration) information that the client application needs

Related concepts
Sample REST user-defined functions (Db2 Installation and Migration)

Deprecated: The SOAPHTTPV and SOAPHTTPC user-defined functions

Db2 provides user-defined functions that allow you to work with SOAP and consume web services in SQL statements. The user-defined functions are two varieties of SOAPHTTPV for VARCHAR data and two varieties of SOAPHTTPC for CLOB data.

Restriction: SOAPHTTPV and SOAPHTTPC user-defined functions have been deprecated. Use SOAPHTTPNV and SOAPHTTPNC user-defined functions instead.

The user-defined functions perform the following actions:
1. Compose a SOAP request
2. Post the request to the service endpoint
3. Receive the SOAP response
4. Return the content of the SOAP body

When a consumer receives the result of a web services request, the SOAP envelope is stripped and the XML document is returned. An application program can process the result data and perform a variety of operations, including inserting or updating a table with the result data.

SOAPHTTPV and SOAPHTTPC are user-defined functions that enable Db2 to work with SOAP and to consume web services in SQL statements. These functions are overloaded functions that are used for VARCHAR or CLOB data of different sizes, depending on the SOAP body. Web services can be invoked in one of four ways, depending on the size of the input data and the result data. SOAPHTTPV returns VARCHAR(32672) data and SOAPHTTPC returns CLOB(1M) data. Both functions accept either VARCHAR(32672) or CLOB(1M) as the input body.

**Example:** The following example shows an HTTP post header that posts a SOAP request envelope to a host. The SOAP envelope body shows a temperature request for Barcelona.

```sql
POST /soap/servlet/rpcrouter HTTP/1.0
Host: services.xmethods.net
Connection: Keep-Alive
User-Agent: DB2SOAP/1.0
Content-Type: text/xml; charset="UTF-8"
SOAPAction: ""
Content-Length: 410

xmlns:SOAP-ENC=http://schemas.xmlsoap.org/soap/encoding/
xmni:xsi=http://www.w3.org/2001/XMLSchema-instance
xmlns:xsd=http://www.w3.org/2001/XMLSchema >

<SOAP-ENV:Body>
<ns:getTemp xmlns:ns="urn:xmethods-Temperature">
<city>Barcelona</city>
</ns:getTemp>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

**Example:** The following example is the result of the preceding example. This example shows the HTTP response header with the SOAP response envelope. The result shows that the temperature is 85 degrees Fahrenheit in Barcelona.

HTTP/1.1 200 OK
Date: Wed, 31 Jul 2002 22:06:41 GMT
Server: Enhydra-MultiServer/3.5.2
Status: 200
Content-Type: text/xml; charset=utf-8
Servlet-Engine: Lutris Enhydra Application Server/3.5.2
(JSP 1.1; Servlet 2.2; Java 1.3.1_04; Linux 2.4.7-10smp i386; java.vendor=Sun Microsystems Inc.)
Content-Length: 467
Set-Cookie:JSESSIONID=JLEcR34rBc2G7Ikn-0F51ZDK;Path=/soap
X-Cache: MISS from www.xmethods.net
Keep-Alive: timeout=15, max=10
Connection: Keep-Alive

xmlns:xsi=http://www.w3.org/2001/XMLSchema-instance
xmlns:xsd=http://www.w3.org/2001/XMLSchema >

<SOAP-ENV:Body>
<ns1:getTempResponse xmlns:ns1="urn:xmethods-Temperature">
<return xsi:type="xsd:float">85</return>
</ns1:getTempResponse>
</SOAP-ENV:Body></SOAP-ENV:Envelope>
```

**Example:** The following example shows how to insert the result from a web service into a table

```sql
INSERT INTO MYTABLE(XMLCOL) VALUES (DB2XML.SOAPHTTPC('http://www.myserver.com/services/db2sample/list.dadx/SOAP', 'http://tempuri.org/db2sample/list.dadx', '<listDepartments xmlns="http://tempuri.org/db2sample/listdadx">
<deptno>A00</deptno>
</ListDepartments>'))
```

The SOAPHTTPNV and SOAPHTTPNC user-defined functions

Db2 provides SOAPHTTPNV and SOAPHTTPNC user-defined functions that allow you to work with SOAP and consume web services in SQL statements. The user-defined functions are two varieties of SOAPHTTPNV for VARCHAR data and two varieties of SOAPHTTPNC for CLOB data.

The user-defined functions perform the following actions:

1. Post the input SOAP request to the service endpoint
2. Receive and return the SOAP response

SOAPHTTPNV and SOAPHTTPNC allow you to specify a complete SOAP message as input and return complete SOAP messages from the specified web service as a CLOB or VARCHAR representation of the returned XML data. SOAPHTTPNV returns VARCHAR(32672) data and SOAPHTTPNC returns CLOB(1M) data. Both functions accept either VARCHAR(32672) or CLOB(1M) as the input body.

SOAPHTTPNV and SOAPHTTPNC user-defined functions can support SOAP 1.1 or SOAP 1.2. Check with your system administrator to determine which levels of SOAP are supported by the user-defined functions in your environment.

Example

The following example shows how to insert the complete result from a web service into a table using SOAPHTTPNC.

```sql
INSERT INTO EMPLOYEE(XMLCOL)
VALUES (DB2XML.SOAPHTTPNC(
    'http://www.myserver.com/services/db2sample/list.dadx/SOAP',
    'http://tempuri.org/db2sample/list.dadx',
    '<?xml version="1.0" encoding="UTF-8" ?>
    <SOAP-ENV:Envelope
        xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
        xmlns:xsd="http://www.w3.org/2001/XMLSchema"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
    <SOAP-ENV:Body>
        <listDepartments xmlns="http://tempuri.org/db2sample/list.dadx">
        <deptNo>A00</deptNo>
        </listDepartments>
    </SOAP-ENV:Body>
    </SOAP-ENV:Envelope>'))
```

Related tasks

Additional steps for enabling web service user-defined functions (Db2 Installation and Migration)

SQLSTATEs for Db2 as a web services consumer

Db2 returns SQLSTATE values for error conditions that are related to using Db2 as a web services consumer.

The following tables show possible SQLSTATE values.

### Table 131. SQLSTATE values for SOAPHTTPNV and SOAPHTTPNC user-defined functions

<table>
<thead>
<tr>
<th>SQLSTATE</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>38301</td>
<td>An unexpected NULL value was pass as input to the function.</td>
</tr>
<tr>
<td>38302</td>
<td>The function was unable to allocate space.</td>
</tr>
<tr>
<td>38304</td>
<td>An unknown protocol was specified ion the endpoint URL.</td>
</tr>
<tr>
<td>38305</td>
<td>An invalid URL was specified on the endpoint URL.</td>
</tr>
<tr>
<td>38306</td>
<td>An error occurred while attempting to create a TCP/IP socket.</td>
</tr>
<tr>
<td>38307</td>
<td>An error occurred while attempting to bind a TCP/IP socket.</td>
</tr>
</tbody>
</table>
Table 131. SQLSTATE values for SOAPHTTPV and SOAPHTTPC user-defined functions (continued)

<table>
<thead>
<tr>
<th>SQLSTATE</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>38308</td>
<td>The function could not resolve the specified host name.</td>
</tr>
<tr>
<td>38309</td>
<td>An error occurred while attempting to connect to the specified server.</td>
</tr>
<tr>
<td>38310</td>
<td>An error occurred while attempting to retrieve information from the protocol.</td>
</tr>
<tr>
<td>38311</td>
<td>An error occurred while attempting to set socket options.</td>
</tr>
<tr>
<td>38312</td>
<td>The function received unexpected data returned for the web service.</td>
</tr>
<tr>
<td>38313</td>
<td>The web service did not return data of the proper content type.</td>
</tr>
<tr>
<td>38314</td>
<td>An error occurred while initializing the XML parser.</td>
</tr>
<tr>
<td>38315</td>
<td>An error occurred while creating the XML parser.</td>
</tr>
<tr>
<td>38316</td>
<td>An error occurred while establishing a handler for the XML parser.</td>
</tr>
<tr>
<td>38317</td>
<td>The XML parser encountered an error while parsing the result data.</td>
</tr>
<tr>
<td>38318</td>
<td>The XML parser could not convert the result data to the database code page.</td>
</tr>
<tr>
<td>38319</td>
<td>The function could not allocate memory when creating a TCP/IP socket.</td>
</tr>
<tr>
<td>38320</td>
<td>An error occurred while attempting to send the request to the specified server.</td>
</tr>
<tr>
<td>38321</td>
<td>The function was unable to send the entire request to the specified server.</td>
</tr>
<tr>
<td>38322</td>
<td>An error occurred while attempting to read the result data from the specified server.</td>
</tr>
<tr>
<td>38323</td>
<td>An error occurred while waiting for data to be returned from the specified server.</td>
</tr>
<tr>
<td>38324</td>
<td>The function encountered an internal error while attempting to format the input message.</td>
</tr>
<tr>
<td>38325</td>
<td>The function encountered an internal error while attempting to add namespace information to the input message.</td>
</tr>
<tr>
<td>38327</td>
<td>The XML parser could not strip the SOAP envelope from the result message.</td>
</tr>
<tr>
<td>38328</td>
<td>An error occurred while processing an SSL connection.</td>
</tr>
</tbody>
</table>

Table 132. SQLSTATE values for SOAPHTTPNV and SOAPHTTPNC user-defined functions

<table>
<thead>
<tr>
<th>SQLSTATE</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>38350</td>
<td>An unexpected NULL value was specified for the endpoint, action, or SOAP input.</td>
</tr>
<tr>
<td>38351</td>
<td>A dynamic memory allocation error.</td>
</tr>
<tr>
<td>38352</td>
<td>An unknown or unsupported transport protocol.</td>
</tr>
<tr>
<td>38353</td>
<td>An invalid URL was specified.</td>
</tr>
<tr>
<td>38354</td>
<td>An error occurred while resolving the hostname.</td>
</tr>
<tr>
<td>38355</td>
<td>A memory exception for socket.</td>
</tr>
<tr>
<td>38356</td>
<td>An error occurred during socket connect.</td>
</tr>
<tr>
<td>38357</td>
<td>An error occurred while setting socket options.</td>
</tr>
<tr>
<td>38358</td>
<td>An error occurred during input/output control (ioctl) to verify HTTPS enablement.</td>
</tr>
<tr>
<td>38359</td>
<td>An error occurred while reading from the socket.</td>
</tr>
</tbody>
</table>
Table 132. SQLSTATE values for SOAPHTTPNV and SOAPHTTPNC user-defined functions (continued)

<table>
<thead>
<tr>
<th>SQLSTATE</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>38360</td>
<td>An error occurred due to socket timeout.</td>
</tr>
<tr>
<td>38361</td>
<td>No response from the specified host.</td>
</tr>
<tr>
<td>38362</td>
<td>An error occurred due to an unexpected HTTP return or content type</td>
</tr>
<tr>
<td>38363</td>
<td>The TCP/IP stack was not enabled for HTTPS.</td>
</tr>
</tbody>
</table>

**Related tasks**

Additional steps for enabling web service user-defined functions (Db2 Installation and Migration)
Chapter 8. Application compatibility levels in Db2

The application compatibility level of your applications controls the adoption and use of new capabilities and enhancements, and the impact of incompatible changes. The advantage is that you can complete the Db2 12 migration process without the need to update your applications immediately.

After function level 500 or higher is activated, you can continue run applications with the features and behavior of previous versions or specific Db2 12 function levels.

You can change the application compatibility level for each application when you are ready for it to run with the features and behavior of a higher Db2 version or function level. The application compatibility level applies to most SQL statements, including data definition statements (such as CREATE and ALTER statements) and data control statements (such as GRANT and REVOKE statements).

The application compatibility of a package is initially set when you bind a package, based on the following values:

1. The APPLCOMPAT bind option value, if specified.
2. If the bind option is omitted, the APPLCOMPAT subsystem parameter.

For static SQL statements, the APPLCOMPAT column of the SYSIBM.SYSPACKAGE catalog table stores the application compatibility setting. This setting changes for the following reasons:

- You issue a REBIND command for the package and specify a different value for the APPLCOMPAT option. If you omit this option, the previous value for the package is used. If no previous value is available (such as for packages last bound before the introduction of application compatibility) the APPLCOMPAT subsystem parameter value is used.
- An automatic bind of the package occurs. The application compatibility is set to the previous value. If no previous value is available, the APPLCOMPAT subsystem parameter value is used.

For dynamic SQL statements, the CURRENT APPLICATION COMPATIBILITY special register stores the application compatibility setting. This setting changes for the following reasons:

- The special register is initialized to the application compatibility of the package, as described above.
- During execution of the package, SET CURRENT APPLICATION COMPATIBILITY statements can change the special register. The value must be equivalent to the APPLCOMPAT bind option for the package or lower, if the value is V12R1M500 or above.

For new installations, the default APPLCOMPAT subsystem parameter value is V12R1M500. However, you can specify a higher value. For migrated environments, the default value is the value from the migration input member.

Tip: When you migrate to Db2 12, or activate any higher function level, change the APPLCOMPAT subsystem parameter value only after all applications can use the SQL capabilities of Db2 12 or the higher function level. For details, see “Enabling default application compatibility with function level 500 or higher” on page 832.

Supported application compatibility levels

Db2 12 supports the following application compatibility levels in most contexts:

VvvRrMmmm Compatibility with the behavior of the identified Db2 function level. For example, V12R1M510 specifies compatibility the highest available function level. The equivalent function level or higher must be activated. For a list of supported Db2 12 function levels, see Db2 12 function levels (Db2 for z/OS What's New?).

V12R1 Compatibility with the behavior of Db2 12 function level 500. This value this value has the same result as specifying V12R1M500.
V11R1
Compatibility with the behavior of Db2 11 new-function mode. After migration to Db2 12, this value has the same result as specifying V12R1M100.

V10R1
Compatibility with the behavior of DB2 10 new-function mode.

Example: V10R1 application compatibility
The following example shows the results of using a capability that is introduced in the application compatibility level V11R1, with application compatibility level set to V10R1. Assume that the APPLCOMPAT subsystem parameter value is V10R1. The example CREATE PROCEDURE statement does not specify the APPLCOMPAT keyword. In this example the CREATE TYPE statement is successful but the CREATE PROCEDURE statement results in SQLCODE -4743.

```
CREATE TYPE PHONENUMBERS AS VARCHAR(12) ARRAY ??(1000000??)

DSNT400I SQLCODE = 000, SUCCESSFUL EXECUTION

CREATE PROCEDURE FIND_CUSTOMERS(
  IN NUMBERS_IN KRAMSC01.PHONENUMBERS,
  IN AREA_CODE CHAR(3),
  OUT NUMBERS_OUT KRAMSC01.PHONENUMBERS)
BEGIN
  SET NUMBERS_OUT =
    (SELECT ARRAY_AGG(T.NUM)
    FROM UNNEST(NUMBERS_IN) AS T(NUM)
    WHERE SUBSTR(T.NUM, 1, 3) = AREA_CODE);
END

DSNT408I SQLCODE = -4743, ERROR: ATTEMPT TO USE A FUNCTION WHEN THE APPLICATION COMPATIBILITY SETTING IS SET FOR A PREVIOUS LEVEL

The APPLCOMPAT bind option value for the CREATE PROCEDURE statement is then set to V11R1 or higher and result of the statement is then successful.

CREATE PROCEDURE FIND_CUSTOMERS(
  IN NUMBERS_IN KRAMSC01.PHONENUMBERS,
  IN AREA_CODE CHAR(3),
  OUT NUMBERS_OUT KRAMSC01.PHONENUMBERS)
APPLCOMPAT V11R1
BEGIN
  SET NUMBERS_OUT =
    (SELECT ARRAY_AGG(T.NUM)
    FROM UNNEST(NUMBERS_IN) AS T(NUM)
    WHERE SUBSTR(T.NUM, 1, 3) = AREA_CODE);
END

DSNT400I SQLCODE = 000, SUCCESSFUL EXECUTION
```

Related concepts
Function levels and related levels in Db2 12 (Db2 for z/OS What's New?)

Related tasks
Controlling Db2 application compatibility (Db2 for z/OS What's New?)

Related reference
APPLCOMPAT bind option (Db2 Commands)
CURRENT APPLICATION COMPATIBILITY (Db2 SQL)
APPL COMPAT LEVEL field (APPLCOMPAT subsystem parameter) (Db2 Installation and Migration)
SYSPACKAGE catalog table (Db2 SQL)
SET CURRENT APPLICATION COMPATIBILITY (Db2 SQL)
-ACTIVATE (Db2) (Db2 Commands)
In Db2 12, you can use the application compatibility level to control the adoption of new SQL capabilities and enhancements of particular function levels.

You can use the application compatibility level of applications, and objects such as routines or triggers, to control the adoption and use of new and changed SQL capabilities that are introduced in function levels. Generally, applications, and routines or triggers, cannot use new or changed SQL capabilities unless the effective application compatibility level is equivalent to or higher than the function level that introduced the changes. The application compatibility level applies to most SQL statements, including data definition statements (such as CREATE and ALTER statements) and data control statements (such as GRANT and REVOKE statements).

The corresponding function level or higher must be activated when you bind packages at an application compatibility level. However, if you activate a lower function level (or * function level), applications can continue to run with the higher application compatibility level. To prevent the continued use of SQL capabilities introduced in the higher function level, you must also modify the application and change the effective application compatibility level to the lower level.

For IBM data server clients or drivers that need to exploit Db2 for z/OS capabilities that are delivered with function level V12R1M501 or greater, you need to take additional steps. See “V12R1Mnnn application compatibility levels for data server clients and drivers” on page 819 for details.

Tip: Do not raise the default application compatibility level of the Db2 subsystem immediately after migrating or activating a new function level. Instead, wait until applications have been verified to work correctly at the higher function level, and any incompatibilities have been resolved. For details, see “Enabling default application compatibility with function level 500 or higher” on page 832.

Application compatibility levels are specified in commands and message output by nine-character strings that correspond to the Db2 version, release, and modification value of the corresponding function level. See the activation details for each function level for a summary the new features that are controlled by the corresponding application compatibility level.

For example, V12R1M510 specifies compatibility the highest available function level. The equivalent function level or higher must be activated. For a list of supported Db2 12 function levels, see Db2 12 function levels (Db2 for z/OS What's New?).

Related reference
APPLCOMPAT bind option (Db2 Commands)
APPL COMPAT LEVEL field (APPLCOMPAT subsystem parameter) (Db2 Installation and Migration)
CURRENT APPLICATION COMPATIBILITY (Db2 SQL)

IBM data server clients and drivers that use Db2 for z/OS capabilities with a function level requirement of greater than V12R1M500 require extra program preparation steps.

Before you begin
Take these actions on the clients that connect to your Db2 for z/OS system:

• Determine whether you need to upgrade the data server clients or drivers to support V12R1M501 application compatibility:
  - If your applications include function that requires a minimum application compatibility level of V12R1M501, you need to upgrade to Db2 Connect Version 11.1 Modification 2 Fix Pack 2 or later.
  - If your applications include function that requires a minimum application compatibility level of V12R1M500 or earlier, you can use any Db2 Connect version.
The minimum data server client or driver levels for exploitation of application compatibility of V12R1M501 or later are:

- IBM Data Server Driver for JDBC and SQLJ: Versions 3.72 and 4.23, or later. For information on the driver versions that are delivered with each Db2 Connect version, see IBM Data Server Driver for JDBC and SQLJ levels and Db2 on Linux, UNIX, and Windows systems or Db2 Connect versions (Db2 Application Programming for Java).

- Other IBM data server clients and drivers: Db2 for Linux, UNIX, and Windows, Version 11.1 Modification 2 Fix Pack 2, or later.

- Run the db2connectactivate utility to activate the Version 11.1 license certificate file for Db2 Connect Unlimited Edition on Db2 for z/OS. Specify the options to bind the driver and client packages into the NULLID collection, with APPLCOMPAT V12R1M500. For example:

```
db2connectactivate.sh -host sys1.svl.ibm.com -port 5021 -database STLEC1 -user dbadm -password dbadmpass -bindoptions "APPLCOMPAT V12R1M500" -collection NULLID
```

For more information, see:

Activating the license key for Db2 Connect Unlimited Edition (IBM Z)
db2connectactivate - Server license activation utility

About this task

This procedure sets the V12R1Mnnn application compatibility for a client or driver that needs to utilize server capabilities that require an application compatibility of greater than V12R1M500. If the client or driver does not utilize the capabilities of a new function level, you do not need to update its application compatibility settings.

Procedure

1. Set the Db2 for z/OS server application compatibility (APPLCOMPAT) level.
   a) For client applications that contain static SQL statements, rebind the static application packages with the new APPLCOMPAT value, on the client and on the server.
   b) For client applications that contain only dynamic SQL statements, bind or rebind the client or driver packages with the new APPLCOMPAT value, on the server.

   Tip: Binding package copies and keeping the original driver packages lets you access new capabilities for applications that need them, while ensuring stability for applications that should not be exposed to incompatibilities.

   For drivers only, you can use jobs that are provided with Db2 for z/OS, in data set prefix.SDSNSAMP, to bind or rebind the driver packages on the server. To run those jobs, follow these steps:
   i) Customize jobs DSNTIJLC and DSNTIJLR, using the instructions in the job prologs.
   ii) If there is a possibility that you still need to run applications under a driver that is at the old application compatibility level, run DSNTIJLC to bind copies of the driver packages at the new application compatibility level, while leaving the packages in the NULLID collection at the old application compatibility level. In most cases, you should do this.

   If you are sure that you do not need to run applications under a driver at the old application compatibility level, run job DSNTIJLR to rebind the client or driver packages at the new application compatibility level.

   iii) If you bound copies of the driver packages for the new function level, switch the drivers to the new function level by modifying the property that controls the current package set to match the collection ID of the new package copies.

   • For CLI or ODBC drivers, change the CLI/ODBC CurrentPackageSet configuration keyword value.
• For the IBM Data Server Driver for JDBC and SQLJ, change the
  `DB2BaseDataSource.currentPackageSet Connection` or `DataSource` property value.

2. Set the client application compatibility value to control the capabilities of client applications when a
   client or driver contains changes that enable new server capabilities. If you set the client application
   compatibility level, its value must be less than or equal to the server application compatibility level.

Take one of the following actions to set the client application compatibility value.

• For CLI or ODBC drivers or IBM Data Server clients, change the CLI/ODBC `ClientApplCompat`
  configuration keyword value.

Do this by adding a line similar to this example to the `<databases>` section or the `<dsn>` section in
the `db2dsdriver.cfg` file.

```xml
<parameter name="clientApplCompat" value="V12R1M501"/>
```

For more information, see:

- IBM data server driver configuration file
- Installing the IBM Data Server Driver Package software on the Linux and UNIX operating systems
  (includes information on creating and populating the `db2dsdriver.cfg` file)
- ClientApplCompat IBM data server driver configuration keyword

• For the IBM Data Server Driver for JDBC and SQLJ, change the
  `DB2BaseDataSource.clientApplcompat Connection` or `DataSource` property value.

Related reference

- IBM Data Server Driver for JDBC and SQLJ properties for Db2 for z/OS (Db2 Application Programming for
  Java)
- DISPLAY LOCATION (Db2) (Db2 Commands)
- ACTIVATE (Db2) (Db2 Commands)

Related information

- 30025 (Db2 Codes)
- DSNL200I (Db2 Messages)

DSNTIJLC

Migrate Db2 Connect packages to support a new function level.

```xml
<parameter name="clientApplCompat" value="V12R1M501"/>
```
**bind from copy. The DB2-supplied setting is 'NULLID_V12R1M500'. Use the SET TGTCOLID statement in job step DSNTIRU to specify a different setting.**

**&SRCCOLID is the name of the collection-ID to copy from. The DB2-supplied setting is 'NULLID'. Use the SET SRCCOLID statement in job step DSNTIRU to specify a different setting.**

**&APPLCMPT is the DB2 application compatibility level. The DB2-supplied setting is 'V12R1M500'. Use the SET APPLCMPT statement in job step DSNTIRU to specify a different setting.**

**Attention JES3 users: Symbolic substitution within in-stream data sets in JES3 requires z/OS 2.2 or above. In order to run this job on JES2 using z/OS 2.1, you need to make the following manual changes: (a) Remove the EXPORT SYMLIST and all SET statements (b) Change all occurrences of &TGTCOLID to the name of the collection-ID to bind from copy. (c) Change all occurrences of &SRCCOLID to the name of the collection-ID to copy from. (d) Change all occurrences of &APPLCMPT to the DB2 application compatibility setting.**

**Before running this job, customize it as follows: (a) Add a valid job card. (b) Locate and change all occurrences of the following strings as indicated: - '!DSN!' to the name of the DB2 subsystem. - 'DSN!!0' to the prefix of the DB2 target libraries for the DB2 subsystem. (c) Set TGTCOLID, SRCCOLID, and APPLCMPT as described above.**

**CHANGE LOG =**

**11/08/2016 Job created S28617 PI74456**

**JOBLIB DD DISP=SHR,**

**DSN=DSN!!0.SDSNLOAD**

**Symbolic substitution requires z/OS 2.2, or z/OS 2.1 with JES2.**

**EXPORT SYMLIST=(TGTCOLID,SRCCOLID,APPLCMPT)**

**SET TGTCOLID='NULLID_V12R1M500'**

**SET SRCCOLID='NULLID'**

**SET APPLCMPT='V12R1M500'**

**DSNTIRU EXEC PGM=IKJEFT01,DYNAMNBR=20,COND=(4,LT)**

**SYSTSPRT DD SYSOUT=***

**SYSPRINT DD SYSOUT=**

**SYSDUMP DD SYSOUT=**

**SYSTSIN DD *,SYMBOLS=JCLONLY**

**DSN SYSTEM(!DSN!)**

**BIND PACKAGE (&TGTCOLID) +**

**COPY(&SRCCOLID..SYSLH100) +**

**APPLCOMPAT(&APPLCMPT)**

**BIND PACKAGE (&TGTCOLID) +**

**COPY(&SRCCOLID..SYSLH101) +**

**APPLCOMPAT(&APPLCMPT)**

**BIND PACKAGE (&TGTCOLID) +**

**COPY(&SRCCOLID..SYSLH102) +**

**APPLCOMPAT(&APPLCMPT)**

**BIND PACKAGE (&TGTCOLID) +**

**COPY(&SRCCOLID..SYSLH200) +**

**APPLCOMPAT(&APPLCMPT)**

**BIND PACKAGE (&TGTCOLID) +**

**COPY(&SRCCOLID..SYSLH201) +**

**APPLCOMPAT(&APPLCMPT)**

**BIND PACKAGE (&TGTCOLID) +**

**COPY(&SRCCOLID..SYSLH202) +**

**APPLCOMPAT(&APPLCMPT)**

**BIND PACKAGE (&TGTCOLID) +**

**COPY(&SRCCOLID..SYSLH300) +**

**APPLCOMPAT(&APPLCMPT)**

**BIND PACKAGE (&TGTCOLID) +**

**COPY(&SRCCOLID..SYSLH301) +**

**APPLCOMPAT(&APPLCMPT)**

**BIND PACKAGE (&TGTCOLID) +**

**COPY(&SRCCOLID..SYSLH302) +**

**APPLCOMPAT(&APPLCMPT)**

**BIND PACKAGE (&TGTCOLID) +**

822
DSNTIJLR

Migrate Db2 Connect packages to support a new function level.

/****************************************************************************
  * JOB NAME = DSNTIJLR
  */
  /* DESCRIPTIVE NAME = INSTALLATION JOB STREAM
  */
  /*  Licensed Materials - Property of IBM
  */
  /*  5650-DB2
  */
  /*  (C) COPYRIGHT 2016 IBM Corp.All Rights Reserved.
  */
  /*
  * FUNCTION = Migrate DB2 Connect packages to support a new
  * function level.
  *
  */
  /**
  * PSEUDOCODE =
  */
  /** DSNTIRU STEP Rebind the IBM JDBC and CLI standard
  */
  /** set of packages to a new collection in
  */
  /** order to override the APPLCOMPAT package
  */
  /** option.
  */
  /*
  */
  /* NOTES =
  */
  /* (1) This job includes an in-stream data set having
  */
  /* DB2 bind statements that contain substitution
  */
  /** symbols. For example:
  */
  /*  REBIND PACKAGE (&SRCCOLID..SYSLH100) +
  */
  /*  APPLCOMPAT(&APPLCMPT)
  */
  /*  where
  */
  /*  &SRCCOLID is the name of the collection-ID
  */
  /* owning the package to be rebound.
  */
  /*  The DB2-supplied setting is
  */
  /*  'NULLID_V12R1M500'. Use the SET
  */
  /*  SRCCOLID statement in job step
  */
  /*  DSNTIRU to specify a different
  */
  /*  setting.
  */
  /*  &APPLCMPT is the DB2 application compatibility
  */
}
Attention JES3 users: Symbolic substitution within in-stream data sets in JES3 requires z/OS 2.2 or above. In order to run this job on JES2 using z/OS 2.1, you need to make the following manual changes:
(a) Remove the EXPORT SYMLIST and all SET statements
(b) Change all occurrences of &SRCCOLID to the name of the collection-ID owning the package to be rebound
(c) Change all occurrences of &APPLCMPT to the DB2 application compatibility setting.

(2) Before running this job, customize it as follows:
(a) Add a valid job card.
(b) Locate and change all occurrences of the following strings as indicated:
   - '!DSN!' to the name of the DB2 subsystem.
   - 'DSN!!0' to the prefix of the DB2 target libraries for the DB2 subsystem.
(c) Set SRCCOLID and APPLCMPT as described above.

CHANGE LOG =
11/08/2016 Job created S28617 PI74456

JOBLIB DD DISP=SHR,
      DSN=DSN!!0.SDSNLOAD

Symbolic substitution requires z/OS 2.2, or z/OS 2.1 with JES2.
EXPORT SYMLIST=(SRCCOLID,APPLCMPT)
SET SRCCOLID='NULLID_V12R1M500'
SET APPLCMPT='V12R1M500'
DSNTIRU EXEC PGM=IKJEFT01,DYNAMNBR=20,COND=(4,LT)
SYSTSPRT DD SYSOUT=* 
SYSPRINT DD SYSOUT=* 
SYSUDUMP DD SYSOUT=* 
SYSTSIN DD *,SYMBOLS=JCLONLY
DSN SYSTEM(DB2A)
REBIND PACKAGE (&SRCCOLID..SYSLH100) APPLCOMPAT(&APPLCMPT)
REBIND PACKAGE (&SRCCOLID..SYSLH101) APPLCOMPAT(&APPLCMPT)
REBIND PACKAGE (&SRCCOLID..SYSLH102) APPLCOMPAT(&APPLCMPT)
REBIND PACKAGE (&SRCCOLID..SYSLH200) APPLCOMPAT(&APPLCMPT)
REBIND PACKAGE (&SRCCOLID..SYSLH201) APPLCOMPAT(&APPLCMPT)
REBIND PACKAGE (&SRCCOLID..SYSLH202) APPLCOMPAT(&APPLCMPT)
REBIND PACKAGE (&SRCCOLID..SYSLH300) APPLCOMPAT(&APPLCMPT)
REBIND PACKAGE (&SRCCOLID..SYSLH301) APPLCOMPAT(&APPLCMPT)
REBIND PACKAGE (&SRCCOLID..SYSLH302) APPLCOMPAT(&APPLCMPT)
REBIND PACKAGE (&SRCCOLID..SYSLH400) APPLCOMPAT(&APPLCMPT)
REBIND PACKAGE (&SRCCOLID..SYSLH401) APPLCOMPAT(&APPLCMPT)
REBIND PACKAGE (&SRCCOLID..SYSLH402) APPLCOMPAT(&APPLCMPT)
REBIND PACKAGE (&SRCCOLID..SYSLH403) APPLCOMPAT(&APPLCMPT)
REBIND PACKAGE (&SRCCOLID..SYSLH404) APPLCOMPAT(&APPLCMPT)
REBIND PACKAGE (&SRCCOLID..SYSLH405) APPLCOMPAT(&APPLCMPT)
REBIND PACKAGE (&SRCCOLID..SYSLH406) APPLCOMPAT(&APPLCMPT)
REBIND PACKAGE (&SRCCOLID..SYSLH407) APPLCOMPAT(&APPLCMPT)
REBIND PACKAGE (&SRCCOLID..SYSLH408) APPLCOMPAT(&APPLCMPT)
REBIND PACKAGE (&SRCCOLID..SYSLH409) APPLCOMPAT(&APPLCMPT)
REBIND PACKAGE (&SRCCOLID..SYSLH410) APPLCOMPAT(&APPLCMPT)
REBIND PACKAGE (&SRCCOLID..SYSLH411) APPLCOMPAT(&APPLCMPT)
REBIND PACKAGE (&SRCCOLID..SYSLH412) APPLCOMPAT(&APPLCMPT)
REBIND PACKAGE (&SRCCOLID..SYSLH413) APPLCOMPAT(&APPLCMPT)
REBIND PACKAGE (&SRCCOLID..SYSLH414) APPLCOMPAT(&APPLCMPT)
REBIND PACKAGE (&SRCCOLID..SYSLH415) APPLCOMPAT(&APPLCMPT)
REBIND PACKAGE (&SRCCOLID..SYSLH416) APPLCOMPAT(&APPLCMPT)
REBIND PACKAGE (&SRCCOLID..SYSLH417) APPLCOMPAT(&APPLCMPT)
REBIND PACKAGE (&SRCCOLID..SYSLH418) APPLCOMPAT(&APPLCMPT)
REBIND PACKAGE (&SRCCOLID..SYSLH419) APPLCOMPAT(&APPLCMPT)
REBIND PACKAGE (&SRCCOLID..SYSLH420) APPLCOMPAT(&APPLCMPT)
REBIND PACKAGE (&SRCCOLID..SYSLH421) APPLCOMPAT(&APPLCMPT)
REBIND PACKAGE (&SRCCOLID..SYSLH422) APPLCOMPAT(&APPLCMPT)
REBIND PACKAGE (&SRCCOLID..SYSLH423) APPLCOMPAT(&APPLCMPT)
REBIND PACKAGE (&SRCCOLID..SYSLH424) APPLCOMPAT(&APPLCMPT)
REBIND PACKAGE (&SRCCOLID..SYSLH425) APPLCOMPAT(&APPLCMPT)
REBIND PACKAGE (&SRCCOLID..SYSLH426) APPLCOMPAT(&APPLCMPT)
REBIND PACKAGE (&SRCCOLID..SYSLH427) APPLCOMPAT(&APPLCMPT)
REBIND PACKAGE (&SRCCOLID..SYSLH428) APPLCOMPAT(&APPLCMPT)
REBIND PACKAGE (&SRCCOLID..SYSLH429) APPLCOMPAT(&APPLCMPT)
REBIND PACKAGE (&SRCCOLID..SYSLH430) APPLCOMPAT(&APPLCMPT)
REBIND PACKAGE (&SRCCOLID..SYSLH431) APPLCOMPAT(&APPLCMPT)
REBIND PACKAGE (&SRCCOLID..SYSLH432) APPLCOMPAT(&APPLCMPT)
REBIND PACKAGE (&SRCCOLID..SYSLH433) APPLCOMPAT(&APPLCMPT)
REBIND PACKAGE (&SRCCOLID..SYSLH434) APPLCOMPAT(&APPLCMPT)
REBIND PACKAGE (&SRCCOLID..SYSLH435) APPLCOMPAT(&APPLCMPT)
REBIND PACKAGE (&SRCCOLID..SYSLH436) APPLCOMPAT(&APPLCMPT)
REBIND PACKAGE (&SRCCOLID..SYSLH437) APPLCOMPAT(&APPLCMPT)
REBIND PACKAGE (&SRCCOLID..SYSLH438) APPLCOMPAT(&APPLCMPT)
REBIND PACKAGE (&SRCCOLID..SYSLH439) APPLCOMPAT(&APPLCMPT)
REBIND PACKAGE (&SRCCOLID..SYSLH440) APPLCOMPAT(&APPLCMPT)
REBIND PACKAGE (&SRCCOLID..SYSLH441) APPLCOMPAT(&APPLCMPT)
REBIND PACKAGE (&SRCCOLID..SYSLH442) APPLCOMPAT(&APPLCMPT)
REBIND PACKAGE (&SRCCOLID..SYSLH443) APPLCOMPAT(&APPLCMPT)
REBIND PACKAGE (&SRCCOLID..SYSLH444) APPLCOMPAT(&APPLCMPT)
REBIND PACKAGE (&SRCCOLID..SYSLH445) APPLCOMPAT(&APPLCMPT)
REBIND PACKAGE (&SRCCOLID..SYSLH446) APPLCOMPAT(&APPLCMPT)
REBIND PACKAGE (&SRCCOLID..SYSLH447) APPLCOMPAT(&APPLCMPT)
REBIND PACKAGE (&SRCCOLID..SYSLH448) APPLCOMPAT(&APPLCMPT)
REBIND PACKAGE (&SRCCOLID..SYSLH449) APPLCOMPAT(&APPLCMPT)
REBIND PACKAGE (&SRCCOLID..SYSLH450) APPLCOMPAT(&APPLCMPT)

Chapter 8. Application compatibility levels in Db2 825
Using profile tables to control which Db2 for z/OS application compatibility levels to use for specific data server client applications

Profiles can be used to control which client applications use features that are associated with a specific Db2 for z/OS application compatibility level. This capability allows client applications that do not need to use new features to continue to connect to a Db2 for z/OS server at an earlier application compatibility level.

About this task

In this example procedure, client application ACCTG_APP501 needs to use Db2 for z/OS capabilities that are available at application compatibility level V12R1M501. All other client applications need to use capabilities that are available at application compatibility level V12R1M500 or lower.

Procedure

1. On the client operating system, bind the client driver packages into two collections:
   - One collection with the default collection name NULLID, and with the APPLCOMPAT option set to V12R1M500. If you have already bound the client driver packages into collection NULLID with APPLCOMPAT set to V12R1M500, you do not need to bind them again.
   - Another collection with a different name, such as NULLID_NF, and with the APPLCOMPAT option set to V12R1M501.
   - For the IBM Data Server Driver for JDBC and SQLJ, follow these steps to bind the driver packages:
     - If you have not already done so, invoke the DB2Binder utility with a control statement like this one to create a collection with the default name NULLID, and with application compatibility set to V12R1M500:

       ```
       ```

     - Invoke the DB2Binder utility with a control statement like this one to create a collection named NULLID_NF, with application compatibility set to V12R1M501:

       ```
       ```

   - For the IBM Data Server Driver for ODBC and CLI, follow these steps to bind the driver packages:
     - If CLI/ODBC configuration keyword OnlyUseBigPackages=1, you do not need to bind the driver packages.
     - If CLI/ODBC configuration keyword OnlyUseBigPackages=0, you need to bind small packages with application compatibility set to V12R1M501:

       ```
       db2 bind 'DB2PATH\bnd\idds2sys.lst' blocking all sqlerror continue grant public action replace collection NULLID_NF generic "APPLCOMPAT V12R1M501"
       ```
2. In Db2 for z/OS, create a profile for client application ACCTG_APP501 by inserting rows into tables SYSIBM.DSN_PROFILE_TABLE and SYSIBM.DSN_PROFILE_ATTRIBUTES. The profile directs Db2 to use the driver with packages in collection NULLID_NF when application ACCTG_APP501 runs. Because the driver packages in the NULLID_NF collection are bound with option APPLCOMPAT V12R1M501, ACCTG_APP501 can use capabilities that are available at application compatibility level V12R1M501.

```sql
INSERT INTO SYSIBM.DSN_PROFILE_TABLE
(PROFILEID, CLIENT_APPLNAME, PROFILE_ENABLED)
VALUES (1002, 'ACCTG_APP501', 'Y');
INSERT INTO SYSIBM.DSN_PROFILE_ATTRIBUTES
(PROFILEID, KEYWORDS, ATTRIBUTE1)
VALUES (1002, 'SPECIAL_REGISTER', 'SET CURRENT PACKAGE PATH=NULLID_NF');
```

**Important:** Although PKGNAME can be used as a filtering category for profile table rows that use the 'SPECIAL_REGISTER' value for KEYWORDS, when client drivers are used, you should not use PKGNAME alone or in combination with COLLID.

3. On Db2 for z/OS, issue the -START PROFILE command to load the updated profile tables into memory.

**Results**

Application ACCTG_APP501 can now successfully connect to Db2 for z/OS and use data server driver packages in collection NULLID_NF. All other applications can connect to Db2 for z/OS and use data server driver packages in collection NULLID.

You can verify the application compatibility level and collection that are being used for your client application by adding code to execute a query like this to your application.

```sql
SELECT CURRENT APPLICATION COMPATIBILITY,
GETVARIABLE('SYSIBM.PACKAGE_SCHEMA')
FROM SYSIBM.SYSDUMMY1
```

For example, you might add code like this to a Java application:

```java
String currApplcompat, appCollection;
Connection con;
Statement stmt;
ResultSet rs;
...
stmt = con.createStatement();        // Create a Statement object.
rs = stmt.executeQuery("SELECT CURRENT APPLICATION COMPATIBILITY," +
" GETVARIABLE('SYSIBM.PACKAGE_SCHEMA')" +
" FROM SYSIBM.SYSDUMMY1");           // Get the result table from the query
while (rs.next()) {                  // Position the cursor
  currApplcompat = rs.getString(1);   // Retrieve the application compatibility
  System.out.println("APPLCOMPAT = " + currApplcompat);
  System.out.println("COLLID = " + appCollection);  // Print the collection name
}
```

For application ACCTG_APP501, the query should return a value of V12R1M501 for the current application compatibility, and NULLID_NF for the collection name.

**Related concepts**

Binding database utilities on Db2 Connect

**Related tasks**

V12R1Mnnn application compatibility levels for data server clients and drivers

IBM data server clients and drivers that use Db2 for z/OS capabilities with a function level requirement of greater than V12R1M500 require extra program preparation steps.

Setting special registers by using profile tables (Db2 Administration Guide)

**Related reference**

DB2Binder utility (Db2 Application Programming for Java)

OnlyUseBigPackages CLI/ODBC and IBM data server driver configuration keyword
**V11R1 application compatibility level**

When you set the application compatibility level to V11R1, applications that attempt to use functions and features that are introduced in Db2 12 or later might behave differently or receive an error.

When new function is activated in your Db2 12 environment, you can run individual applications with some of the features and behavior of Db2 11. Your applications can continue to experience V11R1 behavior after new function is activated in Db2 12. Then, you can migrate each application to a new application compatibility value separately until all are migrated. If application compatibility level is set to V11R1 and you attempt to use the new functions of a later version, SQL might behave differently or result in a negative SQLCODE, such as SQLCODE -4743.

You can run package level accounting or monitor traces with IFCID 0239 and review field QPACINCOMPAT, which indicates an SQL incompatible change. If a trace is started for IFCID 0376, and application compatibility is set for a previous version, details about features and functions that have a change in behavior are written in field QW0376FN.

A migrated Db2 12 environment behaves with V11R1 application compatibility until function level 500 or higher is activated.

The following table shows many of the features and functions that are controlled by application compatibility, and the results if you specify V11R1. If a behavior difference is traced, then the IFCID trace function code is shown.

<table>
<thead>
<tr>
<th>Feature or Function</th>
<th>Result with V11R1 application compatibility</th>
<th>IFCID 0376 trace function code</th>
</tr>
</thead>
<tbody>
<tr>
<td>The POWER built-in function returns a result with the DOUBLE data type. The result is out of range.</td>
<td>SQLCODE -802</td>
<td>1201</td>
</tr>
<tr>
<td>CURRENT_SERVER or CURRENT_TIMEZONE is used as a column name or variable name.</td>
<td>SQLCODE -206</td>
<td>1204</td>
</tr>
</tbody>
</table>

**Related concepts**

Application and SQL release incompatibilities (Db2 for z/OS What's New?)

---

**V10R1 application compatibility level**

When you set the application compatibility level to V10R1, applications that attempt to use functions and features that are introduced in Db2 11 or later might behave differently or receive an error.

In Db2 12, you can continue to run individual applications with some of the features and behavior of DB2 10. Your applications can continue to experience V10R1 behavior while in Db2 12, regardless of whether new function is activated. Then, you can migrate each application to a new application compatibility value separately until all are migrated. If application compatibility is set to V10R1 and you attempt to use the new functions of a later version, SQL might behave differently or result in a negative SQLCODE, such as SQLCODE -4743.

You can run package level accounting or monitor traces with IFCID 0239 and review field QPACINCOMPAT, which indicates an SQL incompatible change. If a trace is started for IFCID 0376, and application compatibility is set for a previous version, details about features and functions that have a change in behavior are written in field QW0376FN.
A migrated Db2 12 environment behaves with V11R1 application compatibility until function level 500 or higher is activated. Application and SQL incompatibilities are described in the migration information for each version.

The following table shows many of the features and functions that are controlled by application compatibility, and the results if you specify V10R1. If a behavior difference is traced, then the IFCID trace function code is shown.

<table>
<thead>
<tr>
<th>Feature or Function</th>
<th>Result with V10R1 application compatibility</th>
<th>IFCID 0376 trace function code</th>
</tr>
</thead>
<tbody>
<tr>
<td>An SQL statement in a client application includes an unsupported conversion (from a string type to a numeric type or from a numeric type to a string type), and implicit casting is disabled (DDF_COMPATIBILITY is set to SP_PARMS_NJV or to DISABLE_IMPCAST_NJV).</td>
<td>SQLCODE -301</td>
<td>7&quot;1&quot; on page 831</td>
</tr>
<tr>
<td>A client application executes an SQL CALL statement to execute a Db2 for z/OS stored procedure. The DDF_COMPATIBILITY subsystem parameter is set to SP_PARMS_NJV for client applications other than Java applications, or SP_PARMS_JV for Java applications.</td>
<td>The data types of the data that is returned from the SQL CALL statement match the data types of the CALL statement arguments. This behavior is compatible with the behavior before Version 10.</td>
<td>8&quot;1&quot; on page 831</td>
</tr>
<tr>
<td>A client application accesses Db2 11 from an IBM Data Server Driver for JDBC and SQLJ client. The DDF_COMPATIBILITY subsystem parameter is set to IGNORE_TZ for Java applications.</td>
<td>The Db2 server ignores the TIMEZONE portion, appended by the IBM Data Server Driver for JDBC and SQLJ, of the value in the TIMESTAMP WITH TIMEZONE input to a TIMESTAMP target. This behavior is compatible with the behavior before DB2 10.</td>
<td>9</td>
</tr>
<tr>
<td>BIF_COMPATIBILITY is set to V9_TRIM, and input string-expression is EBCDIC mixed data for the RTRIM, LTRIM, or STRIP built-in function.</td>
<td>The DB2 9 version of SYSIBM.LTRIM(string-expression), SYSIBM.RTRIM(string-expression), or SYSIBM.STRIP(string-expression) is executed.</td>
<td>10</td>
</tr>
<tr>
<td>An implicit insert or update of an XML document node</td>
<td>SQLCODE -20345</td>
<td>1101</td>
</tr>
<tr>
<td>A predicate expression with an explicit cast or an operation with an invalid value that does not affect the results of XPath processing</td>
<td>SQLCODE -20345</td>
<td>1102</td>
</tr>
<tr>
<td>How the resource limit facility uses ASUTIME value for nested routines</td>
<td>SQLCODE -905 is issued only when the ASUTIME limit of the top-level calling package is encountered.</td>
<td>1103</td>
</tr>
<tr>
<td>Feature or Function</td>
<td>Result with V10R1 application compatibility</td>
<td>IFCID 0376 trace function code</td>
</tr>
<tr>
<td>---------------------</td>
<td>-------------------------------------------</td>
<td>------------------------------</td>
</tr>
<tr>
<td>The lengths of values that are returned from CURRENT CLIENT_USERID, CURRENT CLIENT_WRKSTNNNAME, CURRENT CLIENT_APPLNAME, or CURRENT CLIENT_ACCTNG special register are longer than the DB2 10 limits.</td>
<td>The special register values are truncated to the DB2 10 maximum lengths and padded with blanks</td>
<td>1104, 1105, 1106, 1107</td>
</tr>
<tr>
<td>A CAST(string as TIMESTAMP) specification with an input string of length of 8 or an input string of length 13</td>
<td>An explicit cast specification from string as TIMESTAMP interprets an 8-byte character string as a Store Clock value and a 13-byte string as a GENERATE_UNIQUE value. CAST result might be incorrect.</td>
<td>1109</td>
</tr>
<tr>
<td>Invocation of the SPACE or VARCHAR built-in function when the result is defined as VARCHAR(32765), VARCHAR(32766), or VARCHAR(32767)</td>
<td>No error</td>
<td>1110, 1111</td>
</tr>
<tr>
<td>Subsystem parameter XML_RESTRICT_EMPTY_TAG is set to YES, and an empty XML element is serialized as &lt;emptyElement&gt;&lt;/emptyElement&gt;</td>
<td>No error</td>
<td>1112</td>
</tr>
<tr>
<td>Specification of bind option DBPROTOCOL(DDRCBF)</td>
<td>DSNT298I</td>
<td></td>
</tr>
<tr>
<td>A period specification that follows the name of a view in the FROM clause of a query</td>
<td>SQLCODE -4743</td>
<td></td>
</tr>
<tr>
<td>A period clause that follows the name of a target view in an UPDATE or DELETE statement</td>
<td>SQLCODE -4743</td>
<td></td>
</tr>
<tr>
<td>A SET CURRENT TEMPORAL SYSTEM_TIME statement</td>
<td>SQLCODE -4743</td>
<td></td>
</tr>
<tr>
<td>A SET CURRENT TEMPORAL BUSINESS_TIME statement</td>
<td>SQLCODE -4743</td>
<td></td>
</tr>
<tr>
<td>A reference to a global variable</td>
<td>SQLCODE -4743</td>
<td></td>
</tr>
<tr>
<td>Use of array operations and built-in functions such as</td>
<td>SQLCODE -4743</td>
<td></td>
</tr>
<tr>
<td>- Use of the UNNEST collection-derived-table</td>
<td></td>
<td></td>
</tr>
<tr>
<td>- Use of the ARRAY_FIRST, ARRAY_LAST, ARRAY_NEXT, ARRAY_PRIOR, ARRAY_AGG, TRIM_ARRAY, CARDINALITY, MAX_CARDINALITY built-in functions</td>
<td></td>
<td></td>
</tr>
<tr>
<td>- A SET assignment-statement of an array element as a target table</td>
<td></td>
<td></td>
</tr>
<tr>
<td>- A CAST specification with a parameter marker as the source and an array as the data type</td>
<td></td>
<td></td>
</tr>
<tr>
<td>An aggregate function that contains the keyword DISTINCT and references a column that is defined with a column mask</td>
<td>SQLCODE -20478</td>
<td></td>
</tr>
</tbody>
</table>
### Table 134. Behavior of V10R1 application compatibility (continued)

<table>
<thead>
<tr>
<th>Feature or Function</th>
<th>Result with V10R1 application compatibility</th>
<th>IFCID 0376 trace function code</th>
</tr>
</thead>
<tbody>
<tr>
<td>An SQL statement contains the GROUP BY clause and references a column that is defined with a column mask</td>
<td>SQLCODE -20478</td>
<td></td>
</tr>
<tr>
<td>An SQL statement contains the set operator UNION ALL or UNION DISTINCT and references a column that is defined with a column mask</td>
<td>SQLCODE -20478</td>
<td></td>
</tr>
<tr>
<td>A reference to an alias for a sequence object</td>
<td>SQLCODE -4743</td>
<td></td>
</tr>
<tr>
<td>A reference to an unqualified sequence that is not resolved to a public alias</td>
<td>SQLCODE -204</td>
<td></td>
</tr>
<tr>
<td>A SELECT with a table function reference that includes a typed correlation clause</td>
<td>SQLCODE -4743</td>
<td></td>
</tr>
<tr>
<td>A CALL statement that specifies an autonomous procedure</td>
<td>SQLCODE -4743</td>
<td></td>
</tr>
<tr>
<td>The following datetime assignments:</td>
<td>SQLCODE -180</td>
<td></td>
</tr>
<tr>
<td>• A valid string representation of a timestamp to a date column</td>
<td></td>
<td></td>
</tr>
<tr>
<td>• A valid string representation of a timestamp to a time column</td>
<td></td>
<td></td>
</tr>
<tr>
<td>• A valid string representation of a date to a timestamp column</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Notes:

1. To find details about the incompatible parameters, examine the contents of fields QW0376SC_Var, QW0376PR_Var, and QW0376INC_Var. See the DSNWMSGS file for more information.

### Related concepts

- Application and SQL release incompatibilities (Db2 for z/OS What's New?)
- V11R1 application compatibility level

When you set the application compatibility level to V11R1, applications that attempt to use functions and features that are introduced in Db2 12 or later might behave differently or receive an error.

### Managing application incompatibilities

Before you move an application to a new application compatibility level, you need to find application incompatibilities, adjust your applications for those incompatibilities, and verify that the incompatibilities no longer exist.

### Procedure

1. Start a trace that includes IFCID 0239 to capture package information. For example, issue the following START TRACE command:

   ```
   GUPI -START TRACE(ACCTG) CLASS(7,8,10)
   ```

2. Examine the trace output.
3. Start a trace for IFCID 0376 to report incompatibility information about the packages. For example, issue the following START TRACE command:

```
-GUPI
-START TRACE(PERFM) CLASS(32) IFCID(376)
```

4. Run the application.

5. Examine the trace output.

6. Prepare the application for execution. When you bind the packages for the application, use the old APPLCOMPAT value.

7. Run the application.

8. Examine the trace output again to verify that the incompatibilities no longer exist.

**What to do next**

When the application runs at the old level with no application incompatibilities, rebind the package with the APPLCOMPAT value for the new function level.

**Related concepts**

- Performance trace (Db2 Performance)

**Related reference**

- -START TRACE (Db2) (Db2 Commands)
- Trace field descriptions (Db2 Performance)

**Enabling default application compatibility with function level 500 or higher**

The APPLCOMPAT subsystem parameter specifies the default value of the APPLCOMPAT bind option. Before function level 500 or higher is activated, the APPLCOMPAT subsystem parameter must be set to V11R1 or V10R1. These settings ensure that existing SQL applications are bound for compatibility with the earlier release by default.

**Before you begin**

- Activate function level 500 or higher, as described in Activating Db2 12 new function at migration (Db2 Installation and Migration).

- Take the following precautions to ensure that applications are ready to run at the higher application compatibility level by default.
  - Identify and resolve all application incompatibilities of the higher level, as described in “Managing application incompatibilities” on page 831.
  - Rebind any packages that must continue to run at the lower application compatibility level and explicitly specify the APPLCOMPAT bind option for that level.

**Procedure**

To enable default application compatibility with the current function level:
1. Change the APPLCOMPAT subsystem parameter setting. Set the value to V12R1M500 or the equivalent higher active function-level value.

   You can complete this step as described in Updating subsystem parameter and application default values (Db2 Installation and Migration), or by modifying your customized copy of the DSNTIJUZ job.

   The format is VvvRrMmmm, where vv is the version, r is the release, and mmm is the modification level. For example, V12R1M510 identifies function level 510. For a list of all available function levels in Db2 12, see Db2 12 function levels (Db2 for z/OS What’s New?). See the activation details for each function level for a summary the new features that are controlled by the corresponding application compatibility level.

2. Run the first two job steps of DSNTIJUZ to rebuild your subsystem parameter (DSNZPxxx) module.

3. Use the -SET SYSPARM command or restart Db2.

**Results**

Future bind and rebind operations set the application compatibility level of the package to the APPLCOMPAT subsystem parameter value, if the APPLCOMPAT bind option is not specified. Packages that are bound or rebound at the higher level can begin use of SQL capabilities introduced at that level.

**Related concepts**

Application compatibility levels in Db2

The application compatibility level of your applications controls the adoption and use of new capabilities and enhancements, and the impact of incompatible changes. The advantage is that you can complete the Db2 12 migration process without the need to update your applications immediately.

**Related tasks**

Adopting new capabilities in Db2 12 continuous delivery (Db2 for z/OS What’s New?)

Activating Db2 12 new function at migration (Db2 Installation and Migration)

**Related reference**

APPL COMPAT LEVEL field (APPLCOMPAT subsystem parameter) (Db2 Installation and Migration)

APPLCOMPAT bind option (Db2 Commands)
Chapter 9. Preparing an application to run on Db2 for z/OS

To prepare and run applications that contain embedded static SQL statements or dynamic SQL statements, you must precompile, compile, link-edit, and bind them.

Before you begin

To avoid rework, follow these steps:

1. Test your SQL statements by using SPUFI.
2. Compile your program with no SQL statements, and resolve all compiler errors.
3. Proceed with the preparation and the Db2 precompiler or with the host compiler that supports that Db2 coprocessor.

The following types of applications require different methods of program preparation:

- Applications that contain ODBC calls
- Applications in interpreted languages, such as REXX. For information about running REXX programs, which you do not prepare for execution, see “Running a Db2 REXX application” on page 936.
- Java applications, which can contain JDBC calls or embedded SQL statements

Procedure

You can perform these steps by using one of the following methods:

The following topics provide details on preparing and running a Db2 application:

a) “Processing SQL statements” on page 839
b) “Compiling and link-editing an application” on page 863
c) “Binding application packages and plans” on page 864
d) Chapter 10, “Running an application on Db2 for z/OS,” on page 933

Binding a package is not necessary in all cases. These instructions assume that you bind some of your DBRMs into packages and include a package list in your plan.

If you use CICS, you might need additional steps; see:

- “Translating command-level statements in a CICS program” on page 849
- “Example of calling applications in a command procedure” on page 945

For more information about when to bind a package, see “Db2 program preparation overview” on page 893.

Preparing applications by using JCL procedures

A number of methods are available for preparing an application to run. You can:

- Use Db2 interactive (DB2I) panels, which lead you step by step through the preparation process.
- Submit a background job using JCL (which the program preparation panels can create for you).
- Start the DSNH CLIST in TSO foreground or background.
- Use TSO prompters and the DSN command processor.
- Use JCL procedures added to your SYS1.PROCLIB (or equivalent) at Db2 installation time.
- You can invoke the coprocessor from UNIX System Services. If the DBRM is generated in a HFS file, you can also use the command line processor to bind the resulting DBRM. Optionally, you can also
copy the DBRM into a partitioned data set member by using the oput and oget commands and then bind it by using conventional JCL.

This topic describes how to use JCL procedures to prepare a program. For information about using the DB2I panels, see Chapter 9, “Preparing an application to run on Db2 for z/OS,” on page 835.

- **Preparing applications by the Db2 Program Preparation panels**

  If you develop programs using TSO and ISPF, you can prepare them to run by using the Db2 Program Preparation panels. These panels guide you step by step through the process of preparing your application to run. Other ways of preparing a program to run are available, but using Db2 Interactive (DB2I) is the easiest because it leads you automatically from task to task.

  **Important:** If your C++ program satisfies both of the following conditions, you must use a JCL procedure to prepare it:
  - The program consists of more than one data set or member.
  - More than one data set or member contains SQL statements.

  To prepare an application by using the Db2 Program Preparation panels:
  a. If you want to display or suppress message IDs during program preparation, specify one of the following commands on the ISPF command line:
     - `TSO PROFILE MSGID`  
       Message IDs are displayed
     - `TSO PROFILE NOMSGID`  
       Message IDs are suppressed
  b. Open the DB2I Primary Option Menu.
  c. Select the option that corresponds to the Program Preparation panel.
  d. Complete the Program Preparation panel and any subsequent panels. After you complete each panel, DB2I automatically displays the next appropriate panel.

- **Preparation guidelines for DL/I batch programs**

  Use the following guidelines when you prepare a program to access Db2 and DL/I in a batch program:
  - “Processing SQL statements by using the Db2 precompiler” on page 840
  - “Binding a batch program” on page 877
  - “Compiling and link-editing an application” on page 863
  - “Loading and running a batch program” on page 939

  **Related concepts**
  The Db2 command line processor (Db2 Commands)
  TSO attachment facility (Introduction to Db2 for z/OS)

  **Related reference**
  The DB2I primary option menu (Introduction to Db2 for z/OS)
  DSNH (TSO CLIST) (Db2 Commands)

### Setting the DB2I defaults

When you use the Db2 Interactive (DB2I) panels to prepare an application, you can specify the default values that DB2I is to use. These defaults values can include the default application language and default JCL JOB statement. Otherwise, DB2I uses the system default values that were set at installation time.

**Procedure**

As DB2I leads you through a series a panels, enter the default values that you want on the following panels when they are displayed.
### Table 135. DB2I panels to use to set default values

<table>
<thead>
<tr>
<th>If you want to set the following default values...</th>
<th>Use this panel</th>
</tr>
</thead>
<tbody>
<tr>
<td>• subsystem ID</td>
<td>DB2I Defaults Panel 1 panel</td>
</tr>
<tr>
<td>• number of additional times to attempt to connect to Db2</td>
<td></td>
</tr>
<tr>
<td>• programming language</td>
<td></td>
</tr>
<tr>
<td>• number of lines on each page of listing or SPUFI output</td>
<td></td>
</tr>
<tr>
<td>• lowest level of message to return to you during the BIND phase</td>
<td></td>
</tr>
<tr>
<td>• SQL string delimiter for COBOL programs</td>
<td></td>
</tr>
<tr>
<td>• how to represent decimal separators</td>
<td></td>
</tr>
<tr>
<td>• smallest value of the return code (from precompile, compile, link-edit, or bind) that prevents later steps from running</td>
<td></td>
</tr>
<tr>
<td>• default number of input entry rows to generate on the initial display of ISPF panels</td>
<td></td>
</tr>
<tr>
<td>• user ID to associate with the trusted connection for the current DB2I session</td>
<td></td>
</tr>
<tr>
<td>• default JOB statement</td>
<td>DB2I Defaults Panel 2 panel</td>
</tr>
<tr>
<td>• symbol used to delimit a string in a COBOL statement in a COBOL application</td>
<td></td>
</tr>
<tr>
<td>• whether DCLGEN generates a picture clause that has the form PIC G(n) DISPLAY-1 or PIC N(n).</td>
<td></td>
</tr>
</tbody>
</table>
### Table 135. DB2I panels to use to set default values (continued)

<table>
<thead>
<tr>
<th>If you want to set the following default values…</th>
<th>Use this panel</th>
</tr>
</thead>
<tbody>
<tr>
<td>The following package and plan characteristics</td>
<td>Defaults for Bind Package panel</td>
</tr>
<tr>
<td>• isolation level</td>
<td>Defaults for Bind Plan panel</td>
</tr>
<tr>
<td>• whether to check authorization at run time or at bind time</td>
<td></td>
</tr>
<tr>
<td>• when to release locks on resources</td>
<td></td>
</tr>
<tr>
<td>• whether to obtain EXPLAIN information about how SQL statements in the plan or package execute</td>
<td></td>
</tr>
<tr>
<td>• whether you need data currency for ambiguous cursors opened at remote locations</td>
<td></td>
</tr>
<tr>
<td>• whether to use parallel processing</td>
<td></td>
</tr>
<tr>
<td>• whether Db2 determines access paths at bind time and again at execution time</td>
<td></td>
</tr>
<tr>
<td>• whether to defer preparation of dynamic SQL statements</td>
<td></td>
</tr>
<tr>
<td>• whether Db2 keeps dynamic SQL statements after commit points</td>
<td></td>
</tr>
<tr>
<td>• the application encoding scheme</td>
<td></td>
</tr>
<tr>
<td>• whether you want to use optimization hints to determine access paths</td>
<td></td>
</tr>
<tr>
<td>• when Db2 writes the changes for updated group buffer pool-dependent pages</td>
<td></td>
</tr>
<tr>
<td>• whether run time (RUN) or bind time (BIND) rules apply to dynamic SQL statements at run time</td>
<td></td>
</tr>
<tr>
<td>• whether to continue to create a package after finding SQL errors (packages only)</td>
<td></td>
</tr>
<tr>
<td>• when to acquire locks on resources (plans only)</td>
<td></td>
</tr>
<tr>
<td>• whether a CONNECT (Type 2) statement executes according to Db2 rules (Db2) or the SQL standard (STD). (plans only)</td>
<td></td>
</tr>
<tr>
<td>• which remote connections end during a commit or a rollback (plans only)</td>
<td></td>
</tr>
</tbody>
</table>

**Related reference**

**DB2I Defaults Panel 1**

DB2I Defaults Panel 1 lets you change many of the system default values that were set at Db2 installation time.

**DB2I Defaults Panel 2**

After you press Enter on the DB2I Defaults Panel 1, the DB2I Defaults Panel 2 is displayed. If you chose IBMCOB as the language on the DB2I Defaults Panel 1, three fields are displayed. Otherwise, only the first field is displayed.

**Defaults for Bind Package and Defaults for Rebind Package panels**

These DB2I panels lets you change your defaults for BIND PACKAGE and REBIND PACKAGE options.

**Defaults for Bind Plan and Defaults for Rebind Plan panels**
These DB2I panels let you change your defaults for BIND PLAN and REBIND PLAN options.

**Processing SQL statements**

The first step in preparing an SQL application to run is to process the SQL statements in the program. To process the statements, use either the Db2 precompiler or the Db2 coprocessor. During this step, the SQL statements are replaced with calls to Db2 language interface modules, and a DBRM is created.

**About this task**

For assembler or Fortran applications, use the Db2 precompiler to prepare the SQL statements.

For C, C++, COBOL, or PL/I applications, you can use one of the following techniques to process SQL statements:

- Use the Db2 precompiler before you compile your program.
  - You can use this technique with any supported version of C or C++, COBOL, or PL/I.
- Invoke the Db2 coprocessor for the host language that you are using as you compile your program.
  - You can use the Db2 coprocessor with C, C++, COBOL, and PL/I host compilers. To invoke the Db2 coprocessor, specify the SQL compiler option followed by its suboptions, which are those options that are defined for the Db2 precompiler. Some Db2 precompiler options are ignored. You can also invoke the coprocessor from UNIX System Services on z/OS to generate a DBRM in either a partitioned data set or an HFS file.
    - For C or C++, you need IBM z/OS Version 1 Release 8 C/C++ or later.
    - For COBOL, you need Enterprise COBOL for z/OS Version 3 Release 4 or later to use this technique.
    - For PL/I, you need Enterprise PL/I for z/OS Version 3 Release 4 or later to use this technique.

**CICS:** If the application contains CICS commands, you must translate the program before you compile it.

**Db2 version in DSNHDECP module**

When you process SQL statements in your program, if the Db2 version in DSNHDECP is the default system-provided version, Db2 issues a warning and processing continues. In this case, ensure that the information in DSNHDECP that Db2 uses accurately reflects your environment.

**SQL statement processing**

Because most compilers do not recognize SQL statements, you can prevent compiler errors by using either the Db2 precompiler or the Db2 coprocessor.

The precompiler scans the program and returns modified source code, which you can then compile and link edit. The precompiler also produces a DBRM (database request module). You can bind this DBRM to a package using the BIND subcommand. When you complete these steps, you can run your Db2 application.

Alternatively, you can use the Db2 coprocessor for the host language. The Db2 coprocessor performs Db2 precompiler functions at compile time. When you use the Db2 coprocessor, the compiler (rather than the precompiler) scans the program and returns the modified source code. The Db2 coprocessor also produces a DBRM.

**Related concepts**

Using the Db2 C/C++ precompiler (XL C/C++ Programming Guide)
Db2 coprocessor (Enterprise COBOL for z/OS Programming Guide)
Differences between the Db2 precompiler and the Db2 coprocessor
The Db2 precompiler and Db2 coprocessor have architectural differences. You cannot switch from one to the other without considering those differences and adjusting your program accordingly.

Db2 program preparation overview
Before you can run an application program on Db2 for z/OS, you need to prepare it. To prepare the program, create a load module, possibly one or more packages, and an application plan.

Program directories for Db2 12 (Db2 for z/OS in IBM Documentation)

Related tasks
Translating command-level statements in a CICS program
You can translate CICS applications with the CICS command language translator as a part of the program preparation process. CICS command language translators are available only for assembler, C, COBOL, and PL/I languages.

Related reference
Enterprise COBOL for z/OS

Processing SQL statements by using the Db2 precompiler

The Db2 precompiler scans a program and copies all of the SQL statements and host variable information into a DBRM (database request module). The precompiler also returns source code that has been modified so that the SQL statements do not cause errors when you compile the program.

About this task
After the SQL statements and host variable information are copied into a DBRM and the modified source code is returned, you can compile and link-edit this modified source code.

Before you run the Db2 precompiler, use DCLGEN to obtain accurate SQL DECLARE TABLE statements. The precompiler checks table and column references against SQL DECLARE TABLE statements in the program, not the actual tables and columns.

Db2 does not need to be active when you precompile your program.

You do not need to precompile the program on the same Db2 subsystem on which you bind the DBRM and run the program. You can bind a DBRM and run it on a Db2 subsystem at the previous release level, if the original program does not use any properties of Db2 that are unique to the current release. You can also run applications on the current release that were previously bound on subsystems at the previous release level.

Procedure
To process SQL statements by using the Db2 precompiler:
1. Ensure that your program is ready to be processed by the Db2 precompiler by performing the following actions.
   For information about the criteria for programs that are passed to the precompiler, see “Input to the Db2 precompiler” on page 843.
2. If you plan to run multiple precompilation jobs and are not using the DFSMSdfp partitioned data set extended (PDSE), change the Db2 language preparation procedures (DSNHCOb, DSNHCOb2, DSNHICOB, DSNHFOR, DSNHC, DSNHPLI, DSNHASM, DSNHSQL) to specify the DISP=OLD parameter instead of the DISP=SHR parameter.
   The Db2 language preparation procedures in job DSNTIJMV use the DISP=OLD parameter to enforce data integrity. However, the installation process converts the DISP=OLD parameter for the DBRM library data set to DISP=SHR, which can cause data integrity problems when you run multiple precompilation jobs.
3. Start the precompile process by using one of the following methods:
   • DB2I panels. Use the Precompile panel or the Db2 Program Preparation panels. For details, see “DB2I panels that are used for program preparation” on page 900.
   • The DSNH command procedure (a TSO CLIST). For details, see DSNH (TSO CLIST) (Db2 Commands).
   • JCL procedures that are supplied with Db2. For details, see “Db2-supplied JCL procedures for preparing an application” on page 897.
**Recommendation:** Specify the SOURCE and XREF precompiler options to get complete diagnostic output from the Db2 precompiler. This output is useful if you need to precompile and compile program source statements several times before they are error-free and ready to link-edit.

The output that is returned from the Db2 precompiler is described in “Output from the Db2 precompiler” on page 845.

**What to do next**

**Preparing a program with object-oriented extensions by using JCL**

If your C++ or Enterprise COBOL for z/OS program satisfies both of these conditions, you need special JCL to prepare it:

- The program consists of more than one data set or member.
- More than one data set or member contains SQL statements.

You must precompile the contents of each data set or member separately, but the prelinker must receive all of the compiler output together.

JCL procedure DSNHCPP2, which is in member DSNTIJMV of data set DSN1210.SDSNSAMP, shows you one way to do this for C++.

**Precompiling a batch program**

When you add SQL statements to an application program, you must precompile the application program and bind the resulting DBRM into a package, as described in Chapter 9, “Preparing an application to run on Db2 for z/OS,” on page 835.

**Related concepts**

DCLGEN (declarations generator)

Your program should declare the tables and views that it accesses. The Db2 declarations generator, DCLGEN, produces these DECLARE statements for C, COBOL, and PL/I programs, so that you do not need to code the statements yourself. DCLGEN also generates corresponding host variable structures.

**Related reference**

DSNH (TSO CLIST) (Db2 Commands)

**Data sets that the precompiler uses**

When you invoke the precompiler you need to provide data sets that contain input for the precompiler, such as the host programming statements and SQL statements. You also need to provide data sets where the precompiler can store its output, such as the modified source code and diagnostics messages.

<table>
<thead>
<tr>
<th>DD statement</th>
<th>Data set description</th>
<th>Required?</th>
</tr>
</thead>
<tbody>
<tr>
<td>DBRMLIB</td>
<td>Output data set, which contains the SQL statements and host variable information that the Db2 precompiler extracted from the source program. It is called Database Request Module (DBRM). This data set becomes the input to the Db2 bind process. The DCB attributes of the data set are RECFM FB, LRECL 80. DBRMLIB has to be a PDS and a member name must be specified. You can use IEBCOPY, IEHPROGM, TSO commands, COPY and DELETE, or PDS management tools for maintaining the data set.</td>
<td>Yes</td>
</tr>
<tr>
<td>DD statement</td>
<td>Data set description</td>
<td>Required?</td>
</tr>
<tr>
<td>--------------</td>
<td>------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
<td>----------</td>
</tr>
<tr>
<td>STEPLIB</td>
<td>Step library for the job step. In this DD statement, you can specify the name of the library for the precompiler load module, DSNHPC, and the name of the library for your Db2 application programming defaults member, DSNHDECP.</td>
<td>No, but recommended</td>
</tr>
<tr>
<td></td>
<td><strong>Recommendation:</strong> Always use the STEPLIB DD statement to specify the library where your Db2 DSNHDECP module resides to ensure that the proper application defaults are used by the Db2 precompiler. The library that contains your Db2 DSNHDECP module needs to be allocated ahead of the prefix.SDSNLOAD library.</td>
<td></td>
</tr>
<tr>
<td>SYSCIN</td>
<td>Output data set, which contains the modified source that the Db2 precompiler writes out. This data set becomes the input data set to the compiler or assembler. This data set must have attributes RECFM F or FB, and LRECL 80. SYSCIN can be a PDS or a sequential data set. If a PDS is used, the member name must be specified.</td>
<td>Yes</td>
</tr>
<tr>
<td>SYSIN</td>
<td>Input data set, which contains statements in the host programming language and embedded SQL statements. This data set must have the attributes RECFM F or FB, LRECL 80. SYSIN can be a PDS or a sequential data set. If a PDS is used, the member name must be specified.</td>
<td>Yes</td>
</tr>
<tr>
<td>SYSLIB</td>
<td>INCLUDE library, which contains additional SQL and host language statements. The Db2 precompiler includes the member or members that are referenced by SQL INCLUDE statements in the SYSIN input from this DD statement. Multiple data sets can be specified, but they must be partitioned data sets with attributes RECFM F or FB, LRECL 80. SQL INCLUDE statements cannot be nested.</td>
<td>No</td>
</tr>
<tr>
<td>SYSPRINT</td>
<td>Output data set, which contains the output listing from the Db2 precompiler. This data set must have an LRECL of 133 and a RECFM of FBA. SYSPRINT must be a sequential data set.</td>
<td>Yes</td>
</tr>
<tr>
<td>SYSTERM</td>
<td>Terminal output file, which contains diagnostic messages from the Db2 precompiler. The DCB attributes of the data set are determined by the z/OS system. SYSTERM must be a sequential data set.</td>
<td>No</td>
</tr>
</tbody>
</table>
Table 136. DD statements and data sets that the Db2 precompiler uses (continued)

<table>
<thead>
<tr>
<th>DD statement</th>
<th>Data set description</th>
<th>Required?</th>
</tr>
</thead>
<tbody>
<tr>
<td>SYSUT1 and SYSUT2</td>
<td>Internal work files that the precompiler uses to store temporary information as it converts embedded SQL statements to host language statements. Precompilation of assembler and PL/I source code uses only the SYSUT1 data set. The default SPACE parameter values in the Db2-supplied program preparation procedures (DSNHASM, DSNHC, DSNHCPP, DSNHCPP2, DSNHICOB, DSNHPLI, and DSNHFOR) are adequate in most cases. If your application program contains a large number of embedded SQL statements, you might need to increase those values.</td>
<td>No, unless you need to override the default SPACE parameter values.</td>
</tr>
</tbody>
</table>

Related reference
SPACE Parameter (MVS JCL Reference)

Input to the Db2 precompiler

The primary input for the precompiler consists of statements in the host programming language and embedded SQL statements.

You can use the SQL INCLUDE statement to get secondary input from the include library, SYSLIB. The SQL INCLUDE statement reads input from the specified member of SYSLIB until it reaches the end of the member.

Another preprocessor, such as the PL/I macro preprocessor, can generate source statements for the precompiler. Any preprocessor that runs before the precompiler must be able to pass on SQL statements. Similarly, other preprocessors can process the source code, after you precompile and before you compile or assemble.

Input to the Db2 precompiler has the following restrictions:

- The size of a source program that Db2 can precompile is limited by the region size and the virtual memory available to the precompiler. These amounts vary with each system installation.
- The forms of source statements that can pass through the precompiler are limited. For example, constants, comments, and other source syntax that are not accepted by the host compilers (such as a missing right brace in C) can interfere with precompiler source scanning and cause errors. To check for such unacceptable source statements, run the host compiler before the precompiler. You can ignore the compiler error messages for SQL statements or comment out the SQL statements. After the source statements are free of unacceptable compiler errors, you can then uncomment any SQL statements that you previously commented out and continue with the normal Db2 program preparation process for that host language.
- You must write host language statements and SQL statements using the same margins, as specified in the precompiler option MARGINS.
- The input data set, SYSIN, must have the attributes RECFM F or FB, LRECL 80.
- SYSLIB must be a partitioned data set, with attributes RECFM F or FB, LRECL 80.
- Input from the INCLUDE library cannot contain other precompiler INCLUDE statements.
Starting the precompiler dynamically when using JCL procedures

You can call the precompiler from an assembler program by using a macro.

About this task

You can call the precompiler from an assembler program by using one of the macro instructions ATTACH, CALL, LINK, or XCTL.

To call the precompiler, specify DSNHPC as the entry point name. You can pass three address options to the precompiler; the following topics describe their formats. The options are addresses of:

- A precompiler option list
- A list of alternative DD names for the data sets that the precompiler uses
- A page number to use for the first page of the compiler listing on SYSPRINT

Related reference

Using X-macros (MVS Assembler Services Reference)

Precompiler option list format

When you call the precompiler, you can specify a number of options, in a list, for SQL statement processing. You must specify that option list in a particular format.

The option list must begin on a 2-byte boundary. The first 2 bytes contain a binary count of the number of bytes in the list (excluding the count field). The remainder of the list is EBCDIC and can contain precompiler option keywords, separated by one or more blanks, a comma, or both.

DD name list format

When you call the precompiler, you can specify a list of alternative DD names for the data sets that the precompiler uses. You must specify this list in a particular format.

The DD name list must begin on a 2-byte boundary. The first 2 bytes contain a binary count of the number of bytes in the list (excluding the count field). Each entry in the list is an 8-byte field, left-justified, and padded with blanks if needed.

The following table gives the following sequence of entries:

<table>
<thead>
<tr>
<th>Entry</th>
<th>Standard ddname</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Not applicable</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Not applicable</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Not applicable</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>SYSLIB</td>
<td>Library input</td>
</tr>
<tr>
<td>5</td>
<td>SYSIN</td>
<td>Source input</td>
</tr>
<tr>
<td>6</td>
<td>SYSPRINT</td>
<td>Diagnostic listing</td>
</tr>
<tr>
<td>7</td>
<td>Not applicable</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>SYSUT1</td>
<td>Work data</td>
</tr>
<tr>
<td>9</td>
<td>SYSUT2</td>
<td>Work data</td>
</tr>
<tr>
<td>10</td>
<td>Not applicable</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>Not applicable</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>SYSTERM</td>
<td>Diagnostic listing</td>
</tr>
<tr>
<td>13</td>
<td>Not applicable</td>
<td></td>
</tr>
</tbody>
</table>
Page number format
When you call the precompiler, you can specify a page number to use for the first page of the compiler listing on SYSPRINT. You must specify this page number in a particular format.

A 6-byte field beginning on a 2-byte boundary contains the page number. The first 2 bytes must contain the binary value 4 (the length of the remainder of the field). The last 4 bytes contain the page number in character or zoned-decimal format.

The precompiler adds 1 to the last page number that is used in the precompiler listing and puts this value into the page-number field before returning control to the calling routine. Thus, if you call the precompiler again, page numbering is continuous.

Output from the Db2 precompiler
The major output from the Db2 precompiler is a database request module (DBRM). However, the Db2 precompiler also produces modified source statements, a list of source statements, a list of statements that refer to host names and columns, and diagnostics.

Specifically, the precompiler produces the following types of output:

listing output
The Db2 precompiler writes the following information in the SYSPRINT data set:

- Precompiler source listing
  If the Db2 precompiler option SOURCE is specified, a source listing is produced. The source listing includes precompiler source statements, with line numbers that are assigned by the precompiler.
- Precompiler diagnostics
  The precompiler produces diagnostic messages that include precompiler line numbers of statements that have errors.
- Precompiler cross-reference listing
  If the Db2 precompiler option XREF is specified, a cross-reference listing is produced. The cross-reference listing shows the precompiler line numbers of SQL statements that refer to host names and columns.

The SYSPRINT data set has an LRECL of 133 and a RECFM of FBA. This data set uses the CCSID of the source program. Statement numbers in the output of the precompiler listing are displayed as they appear in the listing.

Terminal diagnostics
If a terminal output file, SYSTERM, exists, the Db2 precompiler writes diagnostic messages to it. A portion of the source statement accompanies the messages in this file. You can often use the SYSTERM file instead of the SYSPRINT file to find errors. This data set uses EBCDIC.

Modified source statements
The Db2 precompiler writes the source statements that it processes to SYSCIN, the input data set to the compiler or assembler. This data set must have attributes RECFM F or FB, and LRECL 80. The modified source code contains calls to the Db2 language interface. The SQL statements that the calls replace appear as comments. This data set uses the CCSID of the source program.

Database request modules
The database request module (DBRM) is a data set that contains the SQL statements and host variable information that is extracted from the source program, along with information that identifies...
the program and ties the DBRM to the translated source statements. It becomes the input to the bind process.

The data set requires space to hold all the SQL statements plus space for each host variable name and some header information. The header information alone requires approximately two records for each DBRM, 20 bytes for each SQL record, and 6 bytes for each host variable.

For an exact format of the DBRM, see the DBRM mapping macros, DSNXDBRM and DSNXNBRM, in library prefix.SDSNMACS. The DCB attributes of the data set are RECFM FB, LRECL 80. The precompiler sets the characteristics. You can use IEBCOPY, IEHPROGM, TSOCOPY and DELETE commands, or other PDS management tools for maintaining these data sets.

**Restriction:** Do not modify the contents of the DBRM. If you do, unpredictable results can occur. Db2 does not support modified DBRMs.

In a DBRM, the SQL statements and the list of host variable names use the UTF-8 character encoding scheme.

All other character fields in a DBRM use EBCDIC. The current release marker (DBRMMRIC) in the header of a DBRM is marked according to the release of the precompiler, regardless of the value of NEWFUN.

---

**Processing SQL statements by using the Db2 coprocessor**

As an alternative to the Db2 precompiler, you can use the Db2 coprocessor to process SQL statements. The Db2 coprocessor performs Db2 precompiler functions at compile time.

**About this task**

**Exception:** For PL/I, the Db2 coprocessor is called from the PL/I SQL preprocessor instead of the compiler.

The Db2 coprocessor has fewer restrictions on SQL programs than the Db2 precompiler. When you process SQL statements with the Db2 coprocessor, you can do the following things in your program:

- Use fully qualified names for structured host variables.
- Include SQL statements at any level of a nested program, instead of in only the top-level source file. (Although you can include SQL statements at any level of a nested program, you must compile the entire program as one unit.)
- Use nested SQL INCLUDE statements.
- **For C or C++ programs only:** Write applications with variable length format.
- **For C or C++ programs only:** Use codepage-dependent characters, such as left and right brackets, without using tri-graph notation when the programs use different code pages.

**Procedure**

To process SQL statements by using the Db2 coprocessor, take one of the following actions:

- Submit a JCL job to process that SQL statement. Include the following information:
  - Specify the SQL compiler option when you compile your program:
    - The SQL compiler option indicates that you want the compiler to invoke the Db2 coprocessor. Specify a list of SQL processing options in parentheses after the SQL keyword. Table 139 on page 852 lists the options that you can specify.
    - For COBOL and PL/I, enclose the list of SQL processing options in single or double quotation marks. For PL/I, separate options in the list by a comma, blank, or both, as shown in the following examples:
      - **C/C++**
        ```
        SQL('AP0STSQL STDSQL(NO)')
        ```
For PL/I programs that use BIGINT or LOB data types, specify the following compiler options when you compile your program:

```
LIMITS(FIXEDBIN(63), FIXEDDEC(31))
```

If needed, increase the user's region size so that it can accommodate more memory for the Db2 coprocessor.

Include DD statements for the following data sets in the JCL for your compile step:

- Db2 load library (prefix.SDSNLOAD)
  The Db2 coprocessor calls Db2modules to process the SQL statements. You therefore need to include the name of the Db2 load library data set in the STEPLIB concatenation for the compiler step.
- DBRM library
  The Db2 coprocessor produces a DBRM. DBRMs and the DBRM library are described in “Output from the Db2 precompiler” on page 845. You need to include a DBRMLIB DD statement that specifies the DBRM library data set.
- Library for SQL INCLUDE statements
  If your program contains SQL INCLUDE member-name statements that specify secondary input to the source program, you need to also specify the data set for member-name. Include the name of the data set that contains member-name in the SYSLIB concatenation for the compiler step.

Invoke the Db2 coprocessor from UNIX System Services on z/OS.

If you invoke the Db2 coprocessor from UNIX System Services, you can choose to have the DBRM generated in a partitioned data set or an HFS file.

When you invoke the Db2 coprocessor, specify the SQL compiler option. The SQL compiler option indicates that you want the compiler to invoke the Db2 coprocessor. Specify a list of SQL processing options in parentheses after the SQL keyword. For the list of options that you can specify, see SQL processing options.

The file name for the DBRM is determined as described in DRBMLIB. For host languages other than C and C++, the DBRMLIB option is not supported and the file name is always generated. For C and C++, you can specify one of the following items:

- The name of a partitioned data set. The following example invokes the C/C++ Db2 coprocessor to compile (with the c89 compiler) a sample C program and requests that the resulting DBRM is stored in the test member of the userid.dbrmlib.data data set:

  ```
c89 -Wc,"sql,dbrmlib(/'userid.dbrmlib.data(test)'),langlvl(extended)" -c t.c
  ```

- The name of an HFS file. The name can be qualified, partially qualified, or unqualified. The file path can contain a maximum of 1024 characters, and the file name can contain a maximum of 255 characters. The first 8 characters of the file name, not including the file extension, must be unique within the file system.

  For example, assume that your directory structure is /u/USR001/c/example and that your current working directory is /u/USR001/c. The following table shows examples of how to specify the HFS file names with the DBRMLIB option and how the file names are resolved.
### Table 138. How to specify HFS files to store DBRMs

<table>
<thead>
<tr>
<th>If you specify...</th>
<th>The DBRM is generated in...</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>dbrmlib(/u/USR001/sample.dbrm)</code></td>
<td><code>/u/USR001/sample.dbrm</code></td>
</tr>
<tr>
<td><code>dbrmlib(example/sample.dbrm)</code></td>
<td><code>/u/USR001/c/example/sample.dbrm</code></td>
</tr>
<tr>
<td><code>dbrmlib(../sample.dbrm)</code></td>
<td><code>/u/USR001/sample.dbrm</code></td>
</tr>
<tr>
<td><code>dbrmlib(sample.dbrm)</code></td>
<td><code>/u/USR001/c/sample.dbrm</code></td>
</tr>
</tbody>
</table>

The following example invokes the Db2 coprocessor to compile (with the c89 compiler) a sample C program and requests that the resulting DBRM is stored in the file `test.dbrm` in the `tmp` directory:

```bash
c89 -Wc,"sql,dbrmlib(/tmp/test.dbrm),langlvl(extended)" -c t.c
```

The following example invokes the coprocessor to compile a sample COBOL program with the COBOL Version 6.2 or newer compiler:

```bash
cob2 myprogram.cbl -c myprogram -dbrmlib -qsql
```

The following example invokes the coprocessor to compile a sample PL/I program from the PL/I Version 5.2 or newer compiler:

```bash
pli -c -qpp=sql -qdbrmlib -qrent myprogram.pli
```

If you request that the DBRM be generated in an HFS file, you can bind the resulting DBRM by using the command line processor BIND command. For more information about using the command line processor BIND command, see “Binding a DBRM that is in an HFS file to a package or collection” on page 866. Optionally, you can also copy the DBRM into a partitioned data set member by using the `oput` and `oget` commands and then bind the DBRM by using conventional JCL.

**Support for compiling a COBOL program that includes SQL from an assembler program**

The COBOL compiler provides a facility that enables you to invoke the COBOL compiler by using an assembler program.

If you intend to use the Db2 coprocessor and start the COBOL compiler from an assembler program as part of your Db2 application preparation, you can use the SQL compiler option and provide the alternate `DBRMLIB DD` name the same way that you can specify other alternate DD names. The Db2 coprocessor creates the DBRM member according to your DBRM PDS library and the DBRM member that you specified using the alternate `DBRMLIB DD` name.

To use the alternate `DBRMLIB DD` name, Enterprise COBOL V4.1 and above is required.

**Related reference**

- Starting the compiler from an assembler program
Translating command-level statements in a CICS program

You can translate CICS applications with the CICS command language translator as a part of the program preparation process. CICS command language translators are available only for assembler, C, COBOL, and PL/I languages.

Procedure

Prepare your CICS program in either of these sequences:

<table>
<thead>
<tr>
<th>Sequence</th>
<th>Remarks</th>
</tr>
</thead>
</table>
| a. Db2 precompiler  
b. CICS Command Language Translator. | This sequence is the preferred method of program preparation and the one that the DB2I Program Preparation panels support. If you use the DB2I panels for program preparation, you can specify translator options automatically, rather than needing to provide a separate option string. |
| a. CICS command language translator  
b. Db2 precompiler | This sequence results in a warning message from the CICS translator for each EXEC SQL statement that it encounters. The warning messages have no effect on the result. If you are using double-byte character sets (DBCS), precompiling is recommended before translating, as described previously. |

Use the Db2 precompiler before the CICS translator to prevent the precompiler from mistaking CICS translator output for graphic data.

If your source program is in COBOL, you must specify a string delimiter that is the same for the Db2 precompiler, COBOL compiler, and CICS translator. The defaults for the Db2 precompiler and COBOL compiler are not compatible with the default for the CICS translator.

If the SQL statements in your source program refer to host variables that a pointer stored in the CICS TWA addresses, you must make the host variables addressable to the TWA before you execute those statements. For example, a COBOL application can issue the following statement to establish addressability to the TWA:

```sql
EXEC CICS ADDRESS TWA ('address-of-twa-area')
END-EXEC
```

You can run CICS applications only from CICS address spaces. This restriction applies to the RUN option on the second program DSN command processor. All of those possibilities occur in TSO.

To prepare an application program, you can append JCL from a job that is created by the Db2 Program Preparation panels to the JCL for the CICS command language translator. To run the prepared program under CICS, you might need to define programs and transactions to CICS. Your system programmer must make the appropriate CICS resource or table entries.

prefix.SDSNSAMP contains examples of the JCL that is used to prepare and run a CICS program that includes SQL statements. The set of JCL includes:

- PL/I macro phase
- Db2 precompiling
- CICS Command Language Translation
- Compiling of the host language source statements
- Link-editing of the compiler output
- Binding of the DBRM
- Running of the prepared application.

Related reference
Sample applications in CICS
A set of Db2 sample applications run in the CICS environment.

**Related information**

Resource definition (CICS Transaction Server for z/OS)

## Differences between the Db2 precompiler and the Db2 coprocessor

The Db2 precompiler and Db2 coprocessor have architectural differences. You cannot switch from one to the other without considering those differences and adjusting your program accordingly.

**Recommendation:** Use the coprocessor instead of the precompiler when using Unicode variables in COBOL or PL/I applications.

Depending on whether you use the Db2 precompiler or the Db2 coprocessor, ensure that you account for the following differences:

- **Differences in handling source CCSIDs:**
  The Db2 precompiler and Db2 coprocessor convert the SQL statements of your source program to UTF-8 for parsing.
  
  The precompiler or Db2 coprocessor uses the source CCSID(n) value to convert from that CCSID to CCSID 1208 (UTF-8). The CCSID value must be an EBCDIC CCSID. If you want to prepare a source program that is written in a CCSID that cannot be directly converted to or from CCSID 1208, you must create an indirect conversion.

- **Differences in handling host variable CCSIDs:**
  
  - **COBOL:**
    
    **Db2 precompiler:**
    The Db2 precompiler sets CCSIDs for alphanumeric host variables only when the program includes an explicit DECLARE :hv VARIABLE statement.
    
    **Db2 coprocessor:**
    The COBOL compiler with National Character Support always sets CCSIDs for alphanumeric variables, including host variables that are used within SQL, to the source CCSID. Alternatively, you can specify that you want the COBOL Db2 coprocessor to handle CCSIDs the same way as the precompiler.
    
    **Recommendation:** If you have problems with host variable CCSIDs, use the Db2 precompiler or change your application to include the DECLARE :hv VARIABLE statement to overwrite the CCSID that is specified by the COBOL compiler.
    
    **Example:** Assume that Db2 has mapped a FOR BIT DATA column to a host variable in the following way:

    ```cobol
    01  hv1 pic x(5).
    01  hv2 pic x(5).
    EXEC SQL CREATE TABLE T1 (icolwbit char(5) for bit data, 
    rowid char(5)) END-EXEC.
    EXEC SQL INSERT INTO T1 VALUES (:hv1, :hv2) END-EXEC.
    
    Db2 precompiler: In the modified source from the Db2 precompiler, hv1 and hv2 are represented to Db2 through SQLDA in the following way, without CCSIDs:
    ```

    ```sql
    for hv1: NO CCSID
    20 SQL-PVAR-NAME1 PIC S(4) COMP-4 VALUE +0.
    for hv2: NO CCSID
    20 SQL-PVAR-NAME2 PIC S(4) COMP-4 VALUE +0.
    ```

728 Db2 12 for z/OS: Application Programming and SQL Guide
Db2 coprocessor: In the modified source from the Db2 coprocessor with the National Character Support for COBOL, hv1 and hv2 are represented to Db2 in the following way, with CCSIDs: (Assume that the source CCSID is 1140.)

| for hv1 and hv2, the value for CCSID is set to '1140' ('474'x) in input SQLDA of the INSERT statement. |
| '7F000047400000007F'x |

To ensure that no discrepancy exists between the column with FOR BIT DATA and the host variable with CCSID 1140, add the following statement for :hv1 or use the Db2 precompiler:

```
EXEC SQL DECLARE : hv1 VARIABLE FOR BIT DATA END-EXEC.
```

for hv1 declared with for bit data. The value in SQL---AVAR-NAME-DATA is set to ‘FFFF’x for CCSID instead of ‘474x’.

'7F0000FFFF000000007F'x  <<= with DECLARE :hv1 VARIABLE FOR BIT DATA vs. '7F000047400000007F'x   <<= without

– PL/I

Db2 coprocessor:
You can specify whether CCSIDs are to be associated with host variables by using the following PL/I SQL preprocessor options:

**CCSID0**
Specifies that the PL/I SQL preprocessor is not to set the CCSIDs for all host variables unless they are defined with the SQL DECLARE :hv VARIABLE statement.

**NOCCSID0**
Specifies that the PL/I SQL preprocessor is to set the CCSIDs for all host variables.

Related concepts
z/OS Unicode Services User’s Guide and Reference

Related reference
**Descriptions of SQL processing options**
You can specify any SQL processing options regardless of whether you use the Db2 precompiler or the Db2 coprocessor. However, the Db2 coprocessor might ignore certain options because host language compiler options exist that provide the same information.

**Enterprise COBOL for z/OS**
**SQL preprocessor options (PL/I)** (Enterprise PL/I for z/OS Programming Guide:)

Options for SQL statement processing
Use SQL processing options to specify how the Db2 precompiler and the Db2 coprocessor interpret and process input, and how they present output.

If you are using the Db2 precompiler, specify SQL processing options in one of the following ways:

- With DSNH operands
- With the PARM option of the EXEC JCL statement
- On DB2I panels

If you are using the Db2 coprocessor, specify SQL processing options in one of the following ways:

- For C or C++, specify the options as the argument of the SQL compiler option.
- For COBOL, specify the options as the argument of the SQL compiler option.
- For PL/I, specify the options as the argument of the PP(SQL('option,...')) compiler option.

For examples of how to specify the Db2 coprocessor options, see “Processing SQL statements by using the Db2 coprocessor” on page 846.
Db2 assigns default values for any SQL processing options for which you do not explicitly specify a value. Those defaults are the values that are specified on the APPLICATION PROGRAMMING DEFAULTS installation panels.

**Descriptions of SQL processing options**

You can specify any SQL processing options regardless of whether you use the Db2 precompiler or theDb2 coprocessor. However, the Db2 coprocessor might ignore certain options because host language compiler options exist that provide the same information.

The following table shows the options that you can specify when you use the Db2 precompiler or Db2 coprocessor. The table also includes abbreviations for those options and indicates which options are ignored for a particular host language or by the Db2 coprocessor. This table uses a vertical bar (|) to separate mutually exclusive options, and brackets ([ ]) to indicate that you can sometimes omit the enclosed option.

<table>
<thead>
<tr>
<th>Option keyword</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>APOST¹</td>
<td>Indicates that the Db2 precompiler is to use the apostrophe (') as the string delimiter in host language statements that it generates. This option is not available in all languages. APOST and QUOTE are mutually exclusive options. The default is in the field STRING DELIMITER on Application Programming Defaults Panel 1 during installation. If STRING DELIMITER is the apostrophe ('), APOST is the default.</td>
</tr>
<tr>
<td>APOSTSQL</td>
<td>Recognizes the apostrophe (') as the string delimiter and the double quotation mark (&quot;) as the SQL escape character within SQL statements. APOSTSQL and QUOTESQL are mutually exclusive options. The default is in the field SQL STRING DELIMITER on Application Programming Defaults Panel 1 during installation. If SQL STRING DELIMITER is the apostrophe ('), APOSTSQL is the default.</td>
</tr>
<tr>
<td>ATTACH(TSO</td>
<td>CAF</td>
</tr>
<tr>
<td><strong>Option keyword</strong></td>
<td><strong>Meaning</strong></td>
</tr>
<tr>
<td>-------------------</td>
<td>-------------</td>
</tr>
<tr>
<td><strong>CCSID(n)</strong></td>
<td>Specifies the numeric value ( n ) of the CCSID in which the source program is written. The number ( n ) must be an EBCDIC CCSID. The default setting is the EBCDIC system CCSID as specified on the panel DSNTIPF during installation. The Db2 coprocessor uses the following process to determine the CCSID of the source statements: 1. If the CCSID of the source program is specified by a compiler option, such as the COBOL CODEPAGE compiler option, the Db2 coprocessor uses that CCSID. If you also specify the CCSID suboption of the SQL compiler option that is different from the CCSID compiler option, a warning is returned, and the CCSID suboption value is not used. 2. If the CCSID is not specified by a compiler option: a. If the CCSID suboption of the SQL compiler option is specified and contains a valid EBCDIC CCSID, that CCSID is used. b. If the CCSID suboption of the SQL compiler option is not specified, and the compiler supports an option for specifying the CCSID, such as the COBOL CODEPAGE compiler option, the default for the CCSID compiler option is used. c. If the CCSID suboption of the SQL compiler option is not specified, and the compiler does not support an option for specifying the CCSID, the default CCSID from DSNHDECP or a user-specified application defaults module is used. d. If the CCSID suboption of the SQL option is specified and contains an invalid CCSID, compilation terminates. CCSID supersedes the GRAPHIC and NOGRAPHIC SQL processing options. If you specify CCSID(1026) or CCSID(1155), the Db2 coprocessor does not support the code point 'FC'X for the double quotation mark (&quot;).</td>
</tr>
<tr>
<td><strong>COMMA</strong></td>
<td>Recognizes the comma (,) as the decimal point indicator in decimal or floating point literals in the following cases: • For static SQL statements in COBOL programs • For dynamic SQL statements, when the value of installation parameter DYNRULS is NO and the package or plan that contains the SQL statements has DYNAMICRULES bind, define, or invoke behavior. COMMA and PERIOD are mutually exclusive options. The default (COMMA or PERIOD) is chosen under DECIMAL POINT IS on Application Programming Defaults Panel 1 during installation.</td>
</tr>
<tr>
<td>**CONNECT(2</td>
<td>1)**</td>
</tr>
<tr>
<td>Option keyword</td>
<td>Meaning</td>
</tr>
<tr>
<td>-------------------------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
</tbody>
</table>
| DATE(ISO|USA |EUR|JIS|LOCAL)                        | Specifies that date output should always be returned in a particular format, regardless of the format that is specified as the location default.  
The default is specified in the field DATE FORMAT on Application Programming Defaults Panel 2 during installation.  
The default format is determined by the installation defaults of the system where the program is bound, not by the installation defaults of the system where the program is precompiled.  
You cannot use the LOCAL option unless you have a date exit routine. |
| DEC(15|31) | DEC15 | DEC31 | D15.s | D31.s | Specifies the maximum precision for decimal arithmetic operations.  
The default is in the field DECIMAL ARITHMETIC on Application Programming Defaults Panel 1 during installation.  
If the form Dpp.s is specified, pp must be either 15 or 31, and s, which represents the minimum scale to be used for division, must be a number between 1 and 9. |
| DECP(name)                     | name represents the 1 to 8 character name of the application defaults data-only load module that is to be used.  
The default name DSNHDECP is used if this parameter is omitted. |
| FLAG(I|W|E|S)                        | Suppresses diagnostic messages below the specified severity level (Informational, Warning, Error, and Severe error for severity codes 0, 4, 8, and 12 respectively).  
The default setting is FLAG(I). |
| FLOAT(S390|IEEE)                    | Determines whether the contents of floating-point host variables in assembler, C, C++, or PL/I programs are in IEEE floating-point format or z/Architecture hexadecimal floating-point format. Db2 ignores this option if the value of HOST is anything other than ASM, C, CPP, or PLI.  
The default setting is FLOAT(S390). |
| GRAPHIC                       | This option is no longer used for SQL statement processing. Use the CCSID option instead.  
Indicates that the source code might use mixed data, and that X’0E’and X’0F’ are special control characters (shift-out and shift-in) for EBCDIC data.  
GRAPHIC and NOGRAPHIC are mutually exclusive options. The default (GRAPHIC or NOGRAPHIC) is specified in the field MIXED DATA on Application Programming Defaults Panel 1 during installation. |
### Table 139. SQL processing options (continued)

<table>
<thead>
<tr>
<th>Option keyword</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>HOST(^1)(ASM</td>
<td>C[(FOLD)]</td>
</tr>
<tr>
<td>LEVEL(^1)(aaaa) L</td>
<td>Defines the level of a module, where aaaa is any alphanumeric value of up to seven characters. This option is not recommended for general use, and the DSNH CLIST and the DB2I panels do not support it. For assembler, C, C++, Fortran, and PL/I, you can omit the suboption (aaaa). The resulting consistency token is blank. For COBOL, you need to specify the suboption.</td>
</tr>
<tr>
<td>LINECOUNT(^1)(n) LC</td>
<td>Defines the number of lines per page to be n for the Db2 precompiler listing. This includes header lines that are inserted by the Db2 precompiler. The default setting is LINECOUNT(60).</td>
</tr>
<tr>
<td>MARGINS(^1)(m,n[,c]) MAR</td>
<td>Specifies what part of each source record contains host language or SQL statements. For assembler, this option also specifies where column continuations begin. The first option (m) is the beginning column for statements. The second option (n) is the ending column for statements. The third option (c) specifies where assembler continuations begin. Otherwise, the Db2 precompiler places a continuation indicator in the column immediately following the ending column. Margin values can range 1 - 80. Default values depend on the HOST option that you specify. The DSNH CLIST and the DB2I panels do not support this option. In assembler, the margin option must agree with the ICTL instruction, if presented in the source.</td>
</tr>
<tr>
<td>Option keyword</td>
<td>Meaning</td>
</tr>
<tr>
<td>----------------</td>
<td>---------</td>
</tr>
</tbody>
</table>
| **NEWFUN(Vn)** | **Deprecated function:** The NEWFUN processing option is deprecated. Use the SQLLEVEL option instead.  
Indicates whether to accept the function syntax that is new for Db2 12. |
| **NEWFUN(V12)** | Specifies that any syntax up to Db2 12 is allowed. This value is equivalent to function level V12R1M501. |
| **NEWFUN(V11)** | Specifies that any syntax up to Db2 11 is allowed. |
| **NEWFUN(V10)** | Specifies that any syntax up to DB2 10 is allowed. |
| **NEWFUN(V9)** | Specifies that any syntax up to DB2 9 is allowed. DB2 9 is supported, but causes the precompilation process to support only a DB2 9 level of function. |
| **NEWFUN(V8)** | Specifies that any syntax up to DB2 Version 8 is allowed. V8 is supported, but causes the precompilation process to support only a V8 level of function. |
| **NOFOR** | In static SQL, eliminates the need for the FOR UPDATE or FOR UPDATE OF clause in DECLARE CURSOR statements. When you use NOFOR, your program can make positioned updates to any columns that the program has Db2 authority to update.  
When you do not use NOFOR, if you want to make positioned updates to any columns that the program has Db2 authority to update, you need to specify FOR UPDATE with no column list in your DECLARE CURSOR statements. The FOR UPDATE clause with no column list applies to static or dynamic SQL statements.  
Regardless of whether you use NOFOR, you can specify FOR UPDATE OF with a column list to restrict updates to only the columns that are named in the clause, and you can specify the acquisition of update locks.  
You imply NOFOR when you use the option STDSQL(YES).  
If the resulting DBRM is very large, you might need extra storage when you specify NOFOR or use the FOR UPDATE clause with no column list. |
| **NOGRAPHIC** | This option is no longer used for SQL statement processing. Use the CCSID option instead.  
Indicates the use of X'0E' and X'0F' in a string, but not as control characters.  
GRAPHIC and NOGRAPHIC are mutually exclusive options. The default (GRAPHIC or NOGRAPHIC) is specified in the field MIXED DATA on Application Programming Defaults Panel 1 during installation.  
The NOGRAPHIC option applies to only EBCDIC data. |
| **NOOPTIONS** **NOOPTN** | Suppresses the Db2 precompiler options listing. |
### Table 139. SQL processing options (continued)

<table>
<thead>
<tr>
<th>Option keyword</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>NOPADNTSTR</td>
<td>Indicates that output host variables that are NUL-terminated strings are <strong>not</strong> padded with blanks. That is, additional blanks are not inserted before the NUL-terminator is placed at the end of the string. PADNTSTR and NOPADNTSTR are mutually exclusive options. The default (PADNTSTR or NOPADNTSTR) is specified in the field PAD NUL-TERMINATED on Application Programming Defaults Panel 2 during installation. This option applies to only C and C++ applications.</td>
</tr>
<tr>
<td>NOSOURCE2 NOS</td>
<td>Suppresses the Db2 precompiler source listing. This is the default.</td>
</tr>
<tr>
<td>NOXREF</td>
<td>Suppresses the Db2 precompiler cross-reference listing. This is the default.</td>
</tr>
<tr>
<td>ONEPASS ON</td>
<td>Processes in one pass, to avoid the additional processing time for making two passes. Declarations must appear before SQL references. Default values depend on the HOST option specified. ONEPASS and TWOPASS are mutually exclusive options.</td>
</tr>
<tr>
<td>OPTIONS1 OPTN</td>
<td>Lists Db2 precompiler options. This is the default.</td>
</tr>
<tr>
<td>PADNTSTR</td>
<td>Indicates that output host variables that are NUL-terminated strings are padded with blanks with the NUL-terminator placed at the end of the string. PADNTSTR and NOPADNTSTR are mutually exclusive options. The default (PADNTSTR or NOPADNTSTR) is specified in the field PAD NUL-TERMINATED on Application Programming Defaults Panel 2 during installation. This option applies to only C and C++ applications.</td>
</tr>
<tr>
<td>PERIOD</td>
<td>Recognizes the period (.) as the decimal point indicator in decimal or floating point literals in the following cases: • For static SQL statements in COBOL programs • For dynamic SQL statements, when the value of installation parameter DYNRULS is NO and the package or plan that contains the SQL statements has DYNAMICRULES bind, define, or invoke behavior. COMMA and PERIOD are mutually exclusive options. The default (COMMA or PERIOD) is specified in the field DECIMAL POINT IS on Application Programming Defaults Panel 1 during installation.</td>
</tr>
<tr>
<td>Option keyword</td>
<td>Meaning</td>
</tr>
<tr>
<td>----------------</td>
<td>---------</td>
</tr>
<tr>
<td><strong>QUOTE</strong> 1 Q</td>
<td>Indicates that the Db2 precompiler is to use the quotation mark (&quot;) as the string delimiter in host language statements that it generates. QUOTE is valid only for COBOL applications. QUOTE is not valid for either of the following combinations of precompiler options: • CCSID(1026) and HOST(IBMCOB) • CCSID(1155) and HOST(IBMCOB) The default is specified in the field STRING DELIMITER on Application Programming Defaults Panel 1 during installation. If STRING DELIMITER is the double quotation mark (&quot;) or DEFAULT, QUOTE is the default. APOST and QUOTE are mutually exclusive options.</td>
</tr>
<tr>
<td><strong>QUOTESQL</strong></td>
<td>Recognizes the double quotation mark (&quot;) as the string delimiter and the apostrophe (’) as the SQL escape character within SQL statements. This option applies only to COBOL. The default is specified in the field SQL STRING DELIMITER on Application Programming Defaults Panel 1 during installation. If SQL STRING DELIMITER is the double quotation mark (&quot;) or DEFAULT, QUOTESQL is the default. APOSTSQL and QUOTESQL are mutually exclusive options.</td>
</tr>
<tr>
<td><strong>SOURCE</strong> 1 S</td>
<td>Lists Db2 precompiler source and diagnostics.</td>
</tr>
<tr>
<td>**SQL(ALL</td>
<td>DB2)**</td>
</tr>
</tbody>
</table>
### Table 139. SQL processing options (continued)

<table>
<thead>
<tr>
<th>Option keyword</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>**SQLLEVEL(V10R1</td>
<td>V11R1</td>
</tr>
<tr>
<td><strong>SQLLEVEL(function-level)</strong></td>
<td>Specifies the function level allowed by the precompilation process. The format is VvvRrmmm, where vv is the version, r is the release, and mmm is the modification level. SQLLEVEL V12R1M100 is equivalent to V11R1.</td>
</tr>
<tr>
<td><strong>SQLLEVEL(V11R1)</strong></td>
<td>Specifies that any SQL syntax up to Db2 11 is allowed.</td>
</tr>
<tr>
<td><strong>SQLLEVEL(V10R1)</strong></td>
<td>Specifies that any SQL syntax up to DB2 10 is allowed.</td>
</tr>
<tr>
<td><strong>SQLLEVEL(V9R1)</strong></td>
<td>Specifies that any SQL syntax up to DB2 9 is allowed. DB2 9 is supported, but causes the precompilation process to support only a DB2 9 level of SQL syntax.</td>
</tr>
<tr>
<td><strong>SQLLEVEL(V8R1)</strong></td>
<td>Specifies that any SQL syntax up to DB2 Version 8 is allowed. DB2 Version 8 is supported, but causes the precompilation process to support only a DB2 Version 8 level of SQL syntax.</td>
</tr>
<tr>
<td></td>
<td>The function level activated on the Db2 subsystem does not restrict the SQLLEVEL value. However, you must ensure that you bind the resulting DBRM with the correct application compatibility level on a Db2 subsystem with the correct function level activated.</td>
</tr>
<tr>
<td>**STDSQL(NO</td>
<td>YES)**</td>
</tr>
<tr>
<td></td>
<td>The default is specified in the field STD SQL LANGUAGE on Application Programming Defaults Panel 2 during installation. STDSQL=YES automatically implies the NOFOR option.</td>
</tr>
<tr>
<td>**TIME(ISO</td>
<td>USA</td>
</tr>
<tr>
<td></td>
<td>The default is specified in the field TIME FORMAT on Application Programming Defaults Panel 2 during installation.</td>
</tr>
<tr>
<td></td>
<td>The default format is determined by the installation defaults of the system where the program is bound, not by the installation defaults of the system where the program is precompiled.</td>
</tr>
<tr>
<td></td>
<td>You cannot use the LOCAL option unless you have a time exit routine.</td>
</tr>
<tr>
<td><strong>TWOPASS</strong></td>
<td>Processes in two passes, so that declarations need not precede references. Default values depend on the HOST option that is specified. TWOPASS and TWOPASS are mutually exclusive options. For the Db2 coprocessor, you can specify the TWOPASS option for only PL/I applications. For C/C++ and COBOL applications, the Db2 coprocessor uses the ONEPASS option.</td>
</tr>
</tbody>
</table>
### Table 139. SQL processing options (continued)

<table>
<thead>
<tr>
<th>Option keyword</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>VERSION(aaaa</td>
<td>AUTO)</td>
</tr>
<tr>
<td>XREF</td>
<td>Includes a sorted cross-reference listing of symbols that are used in SQL statements in the listing output.</td>
</tr>
</tbody>
</table>

**Notes:**

1. The Db2 coprocessor ignores this option when the Db2 coprocessor is invoked by the compiler to prepare the application.
2. This option is always in effect when the Db2 coprocessor is invoked by the compiler to prepare the application.
3. You can use STDSQL(86) as in prior releases of Db2. The SQL statement processor treats it the same as STDSQL(YES).
4. Precompiler options do not affect ODBC behavior.
5. The Db2 coprocessor ignores this option when the Db2 coprocessor is invoked by the compiler to prepare the application. However, if you are using PL/I V4.1 or later, it is supported.

**Related concepts**

- Precision for operations with decimal numbers
- Datetime values (Db2 SQL)

**Related tasks**

- Creating a package version
- Creating a package version
- Setting the program level

**Related reference**

- Defaults for SQL processing options
Some SQL statement processing options have default values that are based on values that are specified on the DB2I Application Programming Defaults panels.

### Defaults for SQL processing options

Some SQL statement processing options have default values that are based on values that are specified on the DB2I Application Programming Defaults panels.

The following table shows those options and defaults.

<table>
<thead>
<tr>
<th>Install option</th>
<th>Install default</th>
<th>Equivalent SQL statement processing option</th>
<th>Available SQL statement processing options</th>
</tr>
</thead>
<tbody>
<tr>
<td>STRING DELIMITER</td>
<td>quotation mark (&quot;)</td>
<td>QUOTE</td>
<td>APOSTQUOTE</td>
</tr>
<tr>
<td>SQL STRING DELIMITER</td>
<td>quotation mark (&quot;)</td>
<td>QUOTESQL</td>
<td>APOSTSQLQUOTESQL</td>
</tr>
<tr>
<td>DECIMAL POINT IS</td>
<td>PERIOD</td>
<td>PERIOD</td>
<td>COMMAPERIOD</td>
</tr>
<tr>
<td>DATE FORMAT</td>
<td>ISO</td>
<td>DATE(ISO)</td>
<td>DATE(ISO</td>
</tr>
<tr>
<td>DECIMAL ARITHMETIC</td>
<td>DEC15</td>
<td>DEC(15)</td>
<td>DEC(15</td>
</tr>
<tr>
<td>MIXED DATA</td>
<td>NO</td>
<td>CCSID(n)</td>
<td>CCSID(n)</td>
</tr>
<tr>
<td>LANGUAGE DEFAULT</td>
<td>COBOL</td>
<td>HOST(COBOL)</td>
<td>HOST(ASM</td>
</tr>
<tr>
<td>STD SQL LANGUAGE</td>
<td>NO</td>
<td>STDSQL(NO)</td>
<td>STDSQL(YES</td>
</tr>
<tr>
<td>TIME FORMAT</td>
<td>ISO</td>
<td>TIME(ISO)</td>
<td>TIME(IS</td>
</tr>
</tbody>
</table>

**Notes:** For dynamic SQL statements, another application programming default, USE FOR DYNAMICRULES, determines whether Db2 uses the application programming default or the SQL statement processor option for the following installation options:

- STRING DELIMITER
- SQL STRING DELIMITER
- DECIMAL POINT IS
- DECIMAL ARITHMETIC

If the value of USE FOR DYNAMICRULES is YES, dynamic SQL statements use the application programming defaults. If the value of USE FOR DYNAMICRULES is NO, dynamic SQL statements in packages or plans with bind, define, and invoke behavior use the SQL statement processor options.

Some SQL statement processor options have default values based on the host language. Some options do not apply to some languages. The following table shows the language-dependent options and defaults.

<table>
<thead>
<tr>
<th>HOST value</th>
<th>Defaults</th>
</tr>
</thead>
<tbody>
<tr>
<td>ASM</td>
<td>APOST$^1$, APOSTSQL$^1$, PERIOD$^1$, TWOPASS, MARGINS(1,71,16)</td>
</tr>
<tr>
<td>C or CPP</td>
<td>APOST$^1$, APOSTSQL$^1$, PERIOD$^1$, ONEPASS, MARGINS(1,72)</td>
</tr>
<tr>
<td>IBMCOB</td>
<td>QUOTE$^2$, QUOTESQL$^2$, PERIOD, ONEPASS$^1$, MARGINS(8,72)$^1$</td>
</tr>
</tbody>
</table>

---

Chapter 9. Preparing an application to run on Db2 for z/OS 861
Table 141. Language-dependent Db2 precompiler options and defaults (continued)

<table>
<thead>
<tr>
<th>HOST value</th>
<th>Defaults</th>
</tr>
</thead>
<tbody>
<tr>
<td>FORTRAN</td>
<td>APOST(^1), APOSTSQL(^1), PERIOD(^1), ONEPASS(^1), MARGINS(1,72)(^1)</td>
</tr>
<tr>
<td>PLI</td>
<td>APOST(^1), APOSTSQL(^1), PERIOD(^1), ONEPASS, MARGINS(2,72)</td>
</tr>
<tr>
<td>SQL or SQLPL</td>
<td>APOST(^1), APOSTSQL(^1), PERIOD(^1), ONEPASS, MARGINS(1,72)</td>
</tr>
</tbody>
</table>

Notes:
1. Forced for this language; no alternative is allowed.
2. The default is chosen on Application Programming Defaults Panel 1 during installation. The IBM-supplied installation defaults for string delimiters are QUOTE (host language delimiter) and QUOTESQL (SQL escape character). The installer can replace the IBM-supplied defaults with other defaults. The precompiler options that you specify override any defaults that are in effect.

**SQL statement processing defaults for dynamic statements**

Generally, dynamic statements use the defaults that are specified during installation. However, if the value of application defaults module parameter DYNRULES is NO, you can use these options for dynamic SQL statements in packages or plans with bind, define, or invoke behavior:

- COMMA or PERIOD
- APOST or QUOTE
- APOSTSQL or QUOTESQL
- DEC(15) or DEC(31)

**Related concepts**

Dynamic rules options for dynamic SQL statements
The DYNAMICRULES bind option and the runtime environment determine the rules for the dynamic SQL attributes.

**SQL options for DRDA access**

Certain SQL statement processing options are relevant when you prepare a package to be run with DRDA access.

The following SQL statement processing options are relevant for DRDA access:

**CONNECT**

- Use CONNECT(2), explicitly or by default.

  CONNECT(1) causes your CONNECT statements to allow only the restricted function known as "remote unit of work". Be particularly careful to avoid CONNECT(1) if your application updates more than one DBMS in a single unit of work.

**SQL**

- Use SQL(ALL) explicitly for a package that runs on a server that is not Db2 for z/OS. The precompiler then accepts any statement that obeys DRDA rules.

- Use SQL(DB2), explicitly or by default, if the server is Db2 for z/OS only. The precompiler then rejects any statement that does not obey the rules of Db2 for z/OS.
Compiling and link-editing an application

If you use the Db2 precompiler, your next step in the program preparation process is to compile and link-edit your program. As with the precompile step, you have a choice of methods.

About this task

You can use one of the following methods to compile and link-edit an application:

- DB2I panels. For details, see “DB2I panels that are used for program preparation” on page 900.
- The DSNH command procedure (a TSO CLIST). For details, see DSNH (TSO CLIST) (Db2 Commands).
- JCL procedures supplied with Db2. For details, see “Db2-supplied JCL procedures for preparing an application” on page 897.
- JCL procedures supplied with a host language compiler.

If you use the Db2 coprocessor, you process SQL statements as you compile your program. For programs other than C and C++ programs, you must use JCL procedures when you use the Db2 coprocessor. For C and C++ programs, you can use either JCL procedures or UNIX System Services on z/OS to invoke the Db2 coprocessor.

The purpose of the link-edit step is to produce an executable load module. To enable your application to interface with the Db2 subsystem, you must use a link-edit procedure that builds a load module that satisfies environment-specific requirements.

**TSO and batch**

Include the Db2 TSO attachment facility language interface module (DSNELI) or Db2 call attachment facility language interface module (DSNALI) or the Universal Language Interface module (DSNULI).

**IMS**

Include the Db2 IMS language interface module (DFSLI000), which contains the DSNHLI entry point. Also, the IMS RESLIB must precede the SDSNLOAD library in the link list, JOBLIB, or STEPLIB concatenations.

IMS and Db2 share a common alias name, DSNHLI, for the language interface module. You must do the following when you concatenate your libraries:

- If you use IMS, be sure to concatenate the IMS library first so that the application program compiles with the correct IMS version of DSNHLI.
- If you run your application program only under Db2, be sure to concatenate the Db2 library first.

**CICS**

Include the Db2 CICS language interface module (DSNCLI) or the Universal Language Interface module (DSNULI). You can link DSNCLI with your program in either 24-bit or 31-bit addressing mode (AMODE=31), but DSNULI must be linked with your program in 31-bit addressing mode (AMODE=31).

If your application runs in 31-bit addressing mode, you should link-edit the DSNCLI or DSNULI stub to your application with the attributes AMODE=31 and RMODE=ANY so that your application can run above the 16-MB line.

You also need the CICS EXEC interface module that is appropriate for the programming language. CICS requires that this module be the first control section (CSECT) in the final load module.

The size of the executable load module that is produced by the link-edit step varies depending on the values that the SQL statement processor inserts into the source code of the program.

**Link-editing a batch program**

Db2 has language interface routines for each unique supported environment. Db2 requires the IMS language interface routine for DL/I batch. You need to have DFSLI000 link-edited with the application program.

Related concepts

Universal language interface (DSNULI)
The universal language interface (DSNULI) subcomponent determines the runtime environment and dynamically loads and branches to the appropriate language interface module.

**Related tasks**
Preparing an application to run on Db2 for z/OS
To prepare and run applications that contain embedded static SQL statements or dynamic SQL statements, you must precompile, compile, link-edit, and bind them.

**Related reference**
DSNH (TSO CLIST) (Db2 Commands)

**Related information**
CICS program preparation steps (CICS Transaction Server for z/OS)

---

## Binding application packages and plans

You must bind the DBRM that is produced by the SQL statement processor to a package before your Db2 application can run. The bind process establishes a relationship between an application program and its relational data.

**Before you begin**
You must have appropriate privileges. For more information, see Privileges required for handling plans and packages (Managing Security)

**About this task**
During the precompilation process, the Db2 precompiler produces both modified source code and a database request module (DBRM) for each application program. The modified source code must be compiled and link-edited before the program can be run. DBRMs must be bound to a package. You can then associate that package with a particular application plan.

During the bind process, Db2 also completes the following actions:

- Validates object references in the SQL statements of the program, such as table, view, and column names, against the Db2 catalog. Because the bind process occurs before program execution, errors are detected and can be corrected before the program is executed.
- Verifies the authorization of the bind process to specify the program owner and the authorization of the specified owner to access data that is requested by SQL statements in the program.
- Selects the access paths that Db2 uses to access data for the program. Db2 considers factors such as table size, available indexes, and others, when selecting the access paths.

When determining the maximum size of a plan, you must consider several physical limitations, including the time required to bind the plan, the size of the EDM pool, and fragmentation. As a general rule, the EDM pool should be at least 10 times the size of the largest DBD or plan, whichever is greater.

Each package that you bind can contain only one DBRM.

**Exception:** You do not need to bind a DBRM if the only SQL statement in the program is SET CURRENT PACKAGESET.

Because you do not need a plan or package to execute the SET CURRENT PACKAGESET statement, the ENCODING bind option does not affect the SET CURRENT PACKAGESET statement. An application that needs to provide a host variable value in an encoding scheme other than the system default encoding scheme must use the DECLARE VARIABLE statement to specify the encoding scheme of the host variable.

You must bind plans locally, regardless of whether they reference packages that run remotely. However, you must bind the packages that run at remote locations at those remote locations.

From a Db2 requester, you can run a plan by specifying it in the RUN subcommand, but you cannot run a package directly. You must include the package in a plan and then run the plan.
Tip: Develop a naming convention and strategy for the most effective and efficient use of your plans and packages.

Procedure

To bind application programs, take the following actions.

1. To bind individual DBRMs into packages, use BIND PACKAGE commands with ACTION(REPLACE).
   Packages provide the flexibility for you to test different versions of a program without having to rebind everything in the application plan.
   
   For programs whose corresponding DBRMs are in HFS files, you can use the command line processor to bind the DBRMs to packages. Optionally, you can also copy the DBRM into a partitioned data set member by using the oput and oget commands and then bind it by using conventional JCL.
   
   To create new trigger packages for existing triggers, you must re-create the trigger that is associated with the package. For more information, see “Trigger packages” on page 160.

2. To designate packages in application plans, use the BIND PLAN command with ACTION(REPLACE).
   Plans can specify packages, collections of packages, or a combination of these elements. If you specify one or more DBRMs to include in the plan (by using the MEMBER option of BIND PLAN), Db2 automatically binds those DBRMs into packages and then binds those packages into the plan. The plan contains information about the designated packages and about the data that the application programs intend to use. The plan is stored in the Db2 catalog.

Related concepts

Package copies for plan management (Db2 Performance)

Automatic rebinds

Automatic rebinds (sometimes called “autobinds”) occur when an authorized user runs a package or plan and the runtime structures in the plan or package cannot be used. This situation usually results from changes to the attributes of the data on which the package or plan depends, or changes to the environment in which the package or plan runs.

Related tasks

Binding a DBRM that is in an HFS file to a package or collection

If DBRMs are in z/OS UNIX HFS files, you can use the command line processor to bind the DBRMs to packages at the target Db2 server. Optionally, you can also copy the DBRM into a partitioned data set member by using the TSO/E oput and oget commands and then bind the DBRM by using conventional JCL.

Related reference

BIND PACKAGE (DSN) (Db2 Commands)
BIND PLAN (DSN) (Db2 Commands)
BIND and REBIND options for packages, plans, and services (Db2 Commands)

Creating a package version

If you want to run different versions of a program without needing to make changes to the associated application plan, use package versions. This technique is useful if you need to make changes to your program without causing an interruption to the availability of the program.

About this task

You can create a different package version for each version of the program. Each package has the same package name and collection name, but a different version number is associated with each package. The plan that includes that package includes all versions of that package. Thus, you can run a program that is associated with any one of the package versions without having to rebind the application plan, rename the plan, or change any RUN subcommands that use it.

Procedure

To create a package version:
1. Precompile your program with the option VERSION(version-identifier).
2. Bind the resulting DBRM with the same collection name and package name as any existing versions of that package. When you run the program, Db2 uses the package version that you specified when you precompiled it.

**Example**

Suppose that you bound a plan with the following statement:

```sql
BIND PLAN (PLAN1) PKLIST (COLLECT.*)
```

The following steps show how to create two versions of a package, one for each of two programs.

<table>
<thead>
<tr>
<th>Step number</th>
<th>For package version 1</th>
<th>For package version 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>Bind the DBRM with the collection name COLLECT and the package name PACKA.</td>
<td>Bind the DBRM with the collection name COLLECT and package name PACKA.</td>
</tr>
<tr>
<td>3</td>
<td>Link-edit program 1 into your application.</td>
<td>Link-edit program 2 into your application.</td>
</tr>
<tr>
<td>4</td>
<td>Run the application; it uses program 1 and PACKA, VERSION 1.</td>
<td>Run the application; it uses program 2 and PACKA, VERSION 2.</td>
</tr>
</tbody>
</table>

**Binding a DBRM that is in an HFS file to a package or collection**

If DBRMs are in z/OS UNIX HFS files, you can use the command line processor to bind the DBRMs to packages at the target Db2 server. Optionally, you can also copy the DBRM into a partitioned data set member by using the TSO/E oput and oget commands and then bind the DBRM by using conventional JCL.

**About this task**

**Restrictions:**

You cannot specify the REBIND command with the command line processor. Alternatively, specify the BIND command with the ACTION(REPLACE) option.

You cannot specify the FREE PACKAGE command with the command line processor. Alternatively, specify the DROP PACKAGE statement to drop the existing packages.

**Procedure**

To bind a DBRM that is in an HFS file to a package or collection:

1. Invoke the command line processor and connect to the target Db2 server.
2. Specify the BIND command with the appropriate options.

**Related concepts**

The Db2 command line processor (Db2 Commands)

**Related tasks**

Processing SQL statements by using the Db2 coprocessor

As an alternative to the Db2 precompiler, you can use the Db2 coprocessor to process SQL statements. The Db2 coprocessor performs Db2 precompiler functions at compile time.

**Related reference**

Command line processor BIND command
Use the command line processor BIND command to bind DBRMs that are in z/OS UNIX HFS files to packages.

**Command line processor BIND command**

Use the command line processor BIND command to bind DBRMs that are in z/OS UNIX HFS files to packages.

The following diagram shows the syntax for the command line processor BIND command.

```
BIND < dbrm-file-name > -COLLECTION < collection-name > options-clause
```

Notes:

1. If you do not specify a collection, Db2 uses NULLID.
2. You can specify the options after `collection-name` in any order.

**options-clause:**

Notes:

1 You can specify NOREOPT(VARS) as a synonym of REOPT(NONE).
2 You can specify REOPT(VARS) as a synonym of REOPT(ALWAYS).
The following options are unique to this diagram:

**CURRENTDATA (ALL)**  
Specifies that for all cursors data currency is required and block fetching is inhibited.

**SQLERROR(CHECK)**  
Specifies that the command line processor is to only check for SQL errors in the DBRM. No package is generated.

**IMMEDWRITE(PH1)**  
Specifies that normal write activity is done. This option is equivalent to IMMEDWRITE(NO).

**EXPLAIN(ALL)**  
Specifies that Db2 is to insert information into the appropriate EXPLAIN tables. This option is equivalent to EXPLAIN (YES).

**Related reference**  
BIND and REBIND options for packages, plans, and services (Db2 Commands)

### Binding an application plan

An application plan can include package lists.

**Procedure**

To bind an application plan, use the BIND PLAN subcommand with at least one of the following options:

**MEMBER**  
Specify this option to bind DBRMs to a package and then bind the package list to a plan. After the keyword MEMBER, specify the member names of the DBRMs.

**PKLIST**  
Specify this option to include package lists in the plan. After the keyword PKLIST, specify the names of the packages to include in the package list. To include an entire collection of packages in the list, use an asterisk after the collection name. For example, PKLIST (GROUP1.*).

**Specifying the package list for the PKLIST option of BIND PLAN**

The order in which you specify packages in a package list can affect run time performance. Searching for the specific package involves searching the Db2 directory, which can be costly. When you use `collection-id.*` with the PKLIST keyword, you should specify first the collections in which Db2 is most likely to find a package.

For example, assume that you perform the following bind:

```
BIND PLAN (PLAN1) PKLIST (COLL1.*, COLL2.*, COLL3.*, COLL4.*)
```

Then you execute program PROG1. Db2 does the following package search:

a. Checks to see if program PROG1 is bound as part of the plan  
b. Searches for COLL1.PROG1.timestamp  
c. If it does not find COLL1.PROG1.timestamp, searches for COLL2.PROG1.timestamp  
d. If it does not find COLL2.PROG1.timestamp, searches for COLL3.PROG1.timestamp
e. If it does not find COLL3.PROG1.<timestamp>, searches for COLL4.PROG1.<timestamp>.

When both special registers CURRENT PACKAGE PATH and CURRENT PACKAGESET contain an empty string

If you do not set these special registers, Db2 searches for a DBRM or a package in one of these sequences:

- At the local location (if CURRENT SERVER is blank or specifies that location explicitly), the order is:
  a. All packages that are already allocated to the plan while the plan is running.
  b. All unallocated packages that are explicitly specified in, and all collections that are completely included in, the package list of the plan. Db2 searches for packages in the order that they appear in the package list.

- At a remote location, the order is:
  a. All packages that are already allocated to the plan at that location while the plan is running.
  b. All unallocated packages that are explicitly specified in, and all collections that are completely included in, the package list of the plan, whose locations match the value of CURRENT SERVER. Db2 searches for packages in the order that they appear in the package list.

If you use the BIND PLAN option DEFER(PREPARE), Db2 does not search all collections in the package list.

If the order of search is not important

In many cases, the order in which Db2 searches the packages is not important to you and does not affect performance. For an application that runs only at your local Db2 system, you can name every package differently and include them all in the same collection. The package list on your BIND PLAN subcommand can read:

```plaintext
PKLIST (collection.*)
```

The resulting plan consists of the following information:

- Any programs that are associated with DBRMs in the MEMBER list
- Any programs that are associated with packages and collections that are identified in PKLIST

You can add packages to the collection even after binding the plan. Db2 lets you bind packages having the same package name into the same collection only if their version IDs are different.

If your application uses DRDA access, you must bind some packages at remote locations. Use the same collection name at each location, and identify your package list as:

```plaintext
PKLIST (*.collection.*)
```

If you use an asterisk for part of a name in a package list, Db2 checks the authorization for the package to which the name resolves at run time. To avoid the checking at run time in the preceding example, you can grant EXECUTE authority for the entire collection to the owner of the plan before you bind the plan.

Related tasks

- Improving performance for applications that access distributed data (Db2 Performance)

Related reference

- BIND PLAN (DSN) (Db2 Commands)
- CURRENT PACKAGE PATH (Db2 SQL)
- CURRENT PACKAGESET (Db2 SQL)
How Db2 identifies packages at run time

The Db2 precompiler or Db2 coprocessor identifies each call to Db2 with a consistency token. The same consistency token identifies the DBRM that the SQL statement processor produces and the package to which you bound the DBRM.

When you run the program, Db2 uses the consistency token in matching the call to Db2 to the correct DBRM. Usually, the consistency token is in an internal Db2 format. You can override that token if you want.

You also need other identifiers. The consistency token alone does not necessarily identify a unique package. You can bind the same DBRM to many packages, at different locations and in different collections, and you can include all those packages in the package list of the same plan. All those packages will have the same consistency token. You can specify a particular location or a particular collection at run time.

Related tasks
Setting the program level
The program level defines the level for a particular module. This information is stored in the consistency token, which is in an internal Db2 format. Overriding the program level in the consistency token is possible, if needed, but generally not recommended.

Specifying the location of the package that Db2 is to use

When your program executes SQL statements, Db2 uses the value in the CURRENT SERVER special register to determine the location of the necessary package. If the current server is your local Db2 subsystem and it does not have a location name, the value in the special register is blank.

About this task
You can change the value of CURRENT SERVER by using the SQL CONNECT statement in your program. If you do not use CONNECT, the value of CURRENT SERVER is the location name of your local Db2 subsystem (or blank, if your Db2 subsystem has no location name).

Specifying the package collection that Db2 is to use

To ensure that Db2 uses the intended package collection and does not waste time searching, explicitly specify the package collection that you want Db2 to use.

About this task
You can use the special register CURRENT PACKAGE PATH or CURRENT PACKAGESET (if CURRENT PACKAGE PATH is not set) to specify the collections that are to be used for package resolution. The CURRENT PACKAGESET special register contains the name of a single collection, and the CURRENT PACKAGE PATH special register contains a list of collection names.

If you do not set these registers, they contain an empty string when your application begins to run, and they remain as an empty string. In this case, Db2 searches the available collections.

However, explicitly specifying the intended collection by using the special registers can avoid a potentially costly search through a package list that has many qualifying entries. In addition, Db2 uses the values in these special registers for applications that do not run under a plan.

When you call a stored procedure, the special register CURRENT PACKAGESET contains the value that you specified for the COLLID parameter when you defined the stored procedure. If the routine was defined without a value for the COLLID parameter, the value of the special register is inherited from the calling program. Also, the special register CURRENT PACKAGE PATH contains the value that you specified for the PACKAGE PATH parameter when you defined the stored procedure. When the stored procedure returns control to the calling program, Db2 restores this register to the value that it contained before the call.

Specifying the package collection for a Java routine in a JAR file that is installed in the Db2 catalog:
If the Java routine definition specifies a COLLID value that is different from the collection into which the IBM Data Server Driver for JDBC and SQLJ packages are bound, you need to run the DB2Binder utility with
the -collection option to bind the driver packages into the collection that is specified by the COLLID value in the Java routine definition. If you do not do so, you might receive SQLCODE -805.

Related tasks

Binding an application plan
An application plan can include package lists.

Overriding the values that Db2 uses to resolve package lists
Db2 resolves package lists by searching the available collections in a particular order. To avoid this search, you can specify the values that Db2 should use for package resolution.

Related reference

DB2Binder utility (Db2 Application Programming for Java)

Overriding the values that Db2 uses to resolve package lists
Db2 resolves package lists by searching the available collections in a particular order. To avoid this search, you can specify the values that Db2 should use for package resolution.

About this task

If you set the special register CURRENT PACKAGE PATH or CURRENT PACKAGESET, Db2 skips the check for programs that are part of a plan and uses the values in these registers for package resolution.

If you set CURRENT PACKAGE PATH, Db2 uses the value of CURRENT PACKAGE PATH as the collection name list for package resolution. For example, if CURRENT PACKAGE PATH contains the list COLL1, COLL2, COLL3, COLL4, Db2 searches for the first package that exists in the following order:

COLL1.PROG1.timestamp
COLL2.PROG1.timestamp
COLL3.PROG1.timestamp
COLL4.PROG1.timestamp

If you set CURRENT PACKAGESET and not CURRENT PACKAGE PATH, Db2 uses the value of CURRENT PACKAGESET as the collection for package resolution. For example, if CURRENT PACKAGESET contains COLLS5, Db2 uses COLLS5.PROG1.timestamp for the package search.

When CURRENT PACKAGE PATH is set, the server that receives the request ignores the collection that is specified by the request and instead uses the value of CURRENT PACKAGE PATH at the server to resolve the package. Specifying a collection list with the CURRENT PACKAGE PATH special register can avoid the need to issue multiple SET CURRENT PACKAGESET statements to switch collections for the package search.

The following table shows examples of the relationship between the CURRENT PACKAGE PATH special register and the CURRENT PACKAGESET special register.

<table>
<thead>
<tr>
<th>Example</th>
<th>What happens</th>
</tr>
</thead>
<tbody>
<tr>
<td>SET CURRENT PACKAGESET SELECT ... FROM T1 ...</td>
<td>The collection in PACKAGESET determines which package is invoked.</td>
</tr>
<tr>
<td>SET CURRENT PACKAGE PATH SELECT ... FROM T1 ...</td>
<td>The collections in PACKAGE PATH determine which package is invoked.</td>
</tr>
<tr>
<td>SET CURRENT PACKAGESET SET CURRENT PACKAGE PATH SELECT ... FROM T1 ...</td>
<td>The collections in PACKAGE PATH determine which package is invoked.</td>
</tr>
</tbody>
</table>
Table 142. Scope of CURRENT PACKAGE PATH (continued)

<table>
<thead>
<tr>
<th>Example</th>
<th>What happens</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Example</strong></td>
<td><strong>What happens</strong></td>
</tr>
<tr>
<td>SET CURRENT PACKAGE PATH CONNECT TO S2 ... SELECT ... FROM T1 ...</td>
<td>PACKAGE PATH at server S2 is an empty string because it has not been explicitly set. The values from the PKLIST bind option of the plan that is at the requester determine which package is invoked.¹</td>
</tr>
<tr>
<td>SET CURRENT PACKAGE PATH = 'A,B' CONNECT TO S2 ... SET CURRENT PACKAGE PATH = 'X,Y' SELECT ... FROM T1 ...</td>
<td>The collections in PACKAGE PATH that are set at server S2 determine which package is invoked.</td>
</tr>
<tr>
<td>SET CURRENT PACKAGE PATH SELECT ... FROM S2.QUAL.T1 ...</td>
<td>Three-part table name. On implicit connection to server S2, PACKAGE PATH at server S2 is inherited from the local server. The collections in PACKAGE PATH at server S2 determine which package is invoked.</td>
</tr>
</tbody>
</table>

**Notes:**

1. When CURRENT PACKAGE PATH is set at the requester (and not at the remote server), Db2 passes one collection at a time from the list of collections to the remote server until a package is found or until the end of the list. Each time a package is not found at the server, Db2 returns an error to the requester. The requester then sends the next collection in the list to the remote server.

---

**Bind process for remote access**

You can use several different bind processes to enable access to data at a remote server.

These processes work when the remote server is a Db2 for z/OS system or another type of database system that uses DRDA access.

**Bind a package at the local site and the remote site**

If you have not yet bound a local package, use this technique.

1. Bind the DBRM into a package at the local site.
2. Bind the DBRM into a package at the remote site.
3. Bind a plan with a package list that includes the local package and the remote package.

**Example: Bind a package at the local site and the remote site**

Suppose that you precompiled program MYPROG to generate DBRM MYPROG, and compiled and link-edited program MYPROG. You want to run MYPROG to access tables at site CHICAGO from your local site. Use commands like these to bind local and remote packages and a plan. The number at the end of each line corresponds to a previously described step.

```
BIND PACKAGE(LOCALCOLLID) MEMBER(MYPROG) other bind options 1
BIND PACKAGE (CHICAGO.REMOTECOLLID) MEMBER(MYPROG) other bind options 2
BIND PLAN (MYPLAN) PKLIST(LOCALCOLLID.* ,*.REMOTECOLLID.*) 3
```

**Bind a copy of an existing local package at the remote site**

If you have already bound a local package, you can use this technique.

1. Issue the BIND command with COPY and OPTIONS to create a copy of the local package at the remote site.

   OPTIONS controls the option values for bind options that you do not specify.
2. Bind a plan with a package list that includes the local package and the remote package.

Example: Bind a copy of an existing local package at the remote site

Suppose that you previously prepared program MYPROG for execution at the local site. As part of program preparation, you bound package LOCALCOLLID.MYPROG at the local site. Now you want to run MYPROG to access tables at site CHICAGO from your local site. Use commands like these to bind a copy of the local package at site CHICAGO, and then bind a plan. The number at the end of each line corresponds to a previously described step.

```
BIND PACKAGE(CHICAGO.REMOTECOLLID) COPY(LOCALCOLLID.MYPROG) -
   OPTIONS(COMPOSITE|COMMAND) -
   other bind options
BIND PLAN (MYPLAN) PKLIST(LOCALCOLLID.*,*.REMOTECOLLID.*)
```

Bind options for remote access

Binding a package to run at a remote location is like binding a package to run at your local Db2 subsystem. Binding a plan to run the package is like binding any other plan. However, a few differences exist.

For the general instructions, see Chapter 9, “Preparing an application to run on Db2 for z/OS,” on page 835.

BIND PLAN options for DRDA access

The following options of BIND PLAN are particularly relevant to binding a plan that uses DRDA access:

**DISCONNECT**

For most flexibility, use DISCONNECT(EXPLICIT), explicitly or by default. That requires you to use RELEASE statements in your program to explicitly end connections.

The other values of the option are also useful.

- **DISCONNECT(AUTOMATIC)** ends all remote connections during a commit operation, without the need for RELEASE statements in your program.
- **DISCONNECT(CONDITIONAL)** ends remote connections during a commit operation except when an open cursor defined as WITH HOLD is associated with the connection.

**SQLRULES**

Use SQLRULES(Db2), explicitly or by default.

SQLRULES(STD) applies the rules of the SQL standard to your CONNECT statements, so that CONNECT TO x is an error if you are already connected to x. Use STD only if you want that statement to return an error code.

If your program selects LOB data from a remote location, and you bind the plan for the program with SQLRULES(Db2), the format in which you retrieve the LOB data with a cursor is restricted. After you open the cursor to retrieve the LOB data, you must retrieve all of the data using a LOB variable, or retrieve all of the data using a LOB locator variable. If the value of SQLRULES is STD, this restriction does not exist.

If you intend to switch between LOB variables and LOB locators to retrieve data from a cursor, execute the SET SQLRULES=STD statement before you connect to the remote location.

**CURRENTDATA**

Use CURRENTDATA(NO) to force block fetch for ambiguous cursors.

**ENCODING**

Use this option to control the encoding scheme that is used for static SQL statements in the plan and to set the initial value of the CURRENT APPLICATION ENCODING SCHEME special register.

For applications that execute remotely and use explicit CONNECT statements, Db2 uses the ENCODING value for the plan. For applications that execute remotely and use implicit CONNECT
Binder uses the ENCODING value for the package that is at the site where a statement executes.

**BIND PACKAGE options for DRDA access**

The following options of BIND PACKAGE are relevant to binding a package to be run using DRDA access:

**location-name**

Name the location of the server at which the package runs.

The privileges needed to run the package must be granted to the owner of the package at the server. If you are not the owner, you must also have SYSCTRL authority or the BINDAGENT privilege that is granted locally.

**SQLERROR**

Use `SQLERROR(CONTINUE)` if you used SQL(ALL) when precompiling. That creates a package even if the bind process finds SQL errors, such as statements that are valid on the remote server but that the precompiler did not recognize. Otherwise, use `SQLERROR(NOPACKAGE)`, explicitly or by default.

**COPY**

If you bind with the COPY option to copy a local package to a remote site, Db2 performs authorization checking, reads and updates the catalog, and creates the package at the remote site. Db2 reads the catalog records that are related to the copied package at the local site. Db2 converts values that are returned from the remote site in ISO format if all of the following conditions are true:

- If the local site is installed with time or date format LOCAL
- A package is created at a remote site with the COPY option
- The SQL statement does not specify a different format.

**CURRENTDATA**

Use `CURRENTDATA(NO)` to force block fetch for ambiguous cursors.

**OPTIONS**

When you make a remote copy of a package using BIND PACKAGE with the COPY option, use this option to control the default bind options that Db2 uses. Specify:

- `COMPOSITE` to cause Db2 to use any options you specify in the BIND PACKAGE command. For all other options, Db2 uses the options of the copied package. COMPOSITE is the default.
- `COMMAND` to cause Db2 to use the options you specify in the BIND PACKAGE command. For all other options, Db2 uses the defaults for the server on which the package is bound. This helps ensure that the server supports the options with which the package is bound.

**ENCODING**

Use this option to control the encoding scheme that is used for static SQL statements in the package and to set the initial value of the CURRENT APPLICATION ENCODING SCHEME special register.

When you bind the same package locally and remotely, and you specify the ENCODING bind option for the package, the ENCODING bind option for the local package applies to the remote application. The default ENCODING value for a package that is bound at a remote Db2 for z/OS server is the system default for that server. The system default is specified at installation time in the APPLICATION ENCODING field of panel DSNTIPF, which is the APPENSCH DECP value.

**EXPLAIN**

If you specify the option EXPLAIN(YES) or EXPLAIN(ONLY), and you do not specify the option SQLERROR(CONTINUE), PLAN_TABLE must exist at the remote location at which the package is bound.

Related concepts

- Bind options for locks (Db2 Performance)

Related tasks

- BIND options for distributed applications (Db2 Performance)
Considerations for binding packages at a remote location

When you bind packages at a remote location, you need to understand how the behavior of the remote packages differs from the behavior of local packages.

Scope of a remote bind

When you bind or rebind a package at a remote site, Db2 checks authorizations, reads and updates the catalog, and creates the package in the directory at the remote site. Db2 does not read or update catalogs or check authorizations at the local site.

After you bind a remote package, you can bind, rebind, or free the remote package from the local site or at the remote site.

Authorization for binding and running packages at a remote location

To bind a package at a remote Db2 system, you must have all the privileges or authority there that you would need to bind the package on your local system. To bind a package at another type of a system, such as Db2 for Linux, UNIX, and Windows, you need all privileges that the other system requires to execute the SQL statements in the package, and to access the data objects to which the package refers.

Communications database requirements for binding packages at a remote location

When you bind a package at a remote site, the local communications database must be able to resolve the location name in the package to a remote location.

Remote access through a stored procedure

If a local stored procedure uses a cursor to access data, and the cursor-related statement is bound in a separate package under the stored procedure, you must bind this separate package both locally and remotely. In addition, the invoker or owner of the stored procedure must be authorized to execute both local and remote packages. At your local requesting system, you must bind a plan whose package list includes all of those local and remote packages.

Checking which BIND PACKAGE options a particular server supports

You can request only the options of the BIND PACKAGE command that are supported by the server by specifying those options at the requester.

About this task

To find out which options are supported by a specific server DBMS, refer to the documentation provided for that server.

For specific Db2 bind information, refer to the following documentation:

- For guidance in using Db2 bind options and performing a bind process, see Chapter 9, “Preparing an application to run on Db2 for z/OS,” on page 835.
- For the syntax of Db2 BIND command, see the topics BIND PACKAGE (DSN) (Db2 Commands) and BIND PLAN (DSN) (Db2 Commands).
- For the syntax of Db2 REBIND command, see the topics REBIND PACKAGE (DSN) (Db2 Commands) and REBIND PLAN (DSN) (Db2 Commands).
Binding a batch program

Before a batch program can issue SQL statements, a Db2 plan must exist.

About this task

The owner of the plan or package must have all the privileges that are required to execute the SQL statements embedded in it.

You can specify the plan name to Db2 in one of the following ways:

- In the DDITV02 input data set.
- In subsystem member specification.
- By default; the plan name is then the application load module name that is specified in DDITV02.

Db2 passes the plan name to the IMS attach package. If you do not specify a plan name in DDITV02, and a resource translation table (RTT) does not exist or the name is not in the RTT, Db2 uses the passed name as the plan name. If the name exists in the RTT, the name translates to the plan that is specified for the RTT.

Recommendation: Give the Db2 plan the same name as that of the application load module, which is the IMS attachment facility default. The plan name must be the same as the program name.

Conversion of DBRMs that are bound to a plan to DBRMs that are bound to a package

You must bind all DBRMs into a package, and bind the packages into a plan. One package can have only one DBRM.

The default REBIND PLAN COLLID (*) option converts all plans with DBRMs into plans with a package list. You can use this technique for local applications only. If the plan that you specify already contains both DBRMs and package lists, the newly converted package entries will be inserted into the front of the existing package list.

Important: If the same DBRM is in multiple plans, and you run REBIND PLAN with the same COLLID option value on more than one of those plans, Db2 overlays the previously created package in the collection each time you run REBIND with COLLID. To avoid overlaying packages, specify a different COLLID value for each plan that contains DBRMs that are also in other plans.

For more information on developing a strategy for converting your plans to include only packages, see DB2 9 for z/OS: Packages Revisited (IBM Redbooks).

Example: converting all plans

The following examples converts all DBRMs that are bound with plan X into packages under collection ID: DSN_DEFAULT_COLLID_X.

```
REBIND PLAN(X) COLLID(*);
```

Example: specifying a collection ID

The following examples converts DBRMs that are bound with plan X into packages under the my_collection collection ID.

```
REBIND PLAN(x) COLLID('my_collection');
```
Example: rebinding multiple plans which may contain DBRMs
In the following example, BIND will traverse through each plan that is specified in the REBIND PLAN command statement and will convert the DBRMs accordingly, and until none of the DBRMs are bound with plans.

REBIND PLAN (X1, X2, X3) COLLID (collection_id|*);

Example: rebinding all plans which may contain DBRMs
In the following example, BIND will traverse through all plans that are specified in the SYSPLAN table and will convert the DBRMs accordingly, and until none of the DBRMs are bound with plans.

REBIND PLAN (*) COLLID (collection_id|*);

Example: specifying a package list
The following examples converts all DBRMs that are bound with plan X into packages under collection ID: DSN_DEFAULT_COLLID_X.

• If plan X does not have a package list, the newly converted package entries will be appended to the front of package list Z and then package list Z will be added to plan X.
• If plan X has both a package list and DBRMs, the newly converted package entries will be appended to the front of package list Z and then package list Z will replace the existing package list.
• If plan X has only a package list, then package list Z will replace the existing package list.

REBIND PLAN (x) COLLID (collection_id|*) PKLIST(Z);

Example: specifying no package list
The following examples converts all DBRMs that are bound with plan X into packages under collection ID: DSN_DEFAULT_COLLID_X.

• If plan X has both a package list and DBRMs, the existing package list will be deleted, and the new package list will be bound into plan X.
• If plan X has only DBRMs, the DBRMs will be converted into packages accordingly and added to plan X. The NOPKLIST option will be ignored.
• If plan X does not have DBRMs, then the existing package list, if any, will be deleted.

REBIND PLAN (x) COLLID (collection_id|*) NOPKLIST;

Converting an existing plan into packages to run remotely
If you have an existing application that you want to run at a remote location by using remote access, you need a new plan that includes those remote packages in its package list.

Procedure
To turn an existing plan with member DBRMs into packages to run remotely, perform the following actions for each remote location:
1. Choose a name for a collection to contain member DBRMs, such as REMOTE1.
2. Convert the plan into a plan with a package list of packages.

REBIND PLAN(REMOTE1)COLLID(*)

Specifying COLLID(*) produces the packages under the collection of DSN_DEFAULT_COLLID_planname.
3. Query SYSIBM.SYSPACKDEP, to see if any of the packages have a dependency on an alias. That alias is a definition for a 3-part name.

   a) For each of the packages that have a dependency on an alias:

   ```sql
   BIND PACKAGE(location.remote_server_collid)
   COPY(DSN_DEFAULT_COLLID_planname.pgkid)
   COPYVER (...) OPTIONS(COMPOSITE)
   ```

4. Adjust the location's package list. If prior to this process, the plan had no package list, after “2” on page 878 it will have a package list containing DSN_DEFAULT_COLLID_planname.pgkid.

   ```sql
   REBIND PLAN PKLIST
   (*:DSN_DEFAULT_COLLID_planname.pgkid*:remote_server_collid.* )
   ```

**Results**

When you now run the existing application at your local Db2 system using the new application plan, these things happen:

- You connect immediately to the remote location that is named in the CURRENTSERVER option.
- Db2 searches for the package in the collection REMOTE1 at the remote location.
- Any UPDATE, DELETE, or INSERT statements in your application affect tables at the remote location.
- Any results from SELECT statements are returned to your existing application program, which processes them as though they came from your local Db2 system.

**Setting the program level**

The program level defines the level for a particular module. This information is stored in the consistency token, which is in an internal Db2 format. Overriding the program level in the consistency token is possible, if needed, but generally not recommended.

**Procedure**

Use the LEVEL (aaaa) option.

Db2 uses the value that you choose for aaaa to generate the consistency token. Although this method is not recommended for general use and the DSNH CLIST or the Db2 Program Preparation panels do not support it, this method enables you to perform the following actions:

a. Change the source code (but not the SQL statements) in the Db2 precompiler output of a bound program.

b. Compile and link-edit the changed program.

c. Run the application without rebinding a plan or package.

**Dynamic rules options for dynamic SQL statements**

The DYNAMICRULES bind option and the runtime environment determine the rules for the dynamic SQL attributes.

The BIND or REBIND option DYNAMICRULES determines what values apply at run time for the following dynamic SQL attributes:

- The authorization ID that is used to check authorization
- The qualifier that is used for unqualified objects
- The source for application programming options that Db2 uses to parse and semantically verify dynamic SQL statements
- Whether dynamic SQL statements can include GRANT, REVOKE, ALTER, CREATE, DROP, and RENAME statements
In addition, the runtime environment of a package controls how dynamic SQL statements behave at run time. The two possible runtime environments are:

- The package runs as part of a stand-alone program.
- The package runs as a stored procedure or user-defined function package, or it runs under a stored procedure or user-defined function.

A package that runs under a stored procedure or user-defined function is a package whose associated program meets one of the following conditions:

- The program is called by a stored procedure or user-defined function.
- The program is in a series of nested calls that start with a stored procedure or user-defined function.

### Dynamic SQL statement behavior

The dynamic SQL attributes that are determined by the value of the DYNAMICRULES bind option and the runtime environment are collectively called the dynamic SQL statement behavior or the dynamic rules behavior. The four dynamic rules behaviors are: run, bind, define, and invoke.

The following table shows the combination of DYNAMICRULES value and runtime environment that yield each dynamic SQL behavior.

#### Table 143. How DYNAMICRULES and the runtime environment determine dynamic SQL statement behavior

<table>
<thead>
<tr>
<th>DYNAMICRULES value</th>
<th>Stand-alone program environment</th>
<th>User-defined function or stored procedure environment</th>
</tr>
</thead>
<tbody>
<tr>
<td>RUN</td>
<td>Run</td>
<td>Run</td>
</tr>
<tr>
<td>BIND</td>
<td>Bind</td>
<td>Bind</td>
</tr>
<tr>
<td>DEFINERUN</td>
<td>Run</td>
<td>Define</td>
</tr>
<tr>
<td>DEFINEBIND</td>
<td>Bind</td>
<td>Define</td>
</tr>
<tr>
<td>INVOKERUN</td>
<td>Run</td>
<td>Invoke</td>
</tr>
<tr>
<td>INVOKEBIND</td>
<td>Bind</td>
<td>Invoke</td>
</tr>
</tbody>
</table>

**Note:** BIND and RUN values can be specified for packages, plans, and native SQL procedures. The other values can be specified for packages and native SQL procedures but not for plans.

The following table shows the dynamic SQL attribute values for each type of dynamic SQL behavior.

#### Table 144. Definitions of dynamic SQL statement behaviors

<table>
<thead>
<tr>
<th>Dynamic SQL attribute</th>
<th>Setting for dynamic SQL behavior attributes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Authorization ID</td>
<td>Bind owner, Current SQLID, User-defined function or stored procedure owner, Authorization ID of invoker²</td>
</tr>
<tr>
<td>Default qualifier for unqualified objects</td>
<td>Bind OWNER or QUALIFIER value, CURRENT SCHEMA, User-defined function or stored procedure owner, Authorization ID of invoker</td>
</tr>
<tr>
<td>CURRENT SQLID²</td>
<td>Not applicable, Applies, Not applicable, Not applicable</td>
</tr>
</tbody>
</table>
Table 144. Definitions of dynamic SQL statement behaviors (continued)

<table>
<thead>
<tr>
<th>Dynamic SQL attribute</th>
<th>Bind</th>
<th>Run</th>
<th>Define</th>
<th>Invoke</th>
</tr>
</thead>
<tbody>
<tr>
<td>Source for application programming options</td>
<td>Determined by DSNHDECP or a user-specified application defaults module parameter DYNRULS³</td>
<td>Install panel DSNTIP4</td>
<td>Determined by DSNHDECP or a user-specified application defaults module parameter DYNRULS³</td>
<td>Determined by DSNHDECP or a user-specified application defaults module parameter DYNRULS³</td>
</tr>
<tr>
<td>Can execute GRANT, REVOKE, CREATE, ALTER, DROP, RENAME?</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

Notes:

1. If the invoker is the primary authorization ID of the process or the CURRENT SQLID value, secondary authorization IDs are also checked if they are needed for the required authorization. Otherwise, only one ID, the ID of the invoker, is checked for the required authorization.

2. Db2 uses the value of CURRENT SQLID as the authorization ID for dynamic SQL statements only for plans and packages that have run behavior. For the other dynamic SQL behaviors, Db2 uses the authorization ID that is associated with each dynamic SQL behavior, as shown in this table.

   The value to which CURRENT SQLID is initialized is independent of the dynamic SQL behavior. For standalone programs, CURRENT SQLID is initialized to the primary authorization ID.

   You can execute the SET CURRENT SQLID statement to change the value of CURRENT SQLID for packages with any dynamic SQL behavior, but Db2 uses the CURRENT SQLID value only for plans and packages with run behavior.

3. The value of DSNHDECP or a user-specified application defaults module parameter DYNRULS, which you specify in field USE FOR DYNAMICRULES in installation panel DSNTIP4, determines whether Db2 uses the SQL statement processing options or the application programming defaults for dynamic SQL statements. See “Options for SQL statement processing” on page 851 for more information.

Related concepts
Authorization IDs and dynamic SQL (Db2 SQL)
Authorization behaviors for dynamic SQL statements (Managing Security)

Related reference
DYNAMICRULES bind option (Db2 Commands)

Dynamic plan selection

It is beneficial to use dynamic plan selection and packages together. You can convert individual programs in an application that contains many programs and plans, one at a time, to use a combination of plans and packages. This process reduces the number of plans per application; having fewer plans reduces the effort that is needed to maintain the dynamic plan exit routine.

CICS
You can use packages and dynamic plan selection together, but when you dynamically switch plans, the following conditions must exist:

- All special registers, including CURRENT PACKAGESET, must contain their initial values.
- The value in the CURRENT DEGREE special register cannot have changed during the current transaction.

Assume that you develop the following programs and DBRMs:
Table 145. Example programs and DBRMs

<table>
<thead>
<tr>
<th>Program Name</th>
<th>DBRM Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>MAIN</td>
<td>MAIN</td>
</tr>
<tr>
<td>PROGA</td>
<td>PLANA</td>
</tr>
<tr>
<td>PROGB</td>
<td>PKGB</td>
</tr>
<tr>
<td>PROGC</td>
<td>PLANC</td>
</tr>
</tbody>
</table>

You could create packages using the following bind statement:

```
BIND PACKAGE(PKGB) MEMBER(PKGB)
```

The following scenario illustrates thread association for a task that runs program MAIN. Suppose that you execute the following SQL statements in the indicated order. For each SQL statement, the resulting event is described.

1. EXEC CICS START TRANSID(MAIN)
   TRANSID(MAIN) executes program MAIN.
2. EXEC SQL SELECT...
   Program MAIN issues an SQL SELECT statement. The default dynamic plan exit routine selects plan MAIN.
3. EXEC CICS LINK PROGRAM(PROGA)
   Program PROGA is invoked.
4. EXEC SQL SELECT...
   Db2 does not call the default dynamic plan exit routine, because the program does not issue a sync point. The plan is MAIN.
5. EXEC CICS LINK PROGRAM(PROGB)
   Program PROGB is invoked.
6. EXEC SQL SELECT...
   Db2 does not call the default dynamic plan exit routine, because the program does not issue a sync point. The plan is MAIN and the program uses package PKGB.
7. EXEC CICS SYNCPOINT
   Db2 calls the dynamic plan exit routine when the next SQL statement executes.
8. EXEC CICS LINK PROGRAM(PROGC)
   Program PROGC is invoked.
9. EXEC SQL SELECT...
   Db2 calls the default dynamic plan exit routine and selects PLANC.
10. EXEC SQL SET CURRENT SQLID = 'ABC'
    The CURRENT SQLID special register is assigned the value 'ABC.'
11. EXEC CICS SYNCPOINT
    Db2 does not call the dynamic plan exit routine when the next SQL statement executes because the previous statement modifies the special register CURRENT SQLID.
12. EXEC CICS RETURN
    Control returns to program PROGB.
13. EXEC SQL SELECT...
CICS  With packages, you probably do not need dynamic plan selection and its accompanying exit routine. A package that is listed within a plan is not accessed until it is executed. However, you can use dynamic plan selection and packages together, which can reduce the number of plans in an application and the effort to maintain the dynamic plan exit routine.

Rebinding applications

You must rebind applications to change bind options. You also need to rebind applications when you make changes that affect the plan or package, such as creating an index, but you have not changed the SQL statements.

About this task

In some cases, Db2 automatically rebinds plans or packages for you, depending on the value of the ABIND subsystem parameter. For details, see “Automatic rebinds” on page 891.

The following actions might require that you rebind a package:

- Changing the host language or SQL statements in the application. You must replace the package. Precompile, compile, and link the application program. Then issue a BIND command with the ACTION(REPLACE) option.
- Changing your data attributes in ways that invalidate the package. For details, see “Changes that invalidate packages” on page 14.
- Improving access paths selection after reorganizing data with the REORG utility or collecting database statistics with RUNSTATS or other utilities. For more information, see Maintaining data organization and statistics (Db2 Performance).
- Enabling Db2 to select an access path that uses a newly created index for access to a table.
- Changing the bind options for a package. If an option that you want to change is not available for the REBIND command, issue the BIND command with ACTION(REPLACE) instead.
- Preparing for migration to a new Db2 release. For more information, see Rebind old plans and packages in Db2 11 to avoid disruptive autobinds in Db2 12 (Db2 Installation and Migration).

Related tasks

Identifying packages with characteristics that affect performance, concurrency, or the ability to run (Db2 Performance)

Related reference

AUTO BIND field (ABIND subsystem parameter) (Db2 Installation and Migration)
REBIND PACKAGE (DSN) (Db2 Commands)
BIND PACKAGE (DSN) (Db2 Commands)
BIND and REBIND options for packages, plans, and services (Db2 Commands)

Rebinding a package

You need to rebind a package when you make changes that affect the package but that do not involve changes to the SQL statements. For example, if you create a new index, you need to rebind the package. If you change the SQL, you need to use the BIND PACKAGE command with the ACTION(REPLACE) option.

Before you begin

For a trigger package, use the REBIND TRIGGER PACKAGE subcommand. For more information, see “Trigger packages” on page 160.

Procedure

Use the REBIND PACKAGE subcommand.

You can change any of bind options for a package when you rebind it.
The following table clarifies which packages are bound, depending on how you specify `collection-id` (coll-id), `package-id` (pkg-id), and `version-id` (ver-id) on the REBIND PACKAGE subcommand.

REBIND PACKAGE does not apply to packages for which you do not have the BIND privilege. An asterisk (*) used as an identifier for collections, packages, or versions does not apply to packages at remote sites.

*Table 146. Behavior of REBIND PACKAGE specification.* "All" means all collections, packages, or versions at the local Db2 server for which the authorization ID that issues the command has the BIND privilege.

<table>
<thead>
<tr>
<th>Input</th>
<th>Collections affected</th>
<th>Packages affected</th>
<th>Versions affected</th>
</tr>
</thead>
<tbody>
<tr>
<td>*</td>
<td>all</td>
<td>all</td>
<td>all</td>
</tr>
<tr>
<td>**.(.*)</td>
<td>all</td>
<td>all</td>
<td>all</td>
</tr>
<tr>
<td><em>.</em></td>
<td>all</td>
<td>all</td>
<td>all</td>
</tr>
<tr>
<td>**.(ver-id)</td>
<td>all</td>
<td>all</td>
<td>ver-id</td>
</tr>
<tr>
<td>**.(.)</td>
<td>all</td>
<td>all</td>
<td>empty string</td>
</tr>
<tr>
<td>coll-id.*</td>
<td>coll-id</td>
<td>all</td>
<td>all</td>
</tr>
<tr>
<td>coll-id.<em>.(.</em>)</td>
<td>coll-id</td>
<td>all</td>
<td>all</td>
</tr>
<tr>
<td>coll-id.*.(ver-id)</td>
<td>coll-id</td>
<td>all</td>
<td>ver-id</td>
</tr>
<tr>
<td>coll-id.*(.)</td>
<td>coll-id</td>
<td>all</td>
<td>empty string</td>
</tr>
<tr>
<td>coll-id.pkg-id.(.*)</td>
<td>coll-id</td>
<td>pkg-id</td>
<td>all</td>
</tr>
<tr>
<td>coll-id.pkg-id</td>
<td>coll-id</td>
<td>pkg-id</td>
<td>empty string</td>
</tr>
<tr>
<td>coll-id.pkg-id.(.)</td>
<td>coll-id</td>
<td>pkg-id</td>
<td>empty string</td>
</tr>
<tr>
<td>coll-id.pkg-id.(ver-id)</td>
<td>coll-id</td>
<td>pkg-id</td>
<td>ver-id</td>
</tr>
<tr>
<td><em>.pkg-id.(.</em>)</td>
<td>all</td>
<td>pkg-id</td>
<td>all</td>
</tr>
<tr>
<td>*.pkg-id</td>
<td>all</td>
<td>pkg-id</td>
<td>empty string</td>
</tr>
<tr>
<td>*.pkg-id.(.)</td>
<td>all</td>
<td>pkg-id</td>
<td>empty string</td>
</tr>
<tr>
<td>*.pkg-id.(ver-id)</td>
<td>all</td>
<td>pkg-id</td>
<td>ver-id</td>
</tr>
</tbody>
</table>
Examples

Example: Rebinding a package at a remote location
The following example shows the options for rebinding a package at the remote location. The location name is SNTERSA. The collection is GROUP1, the package ID is PROGA, and the version ID is V1. The connection types shown in the REBIND subcommand replace connection types that are specified on the original BIND subcommand.

```
REBIND PACKAGE(SNTERSA.GROUP1.PROGA.(V1)) ENABLE(CICS,REMOTE)
```

Example: Rebinding all local packages
You can use the asterisk on the REBIND subcommand for local packages, but not for packages at remote sites. Any of the following commands rebinds all versions of all packages in all collections, at the local Db2 system, for which you have the BIND privilege.

```
REBIND PACKAGE (*)
REBIND PACKAGE (*.*)
REBIND PACKAGE (*.*( *))
```

Example: Rebinding all versions of all local packages
Either of the following commands rebinds all versions of all packages in the local collection LEDGER for which you have the BIND privilege.

```
REBIND PACKAGE (LEDGER.*)
REBIND PACKAGE (LEDGER.*.())
```

Example: Rebinding local packages in all collections
Either of the following commands rebinds the empty string version of the package DEBIT in all collections, at the local Db2 system, for which you have the BIND privilege.

```
REBIND PACKAGE (*.DEBIT)
REBIND PACKAGE (*.DEBIT.())
```

Related tasks
Reusing and comparing access paths at bind and rebind (Db2 Performance)

Related reference
BIND and REBIND options for packages, plans, and services (Db2 Commands)
REBIND PACKAGE (DSN) (Db2 Commands)

Rebinding a plan
You need to rebind a plan when you make a change to one of the attributes of the plan, such as the package list.

Procedure
Use the REBIND PLAN subcommand.
You can change any of bind options for that plan.
When you rebind a plan, use the PKLIST keyword to replace any previously specified package list. Omit the PKLIST keyword to use of the previous package list for rebinding. Use the NOPKLIST keyword to delete any package list that was specified when the plan was previously bound.

Examples

Example
The following command rebinds PLANA and changes the package list:
Example
The following command rebinds the plan and drops the entire package list:

```
REBIND PLAN(PLANA) NOPKLIST
```

Related reference
BIND and REBIND options for packages, plans, and services (Db2 Commands)
REBIND PLAN (DSN) (Db2 Commands)

Rebinding lists of plans and packages
In some situations, you need to rebind a set of plans or packages that cannot be described by using asterisks. For example, if a rebind operation terminates, you can generate a rebind subcommand for each object that was not bound.

About this task
One situation in which this technique is useful is to complete a rebind operation that has terminated due to lack of resources. A rebind for many objects, such as REBIND PACKAGE (*) for an ID with SYSADM authority, terminates if a needed resource becomes unavailable. As a result, some objects are successfully rebound and others are not. If you repeat the subcommand, Db2 attempts to rebind all the objects again. But if you generate a rebind subcommand for each object that was not rebound, and issue those subcommands, Db2 does not repeat any work that was already done and is not likely to run out of resources.

For a description of the technique and several examples of its use, see “Sample program to create REBIND subcommands for lists of plans and packages” on page 886.

Generating lists of REBIND commands
To generate a list of REBIND subcommands for a set of packages that cannot be described, use asterisks, and use information in the Db2 catalog. You can then issue the list of subcommands through DSN.

About this task
The following list is an overview of the procedures for REBIND PACKAGE:
1. Use DSNTIAUL to generate the REBIND PACKAGE subcommands for the selected packages.
2. Use DSNTEDIT CLIST to delete extraneous blanks from the REBIND PACKAGE subcommands.
3. Use TSO edit commands to add DSN commands to the sequential data set.
4. Use DSN to execute the REBIND PACKAGE subcommands for the selected packages.

Sample program to create REBIND subcommands for lists of plans and packages
If you cannot use asterisks to identify a list of packages or plans that you want to rebind, you might be able to create the needed REBIND subcommands automatically, by using the sample program DSNTIAUL.

One situation in which this technique might be useful is when a resource becomes unavailable during a rebind of many plans or packages. Db2 normally terminates the rebind and does not rebind the remaining plans or packages. Later, however, you might want to rebind only the objects that remain to be rebound. You can build REBIND subcommands for the remaining plans or packages by using DSNTIAUL to select the plans or packages from the Db2 catalog and to create the REBIND subcommands. You can then submit the subcommands through the DSN command processor, as usual.

You might first need to edit the output from DSNTIAUL so that DSN can accept it as input. The CLIST DSNTEDIT can perform much of that task for you.
This section contains the following topics:

- “Generating lists of REBIND commands” on page 886
- “Sample SELECT statements for generating REBIND commands” on page 887
- “Sample JCL for running lists of REBIND commands” on page 889

Sample SELECT statements for generating REBIND commands

You can select specific plans or packages to be rebound and concatenate the REBIND subcommand syntax around the plan or package names. You can also convert a varying-length string to a fixed-length string, and append additional blanks to the REBIND PLAN and REBIND PACKAGE subcommands, so that the DSN command processor can accept the record length as valid input.

**Building REBIND subcommands:** The examples that follow illustrate the following techniques:

- Using SELECT to select specific packages or plans to be rebound
- Using the CONCAT operator to concatenate the REBIND subcommand syntax around the plan or package names
- Using the SUBSTR function to convert a varying-length string to a fixed-length string
- Appending additional blanks to the REBIND PLAN and REBIND PACKAGE subcommands, so that the DSN command processor can accept the record length as valid input

**If the SELECT statement returns rows,** then DSNTIAUL generates REBIND subcommands for the plans or packages identified in the returned rows. Put those subcommands in a sequential data set, where you can then edit them.

For REBIND PACKAGE subcommands, delete any extraneous blanks in the package name, using either TSO edit commands or the Db2 CLIST DSNTEDIT.

For both REBIND PLAN and REBIND PACKAGE subcommands, add the DSN command that the statement needs as the first line in the sequential data set, and add END as the last line, using TSO edit commands. When you have edited the sequential data set, you can run it to rebind the selected plans or packages.

**If the SELECT statement returns no qualifying rows,** then DSNTIAUL does not generate REBIND subcommands.

The examples in this topic generate REBIND subcommands that work in Db2 for z/OS Db2 12. You might need to modify the examples for prior releases of Db2 that do not allow all of the same syntax.

**Example: REBIND all plans without terminating because of unavailable resources.**

```sql
SELECT SUBSTR('REBIND PLAN('CONCAT NAME CONCAT')                        ',1,45)
FROM SYSIBM.SYSPLAN;
```

**Example: REBIND all versions of all packages without terminating because of unavailable resources.**

```sql
SELECT SUBSTR('REBIND PACKAGE('CONCAT COLLID CONCAT'.CONCAT NAME CONCAT'.(*))        ',1,55)
FROM SYSIBM.SYSPACKAGE;
```

**Example: REBIND all plans bound before a given date and time.**

```sql
SELECT SUBSTR('REBIND PLAN('CONCAT NAME CONCAT')                        ',1,45)
FROM SYSIBM.SYSPLAN
WHERE BINDDATE <= 'yymmdd' OR
      (BINDDATE <= 'yymmdd' AND
       BINDTIME <= 'hhmmssth');
```

where `yymmdd` represents the date portion and `hhmmssth` represents the time portion of the timestamp string.
If the date specified is after 2000, you need to include another condition that includes plans that were bound before year 2000:

```sql
WHERE
    BINDDATE >= '830101' OR
    BINDDATE <= 'yymmdd' OR
    (BINDDATE <= 'yymmdd' AND
     BINDTIME <= 'hhmmssth');
```

**Example: REBIND all versions of all packages bound before a given date and time.**

```sql
SELECT SUBSTR('REBIND PACKAGE('CONCAT COLLID CONCAT'.
    CONCAT NAME CONCAT'.(*))        ',1,55)
FROM SYSIBM.SYSPACKAGE
WHERE BINDTIME <= 'timestamp';
```

where `timestamp` is an ISO timestamp string.

**Example: REBIND all plans bound since a given date and time.**

```sql
SELECT SUBSTR('REBIND PLAN('CONCAT NAME
    CONCAT')                        ',1,45)
FROM SYSIBM.SYSPLAN
WHERE BINDDATE >= 'yymmdd' AND
    BINDTIME >= 'hhmmssth';
```

where `yymmdd` represents the date portion and `hhmmssth` represents the time portion of the timestamp string.

**Example: REBIND all versions of all packages bound since a given date and time.**

```sql
SELECT SUBSTR('REBIND PACKAGE('CONCAT COLLID CONCAT'.
    CONCAT NAME CONCAT'.(*))        ',1,55)
FROM SYSIBM.SYSPACKAGE
WHERE BINDTIME >= 'timestamp';
```

where `timestamp` is an ISO timestamp string.

**Example: REBIND all plans bound within a given date and time range.**

```sql
SELECT SUBSTR('REBIND PLAN('CONCAT NAME
    CONCAT')                        ',1,45)
FROM SYSIBM.SYSPLAN
WHERE
    (BINDDATE >= 'yymmdd' AND
     BINDTIME >= 'hhmmssth') AND
    BINDDATE <= 'yymmdd' AND
    BINDTIME <= 'hhmmssth';
```

where `yymmdd` represents the date portion and `hhmmssth` represents the time portion of the timestamp string.

**Example: REBIND all versions of all packages bound within a given date and time range.**

```sql
SELECT SUBSTR('REBIND PACKAGE('CONCAT COLLID CONCAT'.
    CONCAT NAME CONCAT'.(*))        ',1,55)
FROM SYSIBM.SYSPACKAGE
WHERE BINDTIME >= 'timestamp1' AND
    BINDTIME <= 'timestamp2';
```

where `timestamp1` and `timestamp2` are ISO timestamp strings.

**Example: REBIND all invalid versions of all packages.**

```sql
SELECT SUBSTR('REBIND PACKAGE('CONCAT COLLID CONCAT'.
    CONCAT NAME CONCAT'.(*))        ',1,55)
FROM SYSIBM.SYSPACKAGE
WHERE VALID = 'N';
```
Example: REBIND all plans bound with ISOLATION level of cursor stability.

```
SELECT SUBSTR('REBIND PLAN('CONCAT NAME CONCAT')',1,45)
FROM SYSIBM.SYSPLAN
WHERE ISOLATION = 'S';
```

Example: REBIND all versions of all packages that allow CPU and/or I/O parallelism.

```
SELECT SUBSTR('REBIND PACKAGE('CONCAT COLLID CONCAT'.CONCAT NAME CONCAT'.(*))',1,55)
FROM SYSIBM.SYSPACKAGE
WHERE DEGREE='ANY';
```

Sample JCL for running lists of REBIND commands

You can use JCL to rebind all versions of all packages that are bound within a specified date and time period.

You specify the date and time period for which you want packages to be rebound in a WHERE clause of the SELECT statement that contains the REBIND command. In the following example, the WHERE clause looks like the following clause:

```
WHERE BINDTIME >= 'YYYY-MM-DD-hh.mm.ss' AND
      BINDTIME <= 'YYYY-MM-DD-hh.mm.ss'
```

The date and time period has the following format:

**YYYY**
The four-digit year. For example: 2008.

**MM**
The two-digit month, which can be a value between 01 and 12.

**DD**
The two-digit day, which can be a value between 01 and 31.

**hh**
The two-digit hour, which can be a value between 01 and 24.

**mm**
The two-digit minute, which can be a value between 00 and 59.

**ss**
The two-digit second, which can be a value between 00 and 59.

```
//REBINDS JOB MSGLEVEL=(1,1),CLASS=A,MSGCLASS=A,USER=SYSADM,
//             REGION=1024K
//*********************************************************************/
//SETUP EXEC PGM=IKJEFT01
//SYSTSPRT DD SYSOUT=*
//SYSTSIN DD *
DSN SYSTEM(DSN)
RUN PROGRAM(DSNTIAUL) PLAN(DSNTIBC1) PARMS('SQL') -
   LIB('DSN1210.RUNLIB.LOAD')
END
//SYSPRINT DD SYSOUT=*  
//SYSPUNCH DD SYSOUT=*  
//SYSPUNCH DD SYSOUT=*  
//SYSPRINT DD SYSOUT=*  
DSN=SYSADM.SYSTSIN.DATA,UNIT=SYSDA,DISP=SHR
//*********************************************************************/
//                                                                 
//ÆGENER= '<SUBCOMMANDS TO REBIND ALL PACKAGES BOUND IN YYYY
//                                                                 
//Æ*********************************************************************/
//SYSIN DD *
SELECT SUBSTR('REBIND PACKAGE('CONCAT COLLID CONCAT'.CONCAT NAME CONCAT'.(*))',1,55)
FROM SYSIBM.SYSPACKAGE
WHERE BINDTIME >= 'YYYY-MM-DD-hh.mm.ss' AND
BINDTIME <= 'YYYY-MM-DD-hh.mm.ss';

/*
//*****************************************************************************
//** STRIP THE BLANKS OUT OF THE REBIND SUBCOMMANDS
//**
//*****************************************************************************/
//STRIP EXEC PGM=IKJEFT01
//SYSPROC DD DSN=SYSADM.DSNCLIST,DISP=SHR
//SYSTSPPRT DD SYSOUT=*;
//SYSPRINT DD SYSOUT=*;
//SYSTSPRT DD SYSOUT=*;
//SYSSIN DD *;
//DSNEDIT SYSADM.SYSTSIN.DATA
//SYSIN DD DUMMY
/*!*/

//*****************************************************************************/
//** PUT IN THE DSN COMMAND STATEMENTS
//*****************************************************************************/
//EDIT EXEC PGM=IKJEFT01
//SYSTSPRT DD SYSOUT=*;
//SYSSIN DD *
EDIT 'SYSADM.SYSTSIN.DATA' DATA NONUM
TOP
INSERT DSN SYSTEM(DSN)
BOTTOM
INSERT END
TOP
LIST * 99999
END SAVE
/*!*/

//*****************************************************************************/
//** EXECUTE THE REBIND PACKAGE SUBCOMMANDS THROUGH DSN
//*****************************************************************************/
//LOCAL EXEC PGM=IKJEFT01
//DBRMLIB DD DSN=DSN1210.BRMLIB.DA,
//DISP=SHR
//SYSTSPPRT DD SYSOUT=*;
//SYSPRINT DD SYSOUT=*;
//SYSSIN DD *
//SNDUTMP DD SYSOUT=*;
//SYSTSPRT DD SYSOUT=*;
//SYSSIN DD DSN=SYSADM.SYSTSIN.DATA,
//UNIT=SYSDA,DISP=SHR
/*!*/

The following example shows some sample JCL for rebinding all plans bound without specifying the DEGREE keyword on BIND with DEGREE(ANY).

//REBINDS JOB MSGLEVEL=(1,1),CLASS=A,MSGCLASS=A,USER=SYSADM,
//REGION=1024K
/*****************************************************************************/
//SETUP EXEC TSObATCH
//SYSTSPRT DD SYSOUT=*;
//SYSPUNCH DD SYSOUT=*;
//SYSSREC00 DD DSN=SYSADM.SYSTSIN.DATA,
//UNIT=SYSDA,DISP=SHR
/*****************************************************************************/
//** REBIND ALL PLANS THAT WERE BOUND WITHOUT SPECIFYING THE DEGREE
//** KEYWORD ON BIND WITH DEGREE(ANY)
//*/
/*****************************************************************************/
//SYSTSPRT DD *
//DSN (DSN)
RUN PROGRAM(DSNTIAUL) PLAN(DSNTIBC1) PARM('SQL')
END
//SYSSIN DD *
SELECT SUBSTR('REBIND PLAN('CONCAT NAME
CONCAT') DEGREE(ANY)            ',1,45)
FROM SYSIBM.SYSPRAN
WHERE DEGREE=' ';
/*! */
/*****************************************************************************/
//** PUT IN THE DSN COMMAND STATEMENTS
/*****************************************************************************/

890  Db2 12 for z/OS: Application Programming and SQL Guide
Automatic rebinds

_Automatic rebinds_ (sometimes called "autobinds") occur when an authorized user runs a package or plan and the runtime structures in the plan or package cannot be used. This situation usually results from changes to the attributes of the data on which the package or plan depends, or changes to the environment in which the package or plan runs.

In most cases, Db2 marks a package that must be automatically rebound as _invalid_ by setting VALID='N' in the SYSIBM.SYSPLAN and SYSIBM.SYSPACKAGE catalog tables. For a list of actions that results in Db2 marking a package invalid, see "Changes that invalidate packages" on page 14.

If an automatic rebind fails, Db2 marks a package as _inoperative_ in the OPERATIVE column of SYSIBM.SYSPLAN and SYSIBM.SYSPACKAGE catalog tables.

Controls for automatic binds

Db2 uses automatic binds only when the ABIND subsystem parameter is set to YES or COEXIST. If ABIND is set to NO when an invalid package runs, Db2 returns an error. For details, see AUTO BIND field (ABIND subsystem parameter) (Db2 Installation and Migration).

You can also use resource limit tables to control automatic binds. For details, see Restricting bind operations (Db2 Performance).

Bind options for automatic binds

In general, Db2 uses the same bind options from the most recent bind process for automatic binds. The exceptions are:

- If an option is no longer supported, the automatic rebind option process substitutes a supported option.
- If an option does not have an existing value, the default bind option is used.
- The automatic rebind value for APCOMPARE is NONE.
- The automatic rebind value for APREUSE is WARN, and the automatic rebind value for APREUSESOURCE is CURRENT.
- If there is no existing value for the APPLCOMPAT bind option, the APPLCOMPAT subsystem parameter is used.
Automatic binds with package copies

If a package has previous or original copies as a result of rebinding with the PLANMGMT(BASIC) or PLANMGMT(EXTENDED) options or having the PLANMGMT subsystem parameter set to BASIC or EXTENDED, those copies are not affected by automatic rebind. Automatic rebind replaces only the current copy.

A situation can occur in which automatic rebind causes the previous or original copy to be at a newer Db2 version than the current copy. Suppose that copy A is the current copy, and copy B is the previous copy. Copy A is at a previous and supported version for Db2 packages, but copy B is at an older Db2 version than the minimum supported version. When you switch the packages so that copy B becomes the current copy, and run copy B, Db2 automatically rebinds copy B. Now, copy B is at a newer Db2 version than copy A.

When automatic binds fail

When an automatic bind fails, Db2 issues message DSNT500I to the console with reason ‘00E30305’x, resource type ‘804’x, and resource name collection.package.(version).

If EXPLAIN(YES) was specified for the previous rebind operation, the ABEXP subsystem parameter controls whether Db2 captures EXPLAIN information during automatic rebinds. For details, see EXPLAIN PROCESSING field (ABEXP subsystem parameter) (Db2 Installation and Migration). Automatic rebinds fail for most EXPLAIN errors.

If an automatic bind occurs while running in ACCESS(MAINT) mode the automatic bind is run under the authorization id of SYSOPR. If SYSOPR is not defined as an installation SYSOPR the automatic bind fails.

Related concepts

Automatic binds in coexistence (Db2 Installation and Migration)
Application and SQL release incompatibilities (Db2 for z/OS What's New?)

Related tasks

Rebind old plans and packages in Db2 11 to avoid disruptive autobinds in Db2 12 (Db2 Installation and Migration)

Related reference

BIND PACKAGE (DSN) (Db2 Commands)
BIND PLAN (DSN) (Db2 Commands)
BIND and REBIND options for packages, plans, and services (Db2 Commands)

Related information

DSNT500I (Db2 Messages)
00E30305 (Db2 Codes)

Specifying the rules that apply to SQL behavior at run time

You can specify whether Db2 rules or SQL standard rules apply to SQL behavior at run time.

About this task

Not only does SQLRULES specify the rules under which a type 2 CONNECT statement executes, but it also sets the initial value of the special register CURRENT RULES when the database server is the local Db2 system. When the server is not the local Db2 system, the initial value of CURRENT RULES is DB2. After binding a plan, you can change the value in CURRENT RULES in an application program by using the statement SET CURRENT RULES.

CURRENT RULES determines the SQL rules, Db2 or SQL standard, that apply to SQL behavior at run time. For example, the value in CURRENT RULES affects the behavior of defining check constraints by issuing the ALTER TABLE statement on a populated table:
• If CURRENT RULES has a value of STD and no existing rows in the table violate the check constraint, Db2 adds the constraint to the table definition. Otherwise, an error occurs and Db2 does not add the check constraint to the table definition.

If the table contains data and is already in a check pending status, the ALTER TABLE statement fails.

• If CURRENT RULES has a value of DB2, Db2 adds the constraint to the table definition, defers the enforcing of the check constraints, and places the table space or partition in CHECK-pending status.

You can use the statement SET CURRENT RULES to control the action that the statement ALTER TABLE takes. Assuming that the value of CURRENT RULES is initially STD, the following SQL statements change the SQL rules to DB2, add a check constraint, defer validation of that constraint, place the table in CHECK-pending status, and restore the rules to STD.

```
EXEC SQL
SET CURRENT RULES = 'DB2';
EXEC SQL
ALTER TABLE DSN8C10.EMP
ADD CONSTRAINT C1 CHECK (BONUS <= 1000.0);
EXEC SQL
SET CURRENT RULES = 'STD';
```

See “Check constraints” on page 127 for information about check constraints.

You can also use CURRENT RULES in host variable assignments. For example, if you want to store the value of the CURRENT RULES special register at a particular point in time, you can use assign the value to a host variable, as in the following statement:

```
SET :XRULE = CURRENT RULES;
```

You can also use CURRENT RULES as the argument of a search-condition. For example, the following statement retrieves rows where the COL1 column contains the same value as the CURRENT RULES special register.

```
SELECT * FROM SAMPTBL WHERE COL1 = CURRENT RULES;
```

**Db2 program preparation overview**

Before you can run an application program on Db2 for z/OS, you need to prepare it. To prepare the program, create a load module, possibly one or more packages, and an application plan.

If your application program includes SQL statements, you need to process those SQL statements by using either the Db2 precompiler or the Db2 coprocessor that is provided with a compiler. Both the precompiler and the coprocessor perform the following actions:

• Replaces the SQL statements in your source programs with calls to Db2 language interface modules
• Creates a database request module (DBRM), which communicates your SQL requests to Db2 during the bind process

The following figure illustrates the program preparation process when you use the Db2 precompiler. After you process SQL statements in your source program by using the Db2 precompiler, you create a load module, possibly one or more packages, and an application plan. Creating a load module involves compiling the modified source code that is produced by the precompiler into an object program, and link-editing the object program to create a load module. Creating a package or an application plan, a process unique to Db2, involves binding one or more DBRMs, which are created by the Db2 precompiler, using the BIND PACKAGE command.
The following figure illustrates the program preparation process when you use the Db2 coprocessor. The process is similar to the process for the Db2 precompiler, except that the Db2 coprocessor does not create modified source for your application program.

Figure 41. Program preparation with the Db2 precompiler

Figure 42. Program preparation with the Db2 coprocessor
Input and output data sets for DL/I batch jobs

DL/I batch jobs require an input data set with DD name DDITV02 and an output data set with DD name DDOTV02.

**Db2 DL/I batch input:**

Before you can run a DL/I batch job, you need to provide values for a number of input parameters. The input parameters are positional and delimited by commas.

You can specify values for the following parameters using a DDITV02 data set or a subsystem member:

- SSN, LIT, ESMT, RTT, REO, CRC

You can specify values for the following parameters **only** in a DDITV02 data set:

- CONNECTION_NAME, PLAN, PROG

If you use the DDITV02 data set and specify a subsystem member, the values in the DDITV02 DD statement override the values in the specified subsystem member. If you provide neither, Db2 abnormally terminates the application program with system abend code X'04E' and a unique reason code in register 15.

DDITV02 is the DD name for a data set that has DCB options of LRECL=80 and RECFM=F or FB.

A subsystem member is a member in the IMS procedure library. Its name is derived by concatenating the value of the SSM parameter to the value of the IMSID parameter. You specify the SSM parameter and the IMSID parameter when you invoke the DLIBATCH procedure, which starts the DL/I batch processing environment.

The meanings of the input parameters are:

- **SSN**
  - Specifies the name of the Db2 subsystem. This value is required. You must specify a name in order to make a connection to Db2.
  - The SSN value can be from one to four characters long.
  - If the value in the SSN parameter is the name of an active subsystem in the data sharing group, the application attaches to that subsystem. If the SSN parameter value is not the name of an active subsystem, but the value is a group attachment name, the application attaches to an active Db2 subsystem in the data sharing group.

- **LIT**
  - Specifies a language interface token. Db2 requires a language interface token to route SQL statements when operating in the online IMS environment. Because a batch application program can connect to only one Db2 system, Db2 does not use the LIT value.
  - The LIT value can be from zero to four characters long.
  - **Recommendation:** Specify the LIT value as SYS1.
  - You can omit the LIT value by entering SSN,,ESMT.

- **ESMT**
  - Specifies the name of the Db2 initialization module, DSNMIN10. This value is required.
  - The ESMT value must be eight characters long.

- **RTT**
  - Specifies the resource translation table. This value is optional.
  - The RTT can be from zero to eight characters long.

- **REO**
  - Specifies the region error option. This option determines what to do if Db2 is not operational or the plan is not available. The three options are:
• *R*, the default, results in returning an SQL return code to the application program. The most common SQLCODE issued in this case is -923 (SQLSTATE '57015').

• *Q* results in an abend in the batch environment; however, in the online environment, this value places the input message in the queue again.

• *A* results in an abend in both the batch environment and the online environment.

If the application program uses the XRST call, and if coordinated recovery is required on the XRST call, REO is ignored. In that case, the application program terminates abnormally if Db2 is not operational.

The REO value can be from zero to one character long.

**CRC**

Specifies the command recognition character. Because Db2 commands are not supported in the DL/I batch environment, the command recognition character is not used at this time.

The CRC value can be from zero to one character long.

**CONNECTION_NAME**

Represents the name of the job step that coordinates Db2 activities. This value is optional. If you do not specify this option, the connection name defaults are:

**Type of application**

<table>
<thead>
<tr>
<th>Default connection name</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Batch job</strong></td>
</tr>
<tr>
<td>Job name</td>
</tr>
<tr>
<td><strong>Started task</strong></td>
</tr>
<tr>
<td>Started task name</td>
</tr>
<tr>
<td><strong>TSO user</strong></td>
</tr>
<tr>
<td>TSO authorization ID</td>
</tr>
</tbody>
</table>

If a batch update job fails, you must use a separate job to restart the batch job. The connection name used in the restart job must be the same as the name that is used in the batch job that failed. Alternatively, if the default connection name is used, the restart job must have the same job name as the batch update job that failed.

Db2 requires unique connection names. If two applications try to connect with the same connection name, the second application program fails to connect to Db2.

The CONNECTION_NAME value can be from one to eight characters long.

**PLAN**

Specifies the Db2 plan name. This value is optional. If you do not specify the plan name, the application program module name is checked against the optional resource translation table. If the resource translation table has a match, the translated name is used as the Db2 plan name. If no match exists in the resource translation table, the application program module name is used as the plan name.

The PLAN value can be from zero to eight characters long.

**PROG**

Specifies the application program name. This value is required. It identifies the application program that is to be loaded and to receive control.

The PROG value can be from one to eight characters long.

**Example:** An example of the fields in the record is shown below:

```
DSN, SYS1, DSNMIN10, , R, - , BATCH001, DB2PLAN, PROGA
```

**Db2 DL/I batch output:**

In an online IMS environment, Db2 sends unsolicited status messages to the master terminal operator (MTO) and records on indoubt processing and diagnostic information to the IMS log. In a batch environment, Db2 sends this information to the output data set that is specified in the DDOTV02 DD
statement. Ensure that the output data set has DCB options of RECFM=V or VB, LRECL=4092, and BLKSIZE of at least LRECL + 4. If the DD statement is missing, Db2 issues the message IEC130I and continues processing without any output.

You might want to save and print the data set, as the information is useful for diagnostic purposes. You can use the IMS module, DFSERA10, to print the variable-length data set records in both hexadecimal and character format.

**Related concepts**
- Submitting work to be processed (Db2 Data Sharing Planning and Administration)

## Db2-supplied JCL procedures for preparing an application

You can precompile and prepare an application program using a Db2-supplied JCL procedure. Db2 has a unique JCL procedure for each supported language, with appropriate defaults for starting the Db2 precompiler and host language compiler or assembler. The procedures are in prefix.SDSNSAMP member DSNTIJMV, which installs the procedures.

### Table 147. Procedures for precompiling programs

<table>
<thead>
<tr>
<th>Language</th>
<th>Procedure</th>
<th>Invocation included in...</th>
</tr>
</thead>
<tbody>
<tr>
<td>High-level assembler</td>
<td>DSNHASM</td>
<td>DSNTEJ2A</td>
</tr>
<tr>
<td>C</td>
<td>DSNHC</td>
<td>DSNTEJ2D</td>
</tr>
<tr>
<td>C++</td>
<td>DSNHCPPDSNHCPP2</td>
<td>DSNTEJ2EN/A</td>
</tr>
<tr>
<td>Enterprise COBOL</td>
<td>DSNHICOB</td>
<td>DSNTEJ2C¹</td>
</tr>
<tr>
<td>Fortran</td>
<td>DSNHFOR</td>
<td>DSNTEJ2F</td>
</tr>
<tr>
<td>PL/I</td>
<td>DSNHPLI</td>
<td>DSNTEJ2P</td>
</tr>
<tr>
<td>SQL</td>
<td>DSNHSQL</td>
<td>DSNTEJ63</td>
</tr>
</tbody>
</table>

**Notes:**

1. You must customize these programs to invoke the procedures that are listed in this table.
2. This procedure demonstrates how you can prepare an object-oriented program that consists of two data sets or members, both of which contain SQL.

If you use the PL/I macro processor, you must not use the PL/I *PROCESS statement in the source to pass options to the PL/I compiler. You can specify the needed options on the PARM.PLI= parameter of the EXEC statement in the DSNHPLI procedure.

### JCL to include the appropriate interface code when using the Db2-supplied JCL procedures

To include the proper interface code when you submit the JCL procedures, use an INCLUDE SYSLIB statement in your link-edit JCL. The statement should specify the correct language interface module for the environment.

**TSO, batch**

```plaintext
//LKED.SYSIN DD *
INCLUDE SYSLIB(member)
/*
```

*member* must be DSNELI or DSNULI, except for FORTRAN, in which case *member* must be DSNHFT.
IMS

//LKED.SYSIN DD *
INCLUDE SYSLIB(DFSLI000)
ENTRY (specification)
/

DFSLI000 is the module for DL/I batch attach.
ENTRY specification varies depending on the host language. Include one of the following:

DLITCBL, for COBOL applications
PLICALLA, for PL/I applications
The program name, for assembler language applications.

**Recommendation:** For COBOL applications, specify the PSB linkage directly on the PROCEDURE DIVISION statement instead of on a DLITCBL entry point. When you specify the PSB linkage directly on the PROCEDURE DIVISION statement, you can either omit the ENTRY specification or specify the application program name instead of the DLITCBL entry point.

CICS

//LKED.SYSIN DD *
INCLUDE SYSLIB(member)
/

member must be DSNCLI or DSNULI.

**Related concepts**
“Universal language interface (DSNULI)” on page 113
The universal language interface (DSNULI) subcomponent determines the runtime environment and dynamically loads and branches to the appropriate language interface module.

**Related tasks**
Making the CAF language interface (DSNALI) available
Before you can invoke the call attachment facility (CAF), you must first make DSNALI available.

Compiling and link-editing an application
If you use the Db2 precompiler, your next step in the program preparation process is to compile and link-edit your program. As with the precompile step, you have a choice of methods.

**Tailoring Db2-supplied JCL procedures for preparing CICS programs**

Instead of using the Db2 Program Preparation panels to prepare your CICS program, you can tailor CICS-supplied JCL procedures to do that. To tailor a CICS procedure, you need to add some steps and change some DD statements.

**About this task**

Make changes as needed to perform the following actions:

• Process the program with the Db2 precompiler.
• Bind the application plan. You can do this any time after you precompile the program. You can bind the program either online by the DB2I panels or as a batch step in this or another z/OS job.
• Include a DD statement in the linkage editor step to access the Db2 load library.
• Be sure the linkage editor control statements contain an INCLUDE statement for the Db2 language interface module.
The following example illustrates the necessary changes. This example assumes the use of a COBOL program. For any other programming language, change the CICS procedure name and the Db2 precompiler options.

```cobol
//TESTC01 JOB
//** //***********************************************************************
//** DB2 PRECOMPILE THE COBOL PROGRAM
//***********************************************************************
(1) //PC EXEC PGM=DSNHPC,
(1) //PARAM=HOST(COB2),XREF,SOURCE,FLAG(I),APOST'
(1) //STEPLIB DD DISP=SHR,DSN=prefix.SDSNEXIT
(1) // DD DISP=SHR,DSN=prefix.SDSNLOAD
(1) //DBRMLIB DD DISP=OLD,DSN=USER.DBRMLIB.DATA(TESTC01)
(1) //SYSCIN DD DSN=&&DSNHOUT,DISP=(MOD,PASS),UNIT=SYSDA,
(1) // SPACE=(800,(500,500))
(1) //SYSLIB DD DISP=SHR,DSN=USER.SRCLIB.DATA
(1) //SYSPRINT DD SYSOUT=*  
(1) //SYSTERM DD SYSOUT=*  
(1) //SYSDUMP DD SYSOUT=*  
(1) //SYSIN DD DISP=SHR,DSN=USER.SRCLIB.DATA(TESTC01)
(1) //***********************************************************************
(1) //***    BIND THIS PROGRAM.
(1) //***********************************************************************
(2) //BIND EXEC PGM=IKJEFT01,
(2) // COND=((4,LT,PC))
(2) //STEPLIB DD DISP=SHR,DSN=prefix.SDSNEXIT
(2) // DD DISP=SHR,DSN=prefix.SDSNLOAD
(2) //DBRMLIB DD DISP=OLD,DSN=USER.DBRMLIB.DATA(TESTC01)
(2) //SYSPRINT DD SYSOUT=*  
(2) //SYSTSPRT DD SYSOUT=*  
(2) //SYSDUMP DD SYSOUT=*  
(2) //SYSTSIN DD *
(2) // DSN S(DSN)
(2) // BIND PLAN(TESTC01) MEMBER(TESTC01) ACTION(REP) RETAIN ISOLATION(CS)
END
//***********************************************************************
(2) /// COMPIL THE COBOL PROGRAM
(2) ///***********************************************************************
(3) //CICS EXEC DFHEITVL
(4) //TRN.SYSPRT DD DSN=&&DSNHOUT,DISP=(MOD,DELETE)
(5) //LKED.SYSLMOD DD DSN=USER.RUNLIB.LOAD
(6) //LKED.CICSLOAD DD DISP=SHR,DSN=prefix.SDFHLOAD
//LKED.SYSPRT DD *
(7) // INCLUDE CICSLOAD(DSNCLI)
NAME TESTC01(R)
//***********************************************************************
```

The procedure accounts for these steps:

**Step 1.** Precompile the program. The output of the Db2 precompiler becomes the input to the CICS command language translator.

**Step 2.** Bind the application plan.

**Step 3.** Call the CICS procedure to translate, compile, and link-edit a COBOL program. This procedure has several options that you need to consider.

**Step 4.** Reflect an application load library in the data set name of the SYSLMOD DD statement. You must include the name of this load library in the DFHRPL DD statement of the CICS run time JCL.

**Step 5.** Name the CICS load library that contains the module DSNCLI.

**Step 6.** Direct the linkage editor to include the CICS-Db2 language interface module (DSNCLI). In this example, the order of the various control sections (CSECTs) is of no concern because the structure of the procedure automatically satisfies any order requirements.

For more information about the procedure DFHEITVL, other CICS procedures, or CICS requirements for application programs, please see the appropriate CICS manual.

If you are preparing a particularly large or complex application, you can use another preparation method. For example, if your program requires four of your own link-edit include libraries, you cannot prepare the
program with DB2I, because DB2I limits the number of include libraries to three, plus language, IMS or CICS, and Db2 libraries. Therefore, you would need another preparation method. Be careful to use the correct language interface.

Related reference
Data sets that the precompiler uses
When you invoke the precompiler you need to provide data sets that contain input for the precompiler, such as the host programming statements and SQL statements. You also need to provide data sets where the precompiler can store its output, such as the modified source code and diagnostics messages.

DB2I panels that are used for program preparation

DB2I contains a set of panels that let you prepare an application for execution.

The following table describes each of the panels that you need to use to prepare an application.

<table>
<thead>
<tr>
<th>Panel name</th>
<th>Panel description</th>
</tr>
</thead>
<tbody>
<tr>
<td>“Db2 Program Preparation panel” on page 901</td>
<td>Lets you choose specific program preparation functions to perform. For the functions that you choose, you can also display the associated panels to specify options for performing those functions. This panel also lets you change the DB2I default values and perform other precompile and prelink functions.</td>
</tr>
<tr>
<td>“DB2I Defaults Panel 1” on page 905</td>
<td>Lets you change many of the system defaults that are set at Db2 installation time.</td>
</tr>
<tr>
<td>“DB2I Defaults Panel 2” on page 907</td>
<td>Lets you change your default job statement and set additional COBOL options.</td>
</tr>
<tr>
<td>“Precompile panel” on page 908</td>
<td>Lets you specify values for precompile functions. You can reach this panel directly from the DB2I Primary Option Menu or from the Db2 Program Preparation panel. If you reach this panel from the Program Preparation panel, many of the fields contain values from the Primary and Precompile panels.</td>
</tr>
<tr>
<td>“Bind Package panel” on page 910</td>
<td>Lets you change many options when you bind a package. You can reach this panel directly from the DB2I Primary Option Menu or from the Db2 Program Preparation panel. If you reach this panel from the Db2 Program Preparation panel, many of the fields contain values from the Primary and Precompile panels.</td>
</tr>
<tr>
<td>“Bind Plan panel” on page 912</td>
<td>Lets you change options when you bind an application plan. You can reach this panel directly from the DB2I Primary Option Menu or as a part of the program preparation process. This panel also follows the Bind Package panels.</td>
</tr>
<tr>
<td>“Defaults for Bind Package and Defaults for Rebind Package panels” on page 915</td>
<td>Let you change the defaults for BIND or REBIND PACKAGE or PLAN.</td>
</tr>
<tr>
<td>“System Connection Types panel” on page 919</td>
<td>Lets you specify a system connection type. This panel displays if you choose to enable or disable connections on the Bind or Rebind Package or Plan panels.</td>
</tr>
</tbody>
</table>
Table 148. DB2I panels used for program preparation (continued)

<table>
<thead>
<tr>
<th>Panel name</th>
<th>Panel description</th>
</tr>
</thead>
<tbody>
<tr>
<td>“Panels for entering lists of values” on page 921</td>
<td>Let you enter or modify an unlimited number of values. A list panel looks similar to an ISPF edit session and lets you scroll and use a limited set of commands.</td>
</tr>
<tr>
<td>“Program Preparation: Compile, Link, and Run panel” on page 922</td>
<td>Lets you perform the last two steps in the program preparation process (compile and link-edit). This panel also lets you do the PL/I MACRO PHASE for programs that require this option. For TSO programs, the panel also lets you run programs.</td>
</tr>
</tbody>
</table>

Related reference
The DB2I primary option menu (Introduction to Db2 for z/OS)
DB2I panels that are used to rebind and free plans and packages
A set of DB2I panels lets you bind, rebind, or free packages.

Db2 Program Preparation panel

The Db2 Program Preparation panel lets you choose which specific program preparation function to perform.

For the functions you choose, you can also choose to display the associated panels to specify options for performing those functions. Some of the functions you can select are:

Precompile
The panel for this function lets you control the Db2 precompiler.

Bind a package
The panel for this function lets you bind your program's DBRM to a package and change your defaults for binding the packages.

Bind a plan
The panel for this function lets you create your program's application plan and change your defaults for binding the plans.

Compile, link, and run
The panel for these functions let you control the compiler or assembler and the linkage editor.

TSO and batch: For TSO programs, you can use the program preparation programs to control the host language run time processor and the program itself.

The Program Preparation panel also lets you change the DB2I default values, and perform other precompile and prelink functions.

On the Db2 Program Preparation panel, shown in the following figure, enter the name of the source program data set (this example uses SAMPLEPG.COBOL) and specify the other options you want to include. When finished, press ENTER to view the next panel.
The following explains the functions on the Db2 Program Preparation panel and how to complete the necessary fields in order to start program preparation.

1 INPUT DATA SET NAME

Lets you specify the input data set name. The input data set name can be a PDS or a sequential data set, and can also include a member name. If you do not enclose the data set name in apostrophes, a standard TSO prefix (user ID) qualifies the data set name. The input data set name you specify is used to precompile, bind, link-edit, and run the program.

2 DATA SET NAME QUALIFIER

Lets you qualify temporary data set names involved in the program preparation process. Use any character string from 1 to 8 characters that conforms to normal TSO naming conventions. (The default is TEMP.)

For programs that you prepare in the background or that use EDITJCL for the PREPARATION ENVIRONMENT option, Db2 creates a data set named tsoprefix.qualifier.CNTL to contain the program preparation JCL. The name tsoprefix represents the prefix TSO assigns, and qualifier represents the value you enter in the DATA SET NAME QUALIFIER field. If a data set with this name already exists, Db2 deletes it.

3 PREPARATION ENVIRONMENT

Lets you specify whether program preparation occurs in the foreground or background. You can also specify EDITJCL, in which case you are able to edit and then submit the job. Use:

   FOREGROUND to use the values you specify on the Program Preparation panel and to run immediately.
   BACKGROUND to create and submit a file containing a DSNH CLIST that runs immediately using the JOB control statement from either the DB2I Defaults panel or your site's SUBMIT exit. The file is saved.
   EDITJCL to create and open a file containing a DSNH CLIST in edit mode. You can then submit the CLIST or save it.

4 RUN TIME ENVIRONMENT

Lets you specify the environment (TSO, CAF, CICS, IMS, RRSAF) in which your program runs. All programs are prepared under TSO, but can run in any of the environments. If you specify CICS, IMS, or RRSAF, then you must set the RUN field to NO because you cannot run such programs from the Program Preparation panel. If you set the RUN field to YES, you can specify only TSO or CAF. (Batch programs also run under the TSO Terminal Monitor Program. You therefore need to specify TSO in this field for batch programs.)
5 OTHER DSNH OPTIONS

Lets you specify a list of DSNH options that affect the program preparation process, and that override options specified on other panels. If you are using CICS, these can include options you want to specify to the CICS command translator.

If you specify options in this field, separate them by commas. You can continue listing options on the next line, but the total length of the option list can be no more than 70 bytes.

Fields 6 through 15 let you select the function to perform and to choose whether to show the DB2I panels for the functions you select. Use Y for YES, or N for NO.

If you are willing to accept default values for all the steps, enter N under Display panel? for all the other preparation panels listed.

To make changes to the default values, entering Y under Display panel? for any panel you want to see. DB2I then displays each of the panels that you request. After all the panels display, Db2 proceeds with the steps involved in preparing your program to run.

Variables for all functions used during program preparation are maintained separately from variables entered from the DB2I Primary Option Menu. For example, the bind plan variables you enter on the Program Preparation panel are saved separately from those on any Bind Plan panel that you reach from the Primary Option Menu.

6 CHANGE DEFAULTS

Lets you specify whether to change the DB2I defaults. Enter Y in the Display panel? field next to this option; otherwise enter N. Minimally, you should specify your subsystem identifier and programming language on the Defaults panel.

7 PL/I MACRO PHASE

Lets you specify whether to display the "Program Preparation: Compile, Link, and Run" panel to control the PL/I macro phase by entering PL/I options in the OPTIONS field of that panel. That panel also displays for options COMPILE OR ASSEMBLE, LINK, and RUN.

This field applies to PL/I programs only. If your program is not a PL/I program or does not use the PL/I macro processor, specify N in the Perform function field for this option, which sets the Display panel? field to the default N.

8 PRECOMPILE

Lets you specify whether to display the Precompile panel. To see this panel enter Y in the Display panel? field next to this option; otherwise, enter N.

9 CICS COMMAND TRANSLATION

Lets you specify whether to use the CICS command translator. This field applies to CICS programs only.

IMS and TSO: If you run under TSO or IMS, ignore this step; this allows the Perform function field to default to N.

CICS: If you are using CICS and have precompiled your program, you must translate your program using the CICS command translator.

The command translator does not have a separate DB2I panel. You can specify translation options on the Other Options field of the Db2 Program Preparation panel, or in your source program if it is not an assembler program.

Because you specified a CICS run time environment, the Perform function column defaults to Y. Command translation takes place automatically after you precompile the program.

10 BIND PACKAGE

 Lets you specify whether to display the Bind Package panel. To see it, enter Y in the Display panel? field next to this option; otherwise, enter N.

11 BIND PLAN

Lets you specify whether to display the Bind Plan panel. To see it, enter Y in the Display panel? field next to this option; otherwise, enter N.
12 COMPILe OR ASSEMBLe

Lets you specify whether to display the "Program Preparation: Compile, Link, and Run" panel. To see this panel enter Y in the Display panel? field next to this option; otherwise, enter N.

13 PRELINK

Lets you use the prelink utility to make your C, C++, or Enterprise COBOL for z/OS program reentrant. This utility concatenates compile-time initialization information from one or more text decks into a single initialization unit. To use the utility, enter Y in the Display panel? field next to this option; otherwise, enter N. If you request this step, then you must also request the compiler step and the link-edit step.

14 LINK

Lets you specify whether to display the "Program Preparation: Compile, Link, and Run" panel. To see it, enter Y in the Display panel? field next to this option; otherwise, enter N. If you specify Y in the Display panel? field for the COMPILe OR ASSEMBLe option, you do not need to make any changes to this field; the panel displayed for COMPILe OR ASSEMBLe is the same as the panel displayed for LINK. You can make the changes you want to affect the link-edit step at the same time you make the changes to the compiler step.

15 RUN

Lets you specify whether to run your program. The RUN option is available only if you specify TSO or CAF for RUN TIME ENVIRONMENT.

If you specify Y in the Display panel? field for the COMPILe OR ASSEMBLe or LINK option, you can specify N in this field, because the panel displayed for COMPILe OR ASSEMBLe and for LINK is the same as the panel displayed for RUN.

IMS and CICS: IMS and CICS programs cannot run using DB2I. If you are using IMS or CICS, use N in these fields.

TSO and batch: If you are using TSO and want to run your program, you must enter Y in the Perform function column next to this option. You can also indicate that you want to specify options and values to affect the running of your program, by entering Y in the Display panel column.

Pressing ENTER takes you to the first panel in the series you specified, in this example to the DB2I Defaults panel. If, at any point in your progress from panel to panel, you press the END key, you return to this first panel, from which you can change your processing specifications. Asterisks (*) in the Display panel? column of rows 7 through 14 indicate which panels you have already examined. You can see a panel again by writing a Y over an asterisk.

Related reference

Bind Package panel
The Bind Package panel is the first of two DB2I panels that request information about how you want to bind a package.

Bind Plan panel
The Bind Plan panel is the first of two DB2I panels that request information about how you want to bind an application plan.

DB2I Defaults Panel 1
DB2I Defaults Panel 1 lets you change many of the system default values that were set at Db2 installation time.

Defaults for Bind Package and Defaults for Rebind Package panels
These DB2I panels let you change your defaults for BIND PACKAGE and REBIND PACKAGE options.

Defaults for Bind Plan and Defaults for Rebind Plan panels
These DB2I panels let you change your defaults for BIND PLAN and REBIND PLAN options.

Precompile panel
After you set the DB2I defaults, you can precompile your application. You can reach the Precompile panel by specifying it as a part of the program preparation process from the Db2 Program Preparation panel. Or you can reach it directly from the DB2I Primary Option Menu.

Program Preparation: Compile, Link, and Run panel
The Compile, Link, and Run panel lets you perform the last two steps in the program preparation process (compile and link-edit). This panel also lets you perform the PL/I MACRO PHASE for programs that require this option.

DSNH (TSO CLIST) (Db2 Commands)
Prelinking an application (z/OS Language Environment Programming Guide)

**DB2I Defaults Panel 1**

DB2I Defaults Panel 1 lets you change many of the system default values that were set at Db2 installation time.

The following figure shows the fields that affect the processing of the other DB2I panels.

![Figure 44. DB2I Defaults Panel 1](image)

The following explains the fields on DB2I Defaults Panel 1.

1. **Db2 NAME**

   Lets you specify the Db2 subsystem that processes your DB2I requests. If you specify a different Db2 subsystem, its identifier displays in the SSID (subsystem identifier) field located at the top, right side of your screen. The default is DSN.

2. **Db2 CONNECTION RETRIES**

   Lets you specify the number of additional times to attempt to connect to Db2, if Db2 is not up when the program issues the DSN command. The program preparation process does not use this option.

   Use a number from 0 to 120. The default is 0. Connections are attempted at 30-second intervals.

3. **APPLICATION LANGUAGE**

   Lets you specify the default programming language for your application program. You can specify any of the following languages:

   - **ASM**
     For High Level Assembler/z/OS
   - **C**
     For C language
   - **CPP**
     For C++
   - **IBMCOB**
     For Enterprise COBOL for z/OS. This option is the default.
   - **FORTRAN**
     For VS Fortran
   - **PLI**
     For PL/I
If you specify IBMCOB, Db2 prompts you for more COBOL defaults on panel DSNEOP02. See “DB2I Defaults Panel 2” on page 907.

You cannot specify FORTRAN for IMS or CICS programs.

4 LINES/PAGE OF LISTING
Let you specify the number of lines to print on each page of listing or SPUFI output. The default is 60.

5 MESSAGE LEVEL
Let you specify the lowest level of message to return to you during the BIND phase of the preparation process. Use:

I
  For all information, warning, error, and severe error messages

W
  For warning, error, and severe error messages

E
  For error and severe error messages

S
  For severe error messages only

6 SQL STRING DELIMITER
Let you specify the symbol used to delimit a string in SQL statements in COBOL programs. This option is valid only when the application language is IBMCOB. Use:

DEFAULT
  To use the default defined at installation time

' 
  For an apostrophe

" 
  For a quotation mark

7 DECIMAL POINT
Let you specify how your host language source program represents decimal separators and how SPUFI displays decimal separators in its output. Use a comma (,) or a period (.). The default is a period (.).

8 STOP IF RETURN CODE >=
Let you specify the smallest value of the return code (from precompile, compile, link-edit, or bind) that will prevent later steps from running. Use:

4
  To stop on warnings and more severe errors.

8
  To stop on errors and more severe errors. The default is 8.

9 NUMBER OF ROWS
Let you specify the default number of input entry rows to generate on the initial display of ISPF panels. The number of rows with non-blank entries determines the number of rows that appear on later displays.

10 AS USER
Let you specify a user ID to associate with the trusted connection for the current DB2I session.

Db2 establishes the trusted connection for the user that you specify if the following conditions are true:

- The primary authorization ID that Db2 obtains after running the connection exit is allowed to use the trusted connection without authentication.
- The security label, if defined either implicitly or explicitly in the trusted context for the user, is defined in RACF for the user.
After Db2 establishes the trusted connection, the primary authorization ID, any secondary authorization IDs, any role, and any security label that is associated with the user ID that is specified in the AS USER field are used for the trusted connection. Db2 uses this security label to verify multilevel security for the user.

If the primary authorization ID that is associated with the user ID that is specified in the AS USER field is not allowed to use the trusted connection or requires authentication information, the connection request fails. If Db2 cannot verify the security label, the connection request also fails.

The value that you enter in this field is retained only for the length of the DB2I session. The field is reset to blank when you exit DB2I.

Suppose that the default programming language is PL/I and the default number of lines per page of program listing is 60. Your program is in COBOL, so you want to change field 3, APPLICATION LANGUAGE. You also want to print 80 lines to the page, so you need to change field 4, LINES/PAGE OF LISTING, as well. Figure 44 on page 905 shows the entries that you make in DB2I Defaults Panel 1 to make these changes. In this case, pressing ENTER takes you to Db2 Defaults Panel 2.

**DB2I Defaults Panel 2**

After you press Enter on the DB2I Defaults Panel 1, the DB2I Defaults Panel 2 is displayed. If you chose IBMCOB as the language on the DB2I Defaults Panel 1, three fields are displayed. Otherwise, only the first field is displayed.

The following figure shows the DB2I Defaults Panel 2 when IBMCOB is selected.

<table>
<thead>
<tr>
<th>DSNEOP02</th>
<th>DB2I DEFAULTS PANEL 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>COMMAND ===&gt;</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>Change defaults as desired:</td>
<td></td>
</tr>
<tr>
<td>1  DB2I JOB STATEMENT: (Optional if your site has a SUBMIT exit)</td>
<td></td>
</tr>
<tr>
<td>===&gt; /*USRT001A JOB (ACCOUNT), 'NAME' */</td>
<td></td>
</tr>
<tr>
<td>===&gt; /* */</td>
<td></td>
</tr>
<tr>
<td>===&gt; /* */</td>
<td></td>
</tr>
<tr>
<td>2  COBOL STRING DELIMITER ===&gt; DEFAULT (For IBMCOB)</td>
<td></td>
</tr>
<tr>
<td>(DEFAULT, ' or &quot;)</td>
<td></td>
</tr>
<tr>
<td>3  DBCS SYMBOL FOR DCLGEN ===&gt; G (G/N - Character in PIC clause)</td>
<td></td>
</tr>
</tbody>
</table>

**Figure 45. DB2I Defaults Panel 2**

1 **DB2I JOB STATEMENT**

   Lets you change your default job statement. Specify a job control statement, and optionally, a JOBLIB statement to use either in the background or the EDITJCL program preparation environment. Use a JOBLIB statement to specify run time libraries that your application requires. If your program has a SUBMIT exit routine, Db2 uses that routine. If that routine builds a job control statement, you can leave this field blank.

2 **COBOL STRING DELIMITER**

   Lets you specify the symbol used to delimit a string in a COBOL statement in a COBOL application. Use:

   **DEFAULT**
   
   To use the default defined at installation time
   
   ' For an apostrophe
   
   " For a quotation mark

   Leave this field blank to accept the default value.
3 DBCS Symbol for DCLGEN

Lets you enter either G (the default) or N, to specify whether DCLGEN generates a picture clause that has the form PIC G(n) DISPLAY-1 or PIC N(n).

Leave this field blank to accept the default value.

Pressing ENTER takes you to the next panel you specified on the Db2 Program Preparation panel, in this case, to the Precompile panel.

Precompile panel

After you set the DB2I defaults, you can precompile your application. You can reach the Precompile panel by specifying it as a part of the program preparation process from the Db2 Program Preparation panel. Or you can reach it directly from the DB2I Primary Option Menu.

The way you choose to reach the panel determines the default values of the fields it contains. The following figure shows the Precompile panel.

![Figure 46. The Precompile panel](image)

The following explains the functions on the Precompile panel, and how to enter the fields for preparing to precompile.

1 INPUT DATA SET

Lets you specify the data set name of the source program and SQL statements to precompile. If you reached this panel through the Db2 Program Preparation panel, this field contains the data set name specified there. You can override it on this panel. If you reached this panel directly from the DB2I Primary Option Menu, you must enter the data set name of the program you want to precompile. The data set name can include a member name. If you do not enclose the data set name with apostrophes, a standard TSO prefix (user ID) qualifies the data set name.

2 INCLUDE LIBRARY

Lets you enter the name of a library containing members that the precompiler should include. These members can contain output from DCLGEN. If you do not enclose the name in apostrophes, a standard TSO prefix (user ID) qualifies the name.

You can request additional INCLUDE libraries by entering DSNH CLIST parameters of the form PnLIB(dsname), where n is 2, 3, or 4) on the OTHER OPTIONS field of this panel or on the OTHER DSNH OPTIONS field of the Program Preparation panel.

3 DSNAME QUALIFIER

Lets you specify a character string that qualifies temporary data set names during precompile. Use any character string from 1 to 8 characters in length that conforms to normal TSO naming conventions.

If you reached this panel through the Db2 Program Preparation panel, this field contains the data set name qualifier specified there. You can override it on this panel.
If you reached this panel from the DB2I Primary Option Menu, you can either specify a DSNAME QUALIFIER or let the field take its default value, TEMP.

**IMS and TSO:** For IMS and TSO programs, Db2 stores the precompiled source statements (to pass to the compiler or assemble step) in a data set named `tsoprefix.qualifier.suffix`. A data set named `tsoprefix.qualifier.PCLIST` contains the precompiler print listing.

For programs prepared in the background or that use the PREPARATION ENVIRONMENT option EDITJCL (on the Db2 Program Preparation panel), a data set named `tsoprefix.qualifier.CNTL` contains the program preparation JCL.

In these examples, `tsoprefix` represents the prefix TSO assigns, often the same as the authorization ID. `qualifier` represents the value entered in the DSNAME QUALIFIER field. `suffix` represents the output name, which is one of the following: COBOL, FORTRAN, C, PLI, ASM, DECK, CICSIN, OBJ, or DATA. In the Precompile Panel that is shown above, the data set `tsoprefix.TEMP.COBOL` contains the precompiled source statements, and `tsoprefix.TEMP.PCLIST` contains the precompiler print listing. If data sets with these names already exist, then Db2 deletes them.

**CICS:** For CICS programs, the data set `tsoprefix.qualifier.suffix` receives the precompiled source statements in preparation for CICS command translation.

If you do not plan to do CICS command translation, the source statements in `tsoprefix.qualifier.suffix`, are ready to compile. The data set `tsoprefix.qualifier.PCLIST` contains the precompiler print listing.

When the precompiler completes its work, control passes to the CICS command translator. Because there is no panel for the translator, translation takes place automatically. The data set `tsoprefix.qualifier.CXLIST` contains the output from the command translator.

**4 DBRM DATA SET**

Lets you name the DBRM library data set for the precompiler output. The data set can also include a member name.

When you reach this panel, the field is blank. When you press ENTER, however, the value contained in the DSNAME QUALIFIER field of the panel, concatenated with `DBRM`, specifies the DBRM data set: `qualifier.DBRM`.

You can enter another data set name in this field only if you allocate and catalog the data set before doing so. This is true even if the data set name that you enter corresponds to what is otherwise the default value of this field.

The precompiler sends modified source code to the data set `qualifier.host`, where `host` is the language specified in the APPLICATION LANGUAGE field of DB2I Defaults panel 1.

**5 WHERE TO PRECOMPILE**

Lets you indicate whether to precompile in the foreground or background. You can also specify EDITJCL, in which case you are able to edit and then submit the job.

If you reached this panel from the Db2 Program Preparation panel, the field contains the preparation environment specified there. You can override that value if you want.

If you reached this panel directly from the DB2I Primary Option Menu, you can either specify a processing environment or allow this field to take its default value. Use:

- **FOREGROUND** to immediately precompile the program with the values you specify in these panels.
- **BACKGROUND** to create and immediately submit to run a file containing a DSNH CLIST using the JOB control statement from either DB2I Defaults Panel 2 or your site's SUBMIT exit. The file is saved.
- **EDITJCL** to create and open a file containing a DSNH CLIST in edit mode. You can then submit the CLIST or save it.

**6 VERSION**

Lets you specify the version of the program and its DBRM. If the version contains the maximum number of characters permitted (64), you must enter each character with no intervening blanks from one line to the next. This field is optional.
7 OTHER OPTIONS
Lets you enter any option that the DSNH CLIST accepts, which gives you greater control over your program. The DSNH options you specify in this field override options specified on other panels. The option list can continue to the next line, but the total length of the list can be no more than 70 bytes.

Related reference
DSNH (TSO CLIST) (Db2 Commands)

Bind Package panel
The Bind Package panel is the first of two DB2I panels that request information about how you want to bind a package.

You can reach the Bind Package panel either directly from the DB2I Primary Option Menu, or as a part of the program preparation process. If you enter the Bind Package panel from the Program Preparation panel, many of the Bind Package entries contain values from the Primary and Precompile panels. Figure 47 on page 910 shows the Bind Package panel.

![Bind Package panel](image)

The following information explains the functions on the Bind Package panel and how to fill the necessary fields in order to bind your program.

1 LOCATION NAME
Lets you specify the system at which to bind the package. You can use from 1 to 16 characters to specify the location name. The location name must be defined in the catalog table SYSIBM.LOCATIONS. The default is the local DBMS.

2 COLLECTION-ID
Lets you specify the collection the package is in. You can use from 1 to 128 characters to specify the collection, and the first character must be alphabetic. This field is scrollable.

3 DBRM: COPY:
Lets you specify whether you are creating a new package (DBRM) or making a copy of a package that already exists (COPY). Use:

DBRM
To create a new package. You must specify values in the LIBRARY, PASSWORD, and MEMBER fields.

COPY
To copy an existing package. You must specify values in the COLLECTION-ID and PACKAGE-ID fields. (The VERSION field is optional.)
4 MEMBER or COLLECTION-ID

**MEMBER (for new packages):** If you are creating a new package, this option lets you specify the DBRM to bind. You can specify a member name from 1 to 128 characters. This field is scrollable. The default name depends on the input data set name.

- If the input data set is partitioned, the default name is the member name of the input data set specified in the INPUT DATA SET NAME field of the Db2 Program Preparation panel.
- If the input data set is sequential, the default name is the second qualifier of this input data set.

**COLLECTION-ID (for copying a package):** If you are copying a package, this option specifies the collection ID that contains the original package. You can specify a collection ID from 1 to 128 characters, which must be different from the collection ID specified on the PACKAGE ID field. This field is scrollable.

5 PASSWORD or PACKAGE-ID

**PASSWORD (for new packages):** If you are creating a new package, this lets you enter password for the library you list in the LIBRARY field. You can use this field only if you reached the Bind Package panel directly from the Db2 Primary Option Menu. This field is scrollable.

**PACKAGE-ID (for copying packages):** If you are copying a package, this option lets you specify the name of the original package. You can enter a package ID from 1 to 128 characters. This field is scrollable.

6 LIBRARY or VERSION

**LIBRARY (for new packages):** If you are creating a new package, this lets you specify the names of the libraries that contain the DBRMs specified on the MEMBER field for the bind process. Libraries are searched in the order specified and must in the catalog tables.

**VERSION (for copying packages):** If you are copying a package, this option lets you specify the version of the original package. You can specify a version ID from 1 to 64 characters.

7 OPTIONS

Lets you specify which bind options Db2 uses when you issue BIND PACKAGE with the COPY option. Specify:

- **COMPOSITE (default)** to cause Db2 to use any options you specify in the BIND PACKAGE command. For all other options, Db2 uses the options of the copied package.

- **COMMAND** to cause Db2 to use the options you specify in the BIND PACKAGE command. For all other options, Db2 uses the following values:
  - For a local copy of a package, Db2 uses the defaults for the local Db2 subsystem.
  - For a remote copy of a package, Db2 uses the defaults for the server on which the package is bound.

8 CHANGE CURRENT DEFAULTS?

Lets you specify whether to change the current defaults for binding packages. If you enter YES in this field, you see the Defaults for Bind Package panel as your next step. You can enter your new preferences there; for instructions, see “Defaults for Bind Package and Defaults for Rebind Package panels” on page 915.

9 ENABLE/DISABLE CONNECTIONS?

Lets you specify whether you want to enable and disable system connections types to use with this package. This is valid only if the LOCATION NAME field names your local Db2 system.

Placing YES in this field displays a panel (shown in Figure 53 on page 920) that lets you specify whether various system connections are valid for this application. You can specify connection names to further identify enabled connections within a connection type. A connection name is valid only when you also specify its corresponding connection type.

The default enables all connection types.
10 OWNER OF PACKAGE (AUTHID)
Lets you specify the primary authorization ID of the owner of the new package. That ID is the name owning the package, and the name associated with all accounting and trace records produced by the package.

The owner must have the privileges required to run SQL statements contained in the package.

The default is the primary authorization ID of the bind process.

The field is scrollable, and the maximum field length is 128.

11 QUALIFIER
Lets you specify the default schema for unqualified tables, views, indexes, and aliases. You can specify a schema name from 1 to 128 characters. The default is the authorization ID of the package owner. This field is scrollable.

12 ACTION ON PACKAGE
Lets you specify whether to replace an existing package or create a new one. Use:

   REPLACE (default) to replace the package named in the PACKAGE-ID field if it already exists, and add it if it does not. (Use this option if you are changing the package because the SQL statements in the program changed. If only the SQL environment changes but not the SQL statements, you can use REBIND PACKAGE.)

   ADD to add the package named in the PACKAGE-ID field, only if it does not already exist.

13 INCLUDE PATH?
Indicates whether you will supply a list of schema names that Db2 searches when it resolves unqualified distinct type, user-defined function, and stored procedure names in SQL statements. The default is NO. If you specify YES, Db2 displays a panel in which you specify the names of schemas for Db2 to search.

14 REPLACE VERSION
Lets you specify whether to replace a specific version of an existing package or create a new one. If the package and the version named in the PACKAGE-ID and VERSION fields already exist, you must specify REPLACE. You can specify a version ID from 1 to 64 characters. The default version ID is that specified in the VERSION field.

Bind Plan panel
The Bind Plan panel is the first of two DB2I panels that request information about how you want to bind an application plan.

Like the Precompile panel, you can reach the Bind Plan panel either directly from the DB2I Primary Option Menu, or as a part of the program preparation process. You must have an application plan, even if you bind your application to packages; this panel also follows the Bind Package panels.

If you enter the Bind Plan panel from the Program Preparation panel, many of the Bind Plan entries contain values from the Primary and Precompile panels.
Enter primary package list:
1 LOCATION NAME ............. ===> (Defaults to local)
2 COLLECTION ID ............. ===> > (Required)
3 PACKAGE ID ................. ===> (Package ID or *)
4 ADDITIONAL PACKAGE LISTS .. ===> NO (YES to include more packages)

Enter options as desired:
5 PLAN NAME ................. ===> (Required to create a plan)
6 CHANGE CURRENT DEFAULTS?.. ===> NO (NO or YES)
7 ENABLE/DISABLE CONNECTIONS? ===> NO (NO or YES)
8 OWNER OF PLAN (AUTHID)...... ===> > (Leave blank for your primary ID)
9 QUALIFIER .................... ===> > (For tables, views, and aliases)
10 CACHESIZE ................... ===> 0 (Blank, or value 0-4096)
11 ACTION ON PLAN ............. ===> REPLACE (REPLACE or ADD)
12 RETAIN EXECUTION AUTHORITY. ===> NO (YES to retain user list)
13 CURRENT SERVER ............. ===> (Location name)
14 INCLUDE PATH? ............. ===> NO (NO or YES)

Figure 48. The Bind Plan panel

The following explains the functions on the Bind Plan panel and how to fill the necessary fields in order to bind your program.

1 LOCATION NAME
   Lets you specify the remote system where the package that is named in the PACKAGE ID field is bound. The location name must be defined in the catalog table SYSIBM.LOCATIONS. The default is the local DBMS.

2 COLLECTION ID
   Lets you specify the collection that includes the package that is to be bound into the plan.

   The field is scrollable, and the maximum field length is 128.

3 PACKAGE ID
   Lets you specify the name of the package that is to be bound into the plan.

4 ADDITIONAL PACKAGE LISTS
   Lets you include a list of additional packages in the plan. If you specify YES, a separate panel displays, where you must enter the package location, collection name, and package name for each package to include in the plan. This list is optional.

5 PLAN NAME
   Lets you name the application plan to create. You can specify a name from 1 to 8 characters, and the first character must be alphabetic. If there are no errors, the bind process prepares the plan and enters its description into the EXPLAIN table.

   If you reached this panel through the Db2 Program Preparation panel, the default for this field depends on the value you entered in the INPUT DATA SET NAME field of that panel.

   If you reached this panel directly from the Db2 Primary Option Menu, you must include a plan name if you want to create an application plan. The default name for this field depends on the input data set:

   • If the input data set is partitioned, the default name is the member name.
   • If the input data set is sequential, the default name is the second qualifier of the data set name.

6 CHANGE CURRENT DEFAULTS?
   Lets you specify whether to change the current defaults for binding plans. If you enter YES in this field, you see the Defaults for Bind Plan panel as your next step. You can enter your new preferences there.

7 ENABLE/DISABLE CONNECTIONS?
   Lets you specify whether you want to enable and disable system connections types to use with this package. This is valid only if the LOCATION NAME field names your local Db2 system.

   Placing YES in this field displays a panel (shown in Figure 53 on page 920) that lets you specify whether various system connections are valid for this application. You can specify connection names
to further identify enabled connections within a connection type. A connection name is valid only when you also specify its corresponding connection type.

The default enables all connection types.

8 OWNER OF PLAN (AUTHID)

Lets you specify the primary authorization ID of the owner of the new plan. That ID is the name owning the plan, and the name associated with all accounting and trace records produced by the plan.

The owner must have the privileges required to run SQL statements contained in the plan.

The field is scrollable, and the maximum field length is 128.

9 QUALIFIER

 Lets you specify the default schema for unqualified tables, views, and aliases. You can specify a schema name from 1 to 128 characters, which must conform to the rules for SQL identifiers. If you leave this field blank, the default qualifier is the authorization ID of the plan owner. This field is scrollable.

Lets you specify the default schema for unqualified tables, views, and aliases. You can specify a schema name from 1 to 8 characters, which must conform to the rules for SQL identifiers. If you leave this field blank, the default qualifier is the authorization ID of the plan owner.

10 CACHESIZE

 Lets you specify the size (in bytes) of the authorization cache. Valid values are in the range 0 to 4096. Values that are not multiples of 256 round up to the next highest multiple of 256. A value of 0 indicates that Db2 does not use an authorization cache. The default is 1024.

Each concurrent user of a plan requires 8 bytes of storage, with an additional 32 bytes for overhead.

11 ACTION ON PLAN

Lets you specify whether this is a new or changed application plan. Use:

REPLACE (default) to replace the plan named in the PLAN NAME field if it already exists, and add the plan if it does not exist.

ADD to add the plan named in the PLAN NAME field, only if it does not already exist.

12 RETAIN EXECUTION AUTHORITY

Lets you choose whether or not those users with the authority to bind or run the existing plan are to keep that authority over the changed plan. This applies only when you are replacing an existing plan.

If the plan ownership changes and you specify YES, the new owner grants BIND and EXECUTE authority to the previous plan owner.

If the plan ownership changes and you do not specify YES, then everyone but the new plan owner loses EXECUTE authority (but not BIND authority), and the new plan owner grants BIND authority to the previous plan owner.

13 CURRENT SERVER

Lets you specify the initial server to receive and process SQL statements in this plan. You can specify a name from 1 to 16 characters, which you must previously define in the catalog table SYSIBM.LOCATIONS.

If you specify a remote server, Db2 connects to that server when the first SQL statement executes. The default is the name of the local Db2 subsystem.

14 INCLUDE PATH?

Indicates whether you will supply a list of schema names that Db2 searches when it resolves unqualified distinct type, user-defined function, and stored procedure names in SQL statements. The default is NO. If you specify YES, Db2 displays a panel in which you specify the names of schemas for Db2 to search.

When you finish making changes to this panel, press ENTER to go to the second of the program preparation panels, Program Prep: Compile, Link, and Run.

Related tasks

Caching authorization IDs for plans (Managing Security)
Related reference

Defaults for Bind Plan and Defaults for Rebind Plan panels
These DB2I panels let you change your defaults for BIND PLAN and REBIND PLAN options.

BIND and REBIND options for packages, plans, and services (Db2 Commands)

**Defaults for Bind Package and Defaults for Rebind Package panels**

These DB2I panels lets you change your defaults for BIND PACKAGE and REBIND PACKAGE options.

On the following panel, enter new defaults for binding a package.

```
DSNEBP10                DEFAULTS FOR BIND PACKAGE             SSID: DSN
COMMAND ===> -
----------------- Use the UP/DOWN keys to access all options ------------------
More:     +

Change default options as necessary:
1  ISOLATION LEVEL ........ ===> (CS, RR, RS, UR, or NC)
2  VALIDATION TIME .......... ===> (RUN or BIND)
3  RESOURCE RELEASE TIME ... ===> (COMMIT, DEALLOCATE, or INHERITFROMPLAN)
4  EXPLAIN PATH SELECTION .. ===> (NO or YES)
5  DATA CURRENCY ........... ===> (NO or YES)
6  PARALLEL DEGREE .......... ===> (1 or ANY)
7  SQLERROR PROCESSING ..... ===> (NOPACKAGE or CONTINUE)
8  REOPTIMIZE FOR INPUT VARS ===> (ALWAYS, NONE, ONCE, or AUTO)
9  DEFER PREPARE ........... ===> (NO, YES, or INHERITFROMPLAN)
10 KEEP DYNAMIC SQL PAST COMMIT or ROLLBACK ===> (NO or YES)
11 APPLICATION ENCODING ... ===> (Blank, ASCII, EBCDIC, UNICODE, or ccsid)
12 OPTIMIZATION HINT ...... ===> > (Blank or 'hint-id')
13 IMMEDIATE WRITE ........ ===> (YES, NO, or INHERITFROMPLAN)
14 DYNAMIC RULES ........... ===> (RUN, BIND, DEFINE, or INVOKE)
15 DBPROTOCOL .............. ===> (blank, DRDA, or DRDACBF)
16 ACCESS PATH REUSE ....... ===> NONE (ERROR, NONE, or WARN)
17 ACCESS PATH COMPARISON .. ===> NONE (ERROR, NONE, or WARN)
18 SYSTEM_TIME SENSITIVE ... ===> (blank, NO, or YES)
19 BUSINESS_TIME SENSITIVE ... ===> (blank, NO, or YES)
20 ARCHIVE SENSITIVE ....... ===> (blank, NO, or YES)
21 APPLICATION COMPATIBILITY ===> (blank, DB2 function level, V10R1, or V11R1)

PRESS: ENTER to continue   UP/DOWN to scroll   RETURN to EXIT
```

Figure 49. The Defaults for Bind Package panel

On the following panel, enter new defaults for rebinding a package.

With a few minor exceptions, the options on this panel are the same as the options for the defaults for rebinding a package. However, the defaults for REBIND PACKAGE are different from those shown in the preceding figure, and you can specify SAME in any field to specify the values used the last time the package was bound. For rebinding, the default value for all fields is SAME.
Figure 50. The Defaults for Rebind Package panel

The following table lists the fields on the Defaults for Bind Package and Defaults for Rebind Package panels, and the corresponding bind and rebind options.

Table 149. Defaults for Bind Package and Defaults for Rebind Package panel fields and corresponding bind or rebind options

<table>
<thead>
<tr>
<th>Field name</th>
<th>Bind or rebind option</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACCESS PATH COMPARISON</td>
<td>APCOMPARE</td>
</tr>
<tr>
<td>ACCESS PATH RETAIN DUPS</td>
<td>APRETAINDUP</td>
</tr>
<tr>
<td>ACCESS PATH REUSE</td>
<td>APREUSE</td>
</tr>
<tr>
<td>APPLICATION COMPATIBILITY</td>
<td>APPLCOMPAT</td>
</tr>
<tr>
<td>APPLICATION ENCODING</td>
<td>ENCODING</td>
</tr>
<tr>
<td>ARCHIVE SENSITIVE</td>
<td>ARCHIVESENSITIVE</td>
</tr>
<tr>
<td>BUSINESS_TIME SENSITIVE</td>
<td>BUSTINESENSITIVE</td>
</tr>
<tr>
<td>DATA CURRENCY</td>
<td>CURRENTDATA</td>
</tr>
<tr>
<td>DBPROTOCOL</td>
<td>DBPROTOCOL</td>
</tr>
<tr>
<td>DEFER PREPARE</td>
<td>DEFER and NODEFER</td>
</tr>
<tr>
<td>DYNAMIC RULES</td>
<td>DYNAMICRULES</td>
</tr>
<tr>
<td>EXPLAIN PATH SELECTION</td>
<td>EXPLAIN</td>
</tr>
<tr>
<td>IMMEDIATE WRITE</td>
<td>IMMEDWRITE</td>
</tr>
<tr>
<td>Field name</td>
<td>Bind or rebind option</td>
</tr>
<tr>
<td>------------------------------------------------</td>
<td>-----------------------</td>
</tr>
<tr>
<td>ISOLATION LEVEL</td>
<td>ISOLATION</td>
</tr>
<tr>
<td>KEEP DYNAMIC SQL PAST COMMIT OR ROLLBACK</td>
<td>KEEPDYNAMIC</td>
</tr>
<tr>
<td>OPTIMIZATION HINT</td>
<td>OPTHINT</td>
</tr>
<tr>
<td>PARALLEL DEGREE</td>
<td>DEGREE</td>
</tr>
<tr>
<td>PLAN MANAGEMENT</td>
<td>PLANMGMT</td>
</tr>
<tr>
<td>REOPTIMIZE FOR INPUT VARS</td>
<td>REOPT</td>
</tr>
<tr>
<td>RESOURCE RELEASE TIME</td>
<td>RELEASE</td>
</tr>
<tr>
<td>SQLERROR PROCESSING</td>
<td>SQLERROR</td>
</tr>
<tr>
<td>SYSTEM_TIME SENSITIVE</td>
<td>SYSTIMESSENSITIVE</td>
</tr>
<tr>
<td>VALIDATION TIME and PLAN VALIDATION TIME</td>
<td>VALIDATE</td>
</tr>
</tbody>
</table>

**Related concepts**

Dynamic rules options for dynamic SQL statements

The DYNAMICRULES bind option and the runtime environment determine the rules for the dynamic SQL attributes.

Parallel processing (Db2 Performance)

Investigating SQL performance by using EXPLAIN (Db2 Performance)

**Related tasks**

Setting the isolation level of SQL statements in a REXX program

Isolation levels specify the locking behavior for SQL statements. You can set the isolation level for SQL statements in your REXX program to repeatable read (RR), read stability (RS), cursor stability (CS), or uncommitted read (UR).

**Related reference**

BIND and REBIND options for packages, plans, and services (Db2 Commands)

---

**Defaults for Bind Plan and Defaults for Rebind Plan panels**

These DB2I panels let you change your defaults for BIND PLAN and REBIND PLAN options.

On the following panel, enter new defaults for binding a plan.
Change default options as necessary:

1. **ISOLATION LEVEL** ... ====> **RR** (RR, RS, CS, or UR)
2. **VALIDATION TIME** ... ====> **RUN** (RUN or BIND)
3. **RESOURCE RELEASE TIME** ... ====> **COMMIT** (COMMIT, DEALLOCATE, or INHERITFROMPLAN)
4. **EXPLAIN PATH SELECTION** ... ====> **NO** (NO or YES)
5. **DATA CURRENCY** ... ====> **NO** (NO or YES)
6. **PARALLEL DEGREE** ... ====> **1** (1 or ANY)
7. **RESOURCE ACQUISITION TIME** ====> **USE** (USE or ALLOCATE)
8. **REOPTIMIZE FOR INPUT VARS** ====> **NONE** (ALWAYS, NONE, ONCE, AUTO)
9. **DEFER PREPARE** ... ====> **NO** (NO, YES, INHERITFROMPLAN)
10. **KEEP DYNAMIC SQL**
11. **APPLICATION ENCODING** ... ====> **(Blank, ASCII, EBCDIC, UNICODE, or ccsid)**
12. **OPTIMIZATION HINT** ... ====> **>** (Blank or 'hint-id')
13. **IMMEDIATE WRITE** ... ====> **NO** (NO, YES, INHERITFROMPLAN)
14. **DYNAMIC RULES** ... ====> **RUN** (RUN or BIND)
15. **SQLRULES** ... ====> **DB2** (DB2 or STD)
16. **DISCONNECT** ... ====> **EXPLICIT** (EXPLICIT, AUTOMATIC, or CONDITIONAL)
17. **PROGRAM AUTHORIZATION** ... ====> **DISABLE** (DISABLE, ENABLE)

Figure 51. The Defaults for Bind Plan panel

On the following panel, enter new defaults for rebinding a plan.

Change default options as necessary:

1. **ISOLATION LEVEL** ... ====> **SAME** (SAME, RR, RS, CS, or UR)
2. **PLAN VALIDATION TIME** ... ====> **SAME** (SAME, RUN, or BIND)
3. **RESOURCE RELEASE TIME** ... ====> **SAME** (SAME, DEALLOCATE, COMMIT, or INHERITFROMPLAN)
4. **EXPLAIN PATH SELECTION** ... ====> **SAME** (SAME, NO, or YES)
5. **DATA CURRENCY** ... ====> **SAME** (SAME, NO, or YES)
6. **PARALLEL DEGREE** ... ====> **SAME** (SAME, 1 or ANY)
7. **REOPTIMIZE FOR INPUT VARS** ====> **SAME** (SAME, ALWAYS, NONE, ONCE, AUTO)
8. **DEFER PREPARE** ... ====> **SAME** (SAME, NO, YES, or INHERITFROMPLAN)
9. **KEEP DYNAMIC SQL**
10. **APPLICATION ENCODING** ... ====> **SAME** (SAME, Blank, ASCII, EBCDIC, UNICODE, or ccsid)
11. **OPTIMIZATION HINT** ... ====> **SAME** (SAME, 'hint-id')
12. **IMMEDIATE WRITE** ... ====> **SAME** (SAME, NO, YES, or INHERITFROMPLAN)
13. **SQLRULES** ... ====> **SAME** (SAME, DB2 or STD)
14. **DYNAMIC RULES** ... ====> **SAME** (SAME, RUN, or BIND)
15. **RESOURCE ACQUISITION TIME** ====> **SAME** (SAME, DEALLOCATE, or USE)
16. **DISCONNECT** ... ====> **SAME** (SAME, EXPLICIT, AUTOMATIC, or CONDITIONAL)
17. **PROGRAM AUTHORIZATION** ... ====> **SAME** (SAME, DISABLE, ENABLE)

Figure 52. The Defaults for Rebind Plan panel

The following table lists the fields on the Defaults for Bind Package and Defaults for Rebind Package, and the corresponding bind and rebind options.

<table>
<thead>
<tr>
<th>Field name</th>
<th>Bind or rebind option</th>
</tr>
</thead>
<tbody>
<tr>
<td>APPLICATION ENCODING</td>
<td>ENCODING</td>
</tr>
<tr>
<td>DATA CURRENCY</td>
<td>CURRENTDATA</td>
</tr>
</tbody>
</table>
Table 150. Defaults for Bind Plan and Defaults for Rebind Plan panel fields and corresponding bind or rebind options (continued)

<table>
<thead>
<tr>
<th>Field name</th>
<th>Bind or rebind option</th>
</tr>
</thead>
<tbody>
<tr>
<td>DBPROTOCOL</td>
<td>DBPROTOCOL</td>
</tr>
<tr>
<td>DEFER PREPARE</td>
<td>DEFER and NODEFER</td>
</tr>
<tr>
<td>DISCONNECT</td>
<td>DISCONNECT</td>
</tr>
<tr>
<td>DYNAMIC RULES</td>
<td>DYNAMICRULES</td>
</tr>
<tr>
<td>EXPLAIN PATH SELECTION</td>
<td>EXPLAIN</td>
</tr>
<tr>
<td>IMMEDIATE WRITE</td>
<td>IMMEDWRITE</td>
</tr>
<tr>
<td>ISOLATION LEVEL</td>
<td>ISOLATION</td>
</tr>
<tr>
<td>KEEP DYNAMIC SQL PAST COMMIT OR ROLLBACK</td>
<td>KEEPDYNAMIC</td>
</tr>
<tr>
<td>OPTIMIZATION HINT</td>
<td>OPTHINT</td>
</tr>
<tr>
<td>PARALLEL DEGREE</td>
<td>DEGREE</td>
</tr>
<tr>
<td>PROGRAM AUTHORIZATION</td>
<td>PROGAUTH</td>
</tr>
<tr>
<td>REOPTIMIZE FOR INPUT VARS</td>
<td>REOPT</td>
</tr>
<tr>
<td>RESOURCE ACQUISITION TIME</td>
<td>ACQUIRE</td>
</tr>
<tr>
<td>RESOURCE RELEASE TIME</td>
<td>RELEASE</td>
</tr>
<tr>
<td>VALIDATION TIME and PLAN VALIDATION TIME</td>
<td>VALIDATE</td>
</tr>
</tbody>
</table>

**Related concepts**

Dynamic rules options for dynamic SQL statements
The DYNAMICRULES bind option and the runtime environment determine the rules for the dynamic SQL attributes.

Parallel processing (Db2 Performance)
Investigating SQL performance by using EXPLAIN (Db2 Performance)

**Related tasks**

Caching authorization IDs for plans (Managing Security)
Setting the isolation level of SQL statements in a REXX program
Isolation levels specify the locking behavior for SQL statements. You can set the isolation level for SQL statements in your REXX program to repeatable read (RR), read stability (RS), cursor stability (CS), or uncommitted read (UR).

Specifying the rules that apply to SQL behavior at run time
You can specify whether Db2 rules or SQL standard rules apply to SQL behavior at run time.

**System Connection Types panel**

The System Connection Types panel lets you specify which types of connections can use a plan or package.

This panel displays if you enter YES for ENABLE/DISABLE CONNECTIONS? on the Bind or Rebind Package or Plan panels. For the Bind or Rebind Package panel, the REMOTE option does not display as it does in the following panel.
To enable or disable connection types (that is, allow or prevent the connection from running the package or plan), enter the following information.

1 ENABLE ALL CONNECTION TYPES?
   Lets you enter an asterisk (*) to enable all connections. After that entry, you can ignore the rest of the panel.

2 ENABLE/DISABLE SPECIFIC CONNECTION TYPES
   Lets you specify a list of types to enable or disable; you cannot enable some types and disable others in the same operation. If you list types to enable, enter E; that disables all other connection types. If you list types to disable, enter D; that enables all other connection types.

For each connection type that follows, enter Y (yes) if it is on your list, N (no) if it is not. The connection types are:

- **BATCH** for a TSO connection
- **DB2CALL** for a CAF connection
- **RRSAF** for an RRSAF connection
- **CICS** for a CICS connection
- **IMS** for all IMS connections: DLIBATCH, IMSBMP, and IMSMPP
- **DLIBATCH** for a DL/I Batch Support Facility connection
- **IMSBMP** for an IMS connection to a BMP region
- **IMSMPP** for an IMS connection to an MPP or IFP region
- **REMOTE** for all remote locations or no remote locations

For each connection type that has a second arrow, under SPECIFY CONNECTION NAMES?, enter Y if you want to list specific connection names of that type. Leave N (the default) if you do not. If you use Y in any of those fields, you see another panel on which you can enter the connection names.

If you use the DISPLAY command under TSO on this panel, you can determine what you have currently defined as "enabled" or "disabled" in your ISPF DSNSPFT library (member DSNCONNS). The information does not reflect the current state of the Db2 Catalog.

If you type DISPLAY ENABLED on the command line, you get the connection names that are currently enabled for your TSO connection types. For example:

<table>
<thead>
<tr>
<th>CONNECTION</th>
<th>SUBSYSTEM</th>
</tr>
</thead>
<tbody>
<tr>
<td>CICS1</td>
<td>ENABLED</td>
</tr>
<tr>
<td>CICS2</td>
<td>ENABLED</td>
</tr>
<tr>
<td>CICS3</td>
<td>ENABLED</td>
</tr>
<tr>
<td>CICS4</td>
<td>ENABLED</td>
</tr>
<tr>
<td>DLI1</td>
<td>ENABLED</td>
</tr>
<tr>
<td>DLI2</td>
<td>ENABLED</td>
</tr>
<tr>
<td>DLI3</td>
<td>ENABLED</td>
</tr>
</tbody>
</table>

Figure 53. The System Connection Types panel
Related reference
Panels for entering lists of values
Some fields in DB2I panels are associated with command keywords that accept multiple values. Those fields lead you to a list panel that lets you enter or modify multiple values.

BIND and REBIND options for packages, plans, and services (Db2 Commands)

Panels for entering lists of values
Some fields in DB2I panels are associated with command keywords that accept multiple values. Those fields lead you to a list panel that lets you enter or modify multiple values.

A list panel looks like an ISPF edit session and lets you scroll and use a limited set of commands.

The format of each list panel varies, depending on the content and purpose for the panel. The following figure shows a generic sample of a list panel:

```
panelid         Specific subcommand function
COMMAND ====>
SCROLL ====>
SSID: DSN
```

Subcommand operand values:
```
CMD
**** value _
**** value _
****
****
```

Figure 54. Generic example of a DB2I list panel

All of the list panels let you enter limited commands in two places:

- On the system command line, prefixed by ===>
- In a special command area, identified by """

On the system command line, you can use:

**END**
Saves all entered variables, exits the table, and continues to process.

**CANCEL**
Discards all entered variables, terminates processing, and returns to the previous panel.

**SAVE**
Saves all entered variables and remains in the table.

In the special command area, you can use:

**Inn**
Insert *nn* lines after this one.

**Dnn**
Delete this and the following lines for *nn* lines.

**Rnn**
Repeat this line *nn* number of times.

The default for *nn* is 1.

When you finish with a list panel, specify END to same the current panel values and continue processing.
Program Preparation: Compile, Link, and Run panel

The Compile, Link, and Run panel lets you perform the last two steps in the program preparation process (compile and link-edit). This panel also lets you perform the PL/I MACRO PHASE for programs that require this option.

For TSO programs, the panel also lets you run programs.

1,2 INCLUDE LIBRARY
   Lets you specify up to two libraries containing members for the compiler to include. The members can also be output from DCLGEN. You can leave these fields blank. There is no default.

3 OPTIONS
   Lets you specify compiler, assembler, or PL/I macro processor options. You can also enter a list of compiler or assembler options by separating entries with commas, blanks, or both. You can leave these fields blank. There is no default.

4,5,6 INCLUDE LIBRARY
   Lets you enter the names of up to three libraries containing members for the linkage editor to include. You can leave these fields blank. There is no default.

7 LOAD LIBRARY
   Lets you specify the name of the library to hold the load module. The default value is RUNLIB.LOAD.

   If the load library specified is a PDS, and the input data set is a PDS, the member name specified in INPUT DATA SET NAME field of the Program Preparation panel is the load module name. If the input data set is sequential, the second qualifier of the input data set is the load module name.

   You must complete this field if you request LINK or RUN on the Program Preparation panel.

8 PRELINK OPTIONS
   Lets you enter a list of prelinker options. Separate items in the list with commas, blanks, or both. You can leave this field blank. There is no default.

   The prelink utility applies only to programs using C, C++, and Enterprise COBOL for z/OS.

9 LINK OPTIONS
   Lets you enter a list of link-edit options. Separate items in the list with commas, blanks, or both.

   To prepare a program that uses 31-bit addressing and runs above the 16-megabyte line, specify the following link-edit options: AMODE=31, RMODE=ANY.

10 PARAMETERS
   Lets you specify a list of parameters you want to pass either to your host language run time processor, or to your application. Separate items in the list with commas, blanks, or both. You can leave this field blank.
If you are preparing an IMS or CICS program, you must leave this field blank; you cannot use DB2I to run IMS and CICS programs.

Use a slash (/) to separate the options for your run time processor from those for your program.

- For PL/I and Fortran, run time processor parameters must appear on the left of the slash, and the application parameters must appear on the right.

  \[\text{run time processor parameters} / \text{application parameters}\]

- For COBOL, reverse this order. run time processor parameters must appear on the right of the slash, and the application parameters must appear on the left.

- For assembler and C, there is no supported run time environment, and you need not use a slash to pass parameters to the application program.

11 SYSIN DATA SET

Lets you specify the name of a SYSIN (or in Fortran, FT05F001) data set for your application program, if it needs one. If you do not enclose the data set name in apostrophes, a standard TSO prefix (user ID) and suffix is added to it. The default for this field is TERM.

If you are preparing an IMS or CICS program, you must leave this field blank; you cannot use DB2I to run IMS and CICS programs.

12 SYSPRINT DS

Lets you specify the names of a SYSPRINT (or in Fortran, FT06F001) data set for your application program, if it needs one. If you do not enclose the data set name in apostrophes, a standard TSO prefix (user ID) and suffix is added to it. The default for this field is TERM.

If you are preparing an IMS or CICS program, you must leave this field blank; you cannot use DB2I to run IMS and CICS programs.

Your application could need other data sets besides SYSIN and SYSPRINT. If so, remember to catalog and allocate them before you run your program.

When you press ENTER after entering values in this panel, Db2 compiles and link-edits the application. If you specified in the Db2 Program Preparation panel that you want to run the application, Db2 also runs the application.

Related reference

Prelinking an application (z/OS Language Environment Programming Guide)

### DB2I panels that are used to rebind and free plans and packages

A set of DB2I panels lets you bind, rebind, or free packages.

The following table describes additional panels that you can use to rebind and free packages and plans. It also describes the Run panel, which you can use to run application programs that have already been prepared.

<table>
<thead>
<tr>
<th>Panel</th>
<th>Panel description</th>
</tr>
</thead>
<tbody>
<tr>
<td>“Bind/Rebind/Free Selection panel” on page 924</td>
<td></td>
</tr>
<tr>
<td>“Rebind Package panel” on page 925</td>
<td></td>
</tr>
<tr>
<td>“Rebind Trigger Package panel” on page 927</td>
<td></td>
</tr>
<tr>
<td>“Rebind Plan panel” on page 928</td>
<td></td>
</tr>
</tbody>
</table>

The BIND/REBIND/FREE panel lets you select the BIND, REBIND, or FREE, PLAN, PACKAGE, or TRIGGER PACKAGE process that you need.

The Rebind Package panel lets you change options when you rebind a package.

The Rebind Trigger Package panel lets you change options when you rebind a trigger package.

The Rebind Plan panel lets you change options when you rebind an application plan.
Table 151. DB2I panels used to rebind and free plans and packages and used to Run application programs (continued)

<table>
<thead>
<tr>
<th>Panel</th>
<th>Panel description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;Free Package panel&quot; on page 930</td>
<td>The Free Package panel lets you change options when you free a package.</td>
</tr>
<tr>
<td>&quot;Free Plan panel&quot; on page 931</td>
<td>The Free Plan panel lets you change options when you free an application plan.</td>
</tr>
<tr>
<td>&quot;DB2I Run panel&quot; on page 934</td>
<td>The Run panel lets you start an application program. You should use this panel if you have already prepared the program and you only want to run it. You can also run a program by using the &quot;Program Prep: Compile, Prelink, Link, and Run&quot; panel.</td>
</tr>
</tbody>
</table>

Related reference
DB2I panels that are used for program preparation
DB2I contains a set of panels that let you prepare an application for execution.
The DB2I primary option menu (Introduction to Db2 for z/OS)

Bind/Rebind/Free Selection panel
The Bind/Rebind/Free selection panel lets choose whether to bind, rebind, or free plans and packages.

<table>
<thead>
<tr>
<th>DSNEBP01 COMMAND ==&gt;_</th>
<th>BIND/REBIND/FREE</th>
<th>SSID: DSN</th>
</tr>
</thead>
<tbody>
<tr>
<td>Select one of the following and press ENTER:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1  BIND PLAN</td>
<td>(Add or replace an application plan)</td>
<td></td>
</tr>
<tr>
<td>2  REBIND PLAN</td>
<td>(Rebind existing application plan or plans)</td>
<td></td>
</tr>
<tr>
<td>3  FREE PLAN</td>
<td>(Erase application plan or plans)</td>
<td></td>
</tr>
<tr>
<td>4  BIND PACKAGE</td>
<td>(Add or replace a package)</td>
<td></td>
</tr>
<tr>
<td>5  REBIND PACKAGE</td>
<td>(Rebind existing package or packages)</td>
<td></td>
</tr>
<tr>
<td>6  REBIND TRIGGER PACKAGE</td>
<td>(Rebind existing package or packages)</td>
<td></td>
</tr>
<tr>
<td>7  FREE PACKAGE</td>
<td>(Erase a package or packages)</td>
<td></td>
</tr>
</tbody>
</table>

Figure 56. The Bind/Rebind/Free selection panel
This panel lets you select the process you need.

1 BIND PLAN
Lets you build an application plan. You must have an application plan to allocate Db2 resources and support SQL requests during run time. If you select this option, the Bind Plan panel displays. For more information, see “Bind Plan panel” on page 912.

2 REBIND PLAN
Lets you rebuild an application plan when changes to it affect the plan but the SQL statements in the program are the same. For example, you should rebind when you change authorizations, create a new index that the plan uses, or use RUNSTATS. If you select this option, the Rebind Plan panel displays. For more information, see “Rebind Plan panel” on page 928.

3 FREE PLAN
Lets you delete plans from Db2. If you select this option, the Free Plan panel displays. For more information, see “Free Plan panel” on page 931.
4 BIND PACKAGE
Lets you build a package. If you select this option, the Bind Package panel displays. For more information, see "Bind Package panel" on page 910.

5 REBIND PACKAGE
Lets you rebuild a package when changes to it affect the package but the SQL statements in the program are the same. For example, you should rebind when you change authorizations, create a new index that the package uses, or use RUNSTATS. If you select this option, the Rebind Package panel displays. For more information, see "Rebind Package panel" on page 925.

6 REBIND TRIGGER PACKAGE
Lets you rebuild a trigger package when you need to change options for the package. When you execute CREATE TRIGGER, Db2 binds a trigger package using a set of default options. You can use REBIND TRIGGER PACKAGE to change those options. For example, you can use REBIND TRIGGER PACKAGE to change the isolation level for the trigger package. If you select this option, the Rebind Trigger Package panel displays. For more information, see "Rebind Trigger Package panel" on page 927.

7 FREE PACKAGE
Lets you delete a specific version of a package, all versions of a package, or whole collections of packages from Db2. If you select this option, the Free Package panel displays. For more information, see "Free Package panel" on page 930.

Rebind Package panel
The Rebind Package panel is the first of two panels that you use to rebind a package. This panel lets you specify options for rebinding the package.

The following figure shows the rebind package options.

```
DSNEBP08                  REBIND PACKAGE                      SSID: DSN
COMMAND ==>_
1  Rebind all local packages ===> (* to rebind all packages)
   or
2  Enter package name(s) to be rebound: 
   LOCATION NAME ............. ===>                   (Defaults to local)
   3  COLLECTION-ID ............. ===>                 > (Required)
   4  PACKAGE-ID ................ ===>          >        (Required)
   5  VERSION-ID ................ ===>
      (*, Blank, (), or version-id)
   6  ADDITIONAL PACKAGES? ...... ===>            (Yes to include more packages)
      Enter options as desired ...... ===>
   7  CHANGE CURRENT DEFAULTS?... ===>            (NO or YES)
   8  OWNER OF PACKAGE (AUTHID).. ===>          > (SAME, new OWNER)
   9  QUALIFIER ................. ===>          > (SAME, new QUALIFIER)
   10 ENABLE/DISABLE CONNECTIONS? ===>            (NO or YES)
   11 INCLUDE PATH? ............. ===>            (SAME, DEFAULT, or YES)
```

Figure 57. The Rebind Package panel

This panel lets you choose options for rebinding a package.

1 Rebind all local packages
Lets you rebind all packages on the local DBMS. To do so, place an asterisk (*) in this field; otherwise, leave it blank.

2 LOCATION NAME
Lets you specify where to bind the package. If you specify a location name, you should use from 1 to 16 characters, and you must have defined it in the catalog table SYSIBM.LOCATIONS.

3 COLLECTION-ID
Lets you specify the collection of the package to rebind. You must specify a collection ID from 1 to 128 characters, or an asterisk (*) to rebind all collections in the local Db2 system. You cannot use the asterisk to rebind a remote collection. This field is scrollable.
**4 PACKAGE-ID**

Lets you specify the name of the package to rebind. You must specify a package ID from 1 to 8 characters, or an asterisk (*) to rebind all packages in the specified collections in the local Db2 system. You cannot use the asterisk to rebind a remote package.

The field is scrollable, and the maximum field length is 128.

**5 VERSION-ID**

Lets you specify the version of the package to rebind. You must specify a version ID from 1 to 64 characters, or an asterisk (*) to rebind all versions in the specified collections and packages in the local Db2 system. You cannot use the asterisk to rebind a remote version.

**6 ADDITIONAL PACKAGES?**

Lets you indicate whether to name more packages to rebind. Use YES to specify more packages on an additional panel, described on “Panels for entering lists of values” on page 921. The default is NO.

**7 CHANGE CURRENT DEFAULTS?**

Lets you indicate whether to change the binding defaults. Use:

- **NO (default)** to retain the binding defaults of the previous package.
- **YES** to change the binding defaults from the previous package. For information about the defaults for binding packages, see “Defaults for Bind Package and Defaults for Rebind Package panels” on page 915.

**8 OWNER OF PACKAGE (AUTHID)**

Lets you change the authorization ID for the package owner. The owner must have the required privileges to execute the SQL statements in the package. The default is the existing package owner.

The field is scrollable, and the maximum field length is 128.

**9 QUALIFIER**

Lets you specify the default schema for all unqualified table names, views, indexes, and aliases in the package. You can specify a schema name from 1 to 8 characters, which must conform to the rules for the SQL short identifier. The default is the existing qualifier name.

The field is scrollable, and the maximum field length is 128.

**10 ENABLE/DISABLE CONNECTIONS?**

Lets you specify whether you want to enable and disable system connections types to use with this package. This is valid only if the LOCATION NAME field names your local Db2 system.

Placing YES in this field displays a panel (shown in Figure 53 on page 920) that lets you specify whether various system connections are valid for this application.

The default is the values used for the previous package.

**11 INCLUDE PATH?**

Indicates which one of the following actions you want to perform:

- Request that Db2 uses the same schema names as when the package was bound for resolving unqualified distinct type, user-defined function, and stored procedure names in SQL statements. Choose SAME to perform this action. This is the default.
- Supply a list of schema names that Db2 searches when it resolves unqualified distinct type, user-defined function, and stored procedure names in SQL statements. Choose YES to perform this action.
- Request that Db2 resets the SQL path to SYSIBM, SYSFUN, SYSPROC, and the package owner. Choose DEFAULT to perform this action.

If you specify YES, Db2 displays a panel in which you specify the names of schemas for Db2 to search.

**Related reference**

BIND and REBIND options for packages, plans, and services (Db2 Commands)
Rebind Trigger Package panel

The Rebind Trigger Package panel specifies options for rebinding a trigger package.

The following figure shows those options.

```
DSNEBP19 REBIND TRIGGER PACKAGE SSID: DSN
COMMAND ===> _
  1  Rebind all trigger packages ===> (* to rebind all packages)
    or
  2 LOCATON NAME ........... ===> (Defaults to local)
  3 COLLECTION-ID (SCHEMA NAME) ===> > (Required)
  4 PACKAGE-ID (TRIGGER NAME).. ===> > (Required)
  5 ISOLATION LEVEL ........... ===> SAME (SAME, RR, RS, CS, UR, or NC)
  6 RESOURCE RELEASE TIME ..... ===> SAME (SAME, DEALLOCATE, or COMMIT)
  7 EXPLAIN PATH SELECTION .... ===> SAME (SAME, NO, or YES)
  8 DATA CURRENCY ............. ===> SAME (SAME, NO, or YES)
  9 IMMEDIATE WRITE OPTION ... ===> SAME (SAME, NO, or YES)
 10 PLAN MANAGEMENT .......... ===> DEFAULT (DEFAULT, BASIC, EXTENDED, OFF)
 11 ACCESS PATH REUSE ........ ===> DEFAULT (DEFAULT, ERROR, NONE, or WARN)
 12 ACCESS PATH COMPARISON .... ===> DEFAULT (DEFAULT, ERROR, NONE, or WARN)
 13 ACCESS PATH RETAIN DUPS ... ===> DEFAULT (DEFAULT, NO, or YES)
 14 SYSTEM TIME SENSITIVE ..... ===> SAME (SAME, NO, or YES)
 15 BUSINESS_TIME SENSITIVE ... ===> SAME (SAME, NO, or YES)
 16 ARCHIVE SENSITIVE .......... ===> SAME (SAME, NO, or YES)
 17 APPLICATION COMPATIBILITY . ===> SAME (SAME, DB2 function level V10R1, V11R1)
```

Figure 58. The Rebind Trigger Package panel

This panel lets you choose options for rebinding a trigger package.

1 Rebind all trigger packages
   Lets you rebind all packages on the local DBMS. To do so, place an asterisk (*) in this field; otherwise, leave it blank.

2 LOCATION NAME
   Lets you specify where to bind the trigger package. If you specify a location name, you should use from 1 to 16 characters, and you must have defined it in the catalog table SYSIBM.LOCATIONS.

3 COLLECTION-ID (SCHEMA NAME)
   Lets you specify the collection of the trigger package to rebind. You must specify a collection ID from 1 to 128 characters, or an asterisk (*) to rebind all collections in the local Db2 system. You cannot use the asterisk to rebind a remote collection. This field is scrollable.

4 PACKAGE-ID
   Lets you specify the name of the trigger package to rebind. You must specify a package ID from 1 to 128 characters, or an asterisk (*) to rebind all trigger packages in the specified collections in the local Db2 system. You cannot use the asterisk to rebind a remote trigger package. This field is scrollable.

5 ISOLATION LEVEL
   Lets you specify how far to isolate your application from the effects of other running applications. The default is the value used for the old trigger package.

6 RESOURCE RELEASE TIME
   Lets you specify COMMIT or DEALLOCATE to tell when to release locks on resources. The default is that used for the old trigger package.

7 EXPLAIN PATH SELECTION
   Lets you specify YES or NO for whether to obtain EXPLAIN information about how SQL statements in the package execute. The default is the value used for the old trigger package.

The bind process inserts information into the table owner.PLAN_TABLE, where owner is the authorization ID of the plan or package owner. If you defined owner.DSN_STATEMENT_TABLE, Db2 also inserts information about the cost of statement execution into that table. If you specify YES in
this field and BIND in the VALIDATION TIME field, and if you do not correctly define PLAN_TABLE, the bind fails.

8 DATA CURRENCY

Let you specify YES or NO for whether you need data currency for ambiguous cursors opened at remote locations. The default is the value used for the old trigger package.

Data is current if the data within the host structure is identical to the data within the base table. Data is always current for local processing.

9 IMMEDIATE WRITE OPTION

Specifies when Db2 writes the changes for updated group buffer pool-dependent pages. This field applies only to a data sharing environment. The values that you can specify are:

SAME

Choose the value of IMMEDIATE WRITE that you specified when you bound the trigger package. SAME is the default.

NO

Write the changes at or before phase 1 of the commit process. If the transaction is rolled back later, write the additional changes that are caused by the rollback at the end of the abort process. PH1 is equivalent to NO.

YES

Write the changes immediately after group buffer pool-dependent pages are updated.

10 PLAN MANAGEMENT

Specifies the PLANMGMT option to use for rebinding the trigger. DEFAULT means to take the default setting for this option when rebinding for the old trigger package.

11 ACCESS PATH REUSE

Specifies the APREUSE option to use for rebinding the trigger. DEFAULT means to take the default setting for this option when rebinding for the old trigger package.

12 ACCESS PATH COMPARISON

Specifies the APCOMPARE option to use for rebinding the trigger. DEFAULT means to take the default setting for this option when rebinding for the old trigger package.

13 ACCESS PATH RETAIN DUPS

Specifies the APRETAINDUP option to use for rebinding the trigger. DEFAULT means to take the default setting for this option when rebinding for the old trigger package.

14 SYSTEM_TIME SENSITIVE

Specifies the SYSTIMESSENSITIVE option to use for rebinding the trigger. SAME means to take the previous setting for this option when rebinding the old trigger package.

15 BUSINESS_TIME SENSITIVE

Specifies the BUSTIMESSENSITIVE option to use for rebinding the trigger. SAME means to take the previous setting for this option when rebinding the old trigger package.

16 ARCHIVE SENSITIVE

Specifies the ARCHIVESSENSITIVE option to use for rebinding the trigger. SAME means to take the previous setting for this option when rebinding the old trigger package.

17 APPLICATION COMPATIBILITY

Specifies the APPLCOMPT option to use for rebinding the trigger. SAME means to take the previous setting for this option when rebinding the old trigger package.

Related reference

BIND and REBIND options for packages, plans, and services (Db2 Commands)

Rebind Plan panel

The Rebind Plan panel is the first of two panels that you use to rebind a plan. This panel lets you specify options for rebinding the plan.

The following figure shows the rebind plan options.
Enter plan name(s) to be rebound:
1  PLAN NAME ................. ===>           (* to rebind all plans)
2  ADDITIONAL PLANS? .......... ===> NO        (Yes to include more plans)

Enter options as desired:
3  CHANGE CURRENT DEFAULTS?... ===> NO        (NO or YES)
4  OWNER OF PLAN (AUTHID)..... ===> SAME    > (SAME, new OWNER)
5  QUALIFIER ................. ===> SAME    > (SAME, new QUALIFIER)
6  CACHESIZE ................. ===> SAME      (SAME, or value 0-4096)
7  ENABLE/DISABLE CONNECTIONS? ===> NO        (NO or YES)
8  INCLUDE PACKAGE LIST?...... ===> SAME      (SAME, NO, or YES)
9  CURRENT SERVER ............ ===>                     (Location name)
10 INCLUDE PATH? ............. ===> SAME      (SAME, DEFAULT, or YES)

Figure 59. The Rebind Plan panel

This panel lets you specify options for rebinding your plan.

1 PLAN NAME
Let you name the application plan to rebind. You can specify a name from 1 to 8 characters, and the first character must be alphabetic. Do not begin the name with DSN, because it could create name conflicts with Db2. If there are no errors, the bind process prepares the plan and enters its description into the EXPLAIN table. If you leave this field blank, the bind process occurs but produces no plan.

2 ADDITIONAL PLANS?
Lets you indicate whether to name more plans to rebind. Use YES to specify more plans on an additional panel, described at "Panels for entering lists of values" on page 921. The default is NO.

3 CHANGE CURRENT DEFAULTS?
Lets you indicate whether to change the binding defaults. Use:
- NO (default) to retain the binding defaults of the previous plan.
- YES to change the binding defaults from the previous plan.

4 OWNER OF PLAN (AUTHID)
Lets you change the authorization ID for the plan owner. The owner must have the required privileges to execute the SQL statements in the plan. The default is the existing plan owner.

The field is scrollable, and the maximum field length is 128.

5 QUALIFIER
Lets you specify the default schema for all unqualified table names, views, indexes, and aliases in the plan. You can specify a schema name from 1 to 128 characters, which must conform to the rules for the SQL identifier. The default is the authorization ID. This field is scrollable.

6 CACHESIZE
Lets you specify the size (in bytes) of the authorization cache. Valid values are in the range 0 to 4096. Values that are not multiples of 256 round up to the next highest multiple of 256. A value of 0 indicates that Db2 does not use an authorization cache. The default is the cache size specified for the previous plan.

Each concurrent user of a plan requires 8 bytes of storage, with an additional 32 bytes for overhead.

7 ENABLE/DISABLE CONNECTIONS?
Lets you specify whether you want to enable and disable system connections types to use with this plan. This is valid only for rebinding on your local Db2 system.

Placing YES in this field displays a panel (shown in Figure 53 on page 920) that lets you specify whether various system connections are valid for this application.

The default is the values used for the previous plan.
8 INCLUDE PACKAGE LIST?
Lets you include a list of collections and packages in the plan. If you specify YES, a separate panel displays on which you must enter the package location, collection name, and package name for each package to include in the plan (see “Panels for entering lists of values” on page 921). This field can either add a package list to a plan that did not have one, or replace an existing package list.

You can specify a location name from 1 to 16 characters, a collection ID from 1 to 18 characters, and a package ID from 1 to 8 characters. Separate two or more package list parameters with a comma. If you specify a location name, it must be in the catalog table SYSIBM.LOCATIONS. The default location is the package list used for the previous plan.

9 CURRENT SERVER
Lets you specify the initial server to receive and process SQL statements in this plan. You can specify a name from 1 to 16 characters, which you must previously define in the catalog table SYSIBM.LOCATIONS.

If you specify a remote server, Db2 connects to that server when the first SQL statement executes. The default is the name of the local Db2 subsystem.

10 INCLUDE PATH?
Indicates which one of the following actions you want to perform:
- Request that Db2 uses the same schema names as when the plan was bound for resolving unqualified distinct type, user-defined function, and stored procedure names in SQL statements. Choose SAME to perform this action. This is the default.
- Supply a list of schema names that Db2 searches when it resolves unqualified distinct type, user-defined function, and stored procedure names in SQL statements. Choose YES to perform this action.
- Request that Db2 resets the SQL path to SYSIBM, SYSFUN, SYSPROC, and the plan owner. Choose DEFAULT to perform this action.

If you specify YES, Db2 displays a panel in which you specify the names of schemas for Db2 to search.

Related reference
Defaults for Bind Plan and Defaults for Rebind Plan panels
These DB2I panels let you change your defaults for BIND PLAN and REBIND PLAN options.

BIND and REBIND options for packages, plans, and services (Db2 Commands)

Free Package panel
The DB2I Free Package panel is the first of two panels through which you can specify options for freeing an application package.

The following figure shows the free package options.

```
DSNEBP18                  FREE PACKAGE                        SSID: DSN
COMMAND ====>
1  Free ALL packages ........ ===>     (* to free authorized packages)
   or
2  Enter package name(s) to be freed:  (Defaults to local)
3  LOCATION NAME .............. ===>  > (Required)
4  COLLECTION-ID .............. ===>  >(* to free all packages)
5  PACKAGE-ID ................ ===>     >(* to free all packages)
6  VERSION-ID ................ ===>
5, Blank, (), or version-id)
6 ADDITIONAL PACKAGES? ....... ===>      (Yes to include more packages)
7  PLAN MANAGEMENT SCOPE ...... ===> ALL
   (Yes to include more packages)
```

Figure 60. The Free Package panel
This panel lets you specify options for erasing packages.
1 Free ALL packages
   Lets you free (erase) all packages for which you have authorization or to which you have BINDAGENT
   authority. To do so, place an asterisk (*) in this field; otherwise, leave it blank.

2 LOCATION NAME
   Lets you specify the location name of the DBMS to free the package. You can specify a name from 1 to
   16 characters.

3 COLLECTION-ID
   Lets you specify the collection from which you want to delete packages for which you own or have
   BINDAGENT privileges. You can specify a name from 1 to 128 characters, or an asterisk (*) to free all
   collections in the local Db2 system. You cannot use the asterisk to free a remote collection. This field
   is scrollable.

4 PACKAGE-ID
   Lets you specify the name of the package to free. You can specify a name from 1 to 128 characters, or
   an asterisk (*) to free all packages in the specified collections in the local Db2 system. You cannot use
   the asterisk to free a remote package. The name you specify must be in the Db2 catalog tables. This
   field is scrollable.

5 VERSION-ID
   Lets you specify the version of the package to free. You can specify an identifier from 1 to 64
   characters, or an asterisk (*) to free all versions of the specified collections and packages in the local
   Db2 system. You cannot use the asterisk to free a remote version.

6 ADDITIONAL PACKAGES?
   Lets you indicate whether to name more packages to free. Use YES to specify more packages on an
   additional panel, described in “Panels for entering lists of values” on page 921. The default is NO.

7 PLAN MANAGEMENT SCOPE
   Specifies whether Db2 frees all copies of the package, or only the inactive previous and original
   copies. This value corresponds to the PLANMGMTSCOPE option. The default value is ALL.

Free Plan panel

The DB2I Free Plan panel is the first of two panels through which you can specify options for freeing an
application plan.

Figure 61 on page 931 shows the free plan options.

<table>
<thead>
<tr>
<th>DSNEBP09</th>
<th>FREE PLAN</th>
<th>SSID: DSN</th>
</tr>
</thead>
<tbody>
<tr>
<td>COMMAND</td>
<td>==&gt;_</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Enter plan name(s) to be freed:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 PLAN NAME ............... ==&gt;_</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2 ADDITIONAL PLANS? .... ==&gt;_</td>
<td>(* to free all authorized plans)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>(Yes to include more plans)</td>
<td></td>
</tr>
</tbody>
</table>

Figure 61. The Free Plan panel

This panel lets you specify options for freeing plans.

1 PLAN NAME
   Lets you name the application plan to delete from Db2. Use an asterisk to free all plans for which you
   have BIND authority. You can specify a name from 1 to 8 characters, and the first character must be
   alphabetic.

   If there are errors, the free process terminates for that plan and continues with the next plan.

2 ADDITIONAL PLANS?
   Lets you indicate whether to name more plans to free. Use YES to specify more plans on an additional
   panel, described in “Panels for entering lists of values” on page 921. The default is NO.
Chapter 10. Running an application on Db2 for z/OS

You can run your application after you have processed the SQL statements, compiled and link-edited the application, and bound the application.

About this task
At run time, Db2 verifies that the information in the application plan and its associated packages is consistent with the corresponding information in the Db2 catalog. If any destructive changes, such as DROP or REVOKE, occur (either to the data structures that your application accesses or to the binder's authority to access those data structures), Db2 automatically rebinds packages or the plan as needed.

Establishing a test environment: This topic describes how to design a test data structure and how to fill tables with test data.

CICS
Before you run an application, ensure that the following two conditions are met:
- The corresponding entries in the SNT and RACF control areas authorize your application to run.
- The program and its transaction code are defined in the CICS CSD.

The system administrator is responsible for these functions.

DSN command processor
The DSN command processor is a TSO command processor that runs in TSO foreground or under TSO in a JES-initiated batch environment.

It uses the TSO attachment facility to access Db2. The DSN command processor provides an alternative method for running programs that access Db2 in a TSO environment.

When you run an application by using the DSN command processor, that application can run in a trusted connection if Db2 finds a matching trusted context.

You can use the DSN command processor implicitly during program development for functions such as:
- Using the declarations generator (DCLGEN)
- Running the BIND, REBIND, and FREE subcommands on Db2 plans and packages for your program
- Using SPUFI (SQL Processor Using File Input) to test some of the SQL functions in the program

The DSN command processor runs with the TSO terminal monitor program (TMP). Because the TMP runs in either foreground or background, DSN applications run interactively or as batch jobs.

The DSN command processor can provide these services to a program that runs under it:
- Automatic connection to Db2
- Attention key support
- Translation of return codes into error messages

Limitations of the DSN command processor
When using DSN services, your application runs under the control of DSN. Because TSO executes the ATTACH macro to start DSN, and DSN executes the ATTACH macro to start a part of itself, your application gains control that is two task levels below TSO.

Because your program depends on DSN to manage your connection to Db2:
- If Db2 is down, your application cannot begin to run.
- If Db2 terminates, your application also terminates.
- An application can use only one plan.
If these limitations are too severe, consider having your application use the call attachment facility or Resource Recovery Services attachment facility. For more information about these attachment facilities, see “Call attachment facility” on page 37 and “Resource Recovery Services attachment facility” on page 67.

**DSN return code processing**

At the end of a DSN session, register 15 contains the highest value that is placed there by any DSN subcommand that is used in the session or by any program that is run by the RUN subcommand. Your run time environment might format that value as a return code. However, the value does not originate in DSN.

**Related concepts**

TSO attachment facility (Introduction to Db2 for z/OS)

**Related reference**

DSN (TSO) (Db2 Commands)

**Related information**

Command types and environments in Db2 (Db2 Commands)

---

**DB2I Run panel**

The DB2I Run panel lets you start an application program that can contain SQL statements.

You can reach the Run panel only through the DB2I Primary Options Menu. You can accomplish the same task using the "Program Preparation: Compile, Link, and Run" panel. You should use this panel if you have already prepared the program and simply want to run it. Figure 62 on page 934 shows the run options.

```
  DSNERPO1                        RUN                        SSID: DSN
  COMMAND ====> _

  Enter the name of the program you want to run:
  1  DATA SET NAME ====>
  2  PASSWORD..... ====> (Required if data set is password protected)

  Enter the following as desired:
  3  PARAMETERS .. ====> (Required if different from program name)
  5  WHERE TO RUN ===> (FOREGROUND, BACKGROUND, or EDITJCL)
```

*Figure 62. The Run panel*

This panel lets you run existing application programs.

**1 DATA SET NAME**

Lets you specify the name of the partitioned data set that contains the load module. If the module is in a data set that the operating system can find, you can specify the member name only. There is no default.

If you do not enclose the name in apostrophes, a standard TSO prefix (user ID) and suffix (.LOAD) is added.

**2 PASSWORD**

Lets you specify the data set password if needed. The RUN processor does not check whether you need a password. If you do not enter a required password, your program does not run.

**3 PARAMETERS**

Lets you specify a list of parameters you want to pass either to your host language run time processor, or to your application. You should separate items in the list with commas, blanks, or both. You can leave this field blank.

Use a slash (/) to separate the options for your run time processor from those for your program.

- For PL/I and Fortran, run time processor parameters must appear on the left of the slash, and the application parameters must appear on the right.
run time processor parameters / application parameters

- For COBOL, reverse this order. run time processor parameters must appear on the right of the slash, and the application parameters must appear on the left.
- For assembler and C, there is no supported run time environment, and you need not use the slash to pass parameters to the application program.

4 PLAN NAME
Lets you specify the name of the plan to which the program is bound. The default is the member name of the program.

5 WHERE TO RUN
Lets you indicate whether to run in the foreground or background. You can also specify EDITJCL, in which case you are able to edit the job control statement before you run the program. Use:
- FOREGROUND to immediately run the program in the foreground with the specified values.
- BACKGROUND to create and immediately submit to run a file containing a DSNH CLIST using the JOB control statement from either DB2I Defaults Panel 2 or your site's SUBMIT exit. The program runs in the background.
- EDITJCL to create and open a file containing a DSNH CLIST in edit mode. You can then submit the CLIST or save it. The program runs in the background.

Running command processors
To run a command processor (CP), use the following commands from the TSO ready prompt or as a TSO TMP:

```shell
DSN SYSTEM (Db2-subsystem-name)
RUN CP PLAN (plan-name)
```

The RUN subcommand prompts you for more input. The end the DSN processor, use the END command.

Running a program in TSO foreground

Use the DB2I RUN panel to run a program in TSO foreground. Alternatively, you can issue the DSN command, followed by the RUN subcommand of DSN.

About this task
Before running the program, be sure to allocate any data sets that your program needs.

The following example shows how to start a TSO foreground application. The name of the application is SAMPPGM, and ssid is the system ID:

```shell
TSO Prompt:    READY
Enter: DSN SYSTEM(ssid)
DSN Prompt:    DSN
Enter: RUN PROGRAM(SAMPPGM) - PLAN(SAMPLAN) - LIB(SAMPPROJ.SAMPLIB) - PARMS('/D01 D02 D03')
: (Here the program runs and might prompt you for input)
DSN Prompt:    DSN
Enter: END
TSO Prompt:    READY
```

This sequence also works in ISPF option 6. You can package this sequence in a CLIST. Db2 does not support access to multiple Db2 subsystems from a single address space.

The PARMS keyword of the RUN subcommand enables you to pass parameters to the run time processor and to your application program:

```shell
PARMS ('/D01, D02, D03')
```
The slash (/) indicates that you are passing parameters. For some languages, you pass parameters and run time options in the form PARMS('parameters/run time-options'). An example of the PARMS keyword might be:

```
PARMS('D01, D02, D03/')
```

Check your host language publications for the correct form of the PARMS option.

**Running a Db2 REXX application**

You run Db2 REXX applications under TSO. You do not precompile, compile, link-edit, or bind Db2 REXX applications before you run them.

**About this task**

In a batch environment, you might use statements like these to invoke application REXXPROG:

```
//RUNREXX EXEC PGM=IKJEFT01,DYNAMNBR=20
//SYSEXEC DD DISP=SHR,DSN=SYSADM.REXX.EXEC
//SYSTSPRT DD SYSOUT=*%REXXPROG parameters
```

The SYSEXEC data set contains your REXX application, and the SYSTSIN data set contains the command that you use to invoke the application.

**Invoking programs through the Interactive System Productivity Facility**

You can use ISPF to invoke programs that connect to Db2 through the call attachment facility (CAF).

**About this task**

The ISPF/CAF sample connection manager programs (DSN8SPM and DSN8SCM) take advantage of the ISPLINK SELECT services, letting each routine make its own connection to Db2 and establish its own thread and plan.

With the same modular structure as in the previous example, using CAF is likely to provide greater efficiency by reducing the number of CLISTs. This does not mean, however, that any Db2 function executes more quickly.

**Disadvantages:** Compared to the modular structure using DSN, the structure using CAF is likely to require a more complex program, which in turn might require assembler language subroutines. For more information, see “Call attachment facility” on page 37.

**ISPF**

The Interactive System Productivity Facility (ISPF) helps you to construct and execute dialogs. Db2 includes a sample application that illustrates how to use ISPF through the call attachment facility (CAF).

Each scenario has advantages and disadvantages in terms of efficiency, ease of coding, ease of maintenance, and overall flexibility.

**Using ISPF and the DSN command processor**

There are some restrictions on how you make and break connections to Db2 in any structure. If you use the PGM option of ISPF SELECT, ISPF passes control to your load module by the LINK macro; if you use CMD, ISPF passes control by the ATTACH macro.

The DSN command processor permits only single task control block (TCB) connections. Take care not to change the TCB after the first SQL statement. ISPF SELECT services change the TCB if you started DSN
under ISPF, so you cannot use these to pass control from load module to load module. Instead, use LINK, XCTL, or LOAD.

The following figure shows the task control blocks that result from attaching the DSN command processor below TSO or ISPF.

![Figure 63. DSN task structure](image)

**Notes:**

1. The RUN command with the CP option causes DSN to attach your program and create a new TCB.
2. The RUN command without the CP option causes DSN to link to your program.

If you are in ISPF and running under DSN, you can perform an ISPLINK to another program, which calls a CLIST. In turn, the CLIST uses DSN and another application. Each such use of DSN creates a separate unit of recovery (process or transaction) in Db2.

All such initiated DSN work units are unrelated, with regard to isolation (locking) and recovery (commit). It is possible to deadlock with yourself; that is, one unit (DSN) can request a serialized resource (a data page, for example) that another unit (DSN) holds incompatibly.

A COMMIT in one program applies only to that process. There is no facility for coordinating the processes.

**Related concepts**

- Dynamic SQL and the ISPF/CAF application (Db2 Installation and Migration)
- Printing options for the sample application listings (Db2 Installation and Migration)
- Sample applications supplied with Db2 for z/OS

Db2 provides sample applications to help you with Db2 programming techniques and coding practices within each of the four environments: batch, TSO, IMS, and CICS. The sample applications contain various applications that might apply to managing a company.

**DSN command processor**
The DSN command processor is a TSO command processor that runs in TSO foreground or under TSO in a JES-initiated batch environment.

**Invoking a single SQL program through ISPF and DSN**

When you invoke a single SQL program through ISPF and DSN, you should first invoke ISPF, which displays the data and selection panels. When you select the program on the selection panel, ISPF calls a CLIST that runs the program.

**About this task**

A corresponding CLIST might contain:

```
DSN
RUN PROGRAM(MYPROG) PLAN(MYPLAN)
END
```

The application has one large load module and one plan.

**Disadvantages:** For large programs of this type, you want a more modular design, making the plan more flexible and easier to maintain. If you have one large plan, you must rebind the entire plan whenever you change a module that includes SQL statements. To achieve a more modular construction when all parts of the program use SQL, consider using packages. See “Db2 program preparation overview” on page 893. You cannot pass control to another load module that makes SQL calls by using ISPLINK; rather, you must use LINK, XCTL, or LOAD and BALR.

If you want to use ISPLINK, then call ISPF to run under DSN:

```
DSN
RUN PROGRAM(ISPF) PLAN(MYPLAN)
END
```

You then need to leave ISPF before you can start your application.

Furthermore, the entire program is dependent on Db2; if Db2 is not running, no part of the program can begin or continue to run.

**Invoking multiple SQL programs through ISPF and DSN**

You can break a large application into several different functions. Each function communicates through a common pool of shared variables, which is controlled by ISPF.

**About this task**

You might write some functions as separately compiled and loaded programs, others as EXECs or CLISTs. You can start any of those programs or functions through the ISPF SELECT service, and you can start that from a program, a CLIST, or an ISPF selection panel.

When you use the ISPF SELECT service, you can specify whether ISPF should create a new ISPF variable pool before calling the function. You can also break a large application into several independent parts, each with its own ISPF variable pool.

You can call different parts of the program in different ways. For example, you can use the PGM option of ISPF SELECT:

```
PGM(program-name) PARM(parameters)
```

Alternatively, you can use the CMD option:

```
CMD(command)
```
For a part that accesses Db2, the command can name a CLIST that starts DSN:

```
DSN
RUN PROGRAM(PART1) PLAN(PLAN1) PARM(input from panel)
END
```

Breaking the application into separate modules makes it more flexible and easier to maintain. Furthermore, some of the application might be independent of Db2; portions of the application that do not call Db2 can run, even if Db2 is not running. A stopped Db2 database does not interfere with parts of the program that refer only to other databases.

**Disadvantages:** The modular application, on the whole, has to do more work. It calls several CLISTs, and each one must be located, loaded, parsed, interpreted, and executed. It also makes and breaks connections to Db2 more often than the single load module. As a result, you might lose some efficiency.

### Loading and running a batch program

You can run a DL/I batch program by running module DSNMTVO1, which loads your application, or by running the application program directly.

#### Procedure

To run a program using Db2, you need a Db2 plan. The bind process creates the Db2 plan. Db2 first verifies whether the DL/I batch job step can connect to Db2. Then Db2 verifies whether the application program can access Db2 and enforce user identification of batch jobs accessing Db2.

The two ways to submit DL/I batch applications to Db2 are:

- DSNMTVO1 can be specified as the application program name for the batch region. When this method is used, control is given to DSNMTVO1. When the Db2 environment is established, control is passed to the application program.

- The application program name can be specified for the batch region. Control is given to DSNMTVO1 directly to establish the external subsystem environment for Db2. When the Db2 environment is established, control is passed to the application program that is specified in the batch region. To accomplish this, in the batch region startup procedure in your application JCL, specify the following information:
  - MBR=application-name
  - SSM=Db2-subsystem-name

#### Examples

**Example: Submitting a DL/I batch application without using DSNMTVO1**

The skeleton JCL in the following example illustrates a COBOL application program, IVP8CP22, that runs using Db2 DL/I batch support.

```bash
//TEPCTEST JOB 'USER=ADMF001',MSGCLASS=A,MSGLEVEL=(1,1),
//          TIME=1440,CLASS=A,USER=SYSADM,PASSWORD=SYSADM
//*******************************
//BATCH EXEC DLIBATCH,PSB=IVP8CA,MBR=IVP8CP22,
//       BKO=Y,DBRC=N,IRLM=N,SSM=SSDQ
//*******************************
//SYSPRINT DD SYSOUT=A
//REPORT   DD SYSOUT=*  
```

Chapter 10. Running an application on Db2 for z/OS  939
Example: Submitting a DL/I batch application using DSNMTV01

The following skeleton JCL example illustrates a COBOL application program, IVP8CP22, that runs using Db2 DL/I batch support.

- The first step uses the standard DLIBATCH IMS procedure.
- The second step shows how to use the DFSERA10 IMS program to print the contents of the DDOTV02 output data set.

Related concepts

Input and output data sets for DL/I batch jobs

DL/I batch jobs require an input data set with DD name DDITV02 and an output data set with DD name DDOTV02.

Authorization for running a batch DL/I program

When a DL/I batch application tries to run the first SQL statement, Db2 checks whether the authorization ID has the EXECUTE privilege for the plan. Db2 uses the same ID for subsequent authorization checks and also identifies records from the accounting and performance traces.

The primary authorization ID is the value of the USER parameter on the job statement, if that is available. If that parameter is not available, the primary authorization ID is the TSO logon name if the job is submitted. Otherwise, the primary authorization ID is the IMS PSB name. In that case, however, the ID must not begin with the string "SYSADM" because this string causes the job to abnormally terminate. The batch job is rejected if you try to change the authorization ID in an exit routine.
Restarting a batch program

To restart a batch program that updates data, first run the IMS Batch Backout utility, followed by a restart job indicating the last successful checkpoint ID.

About this task

For guidelines on finding the last successful checkpoint, see “Finding the DL/I batch checkpoint ID” on page 942.

Examples

Example: Batch checkout with JCL

The skeleton JCL example that follows illustrates a batch backout for PSB=IVP8CA.

```
//ISOCS04  JOB 3000,ISOIR,MSGLEVEL=(1,1),NOTIFY=ISOIR,
   MSGCLASS=T,CLASS=A
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
// BACKOUT TO LAST CHKPT.                                      *
//   IF RC=0028 LOG WITH NO-UPDATE                    *
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
// BACKOUT  EXEC PGM=DFSRRC00,
//          PARM='DLI,DFSBBO00,IVP8CA,,,,,,,,,,,Y,N,,Y',
//           REGION=2600K,COND=EVEN             ---> DBRC ON
//STEPLIB DD  DSN=IMS.RESLIB,DISP=SHR
//IMS       DD   DSN=IMS.PSBLIB,DISP=SHR
//         DD   DSN=IMS.DBDLIB,DISP=SHR
//         DD   DSN=IMS.LOGYLIB,DISP=SHR
//         DD   DSN=IMS.RESLIB,DISP=SHR
//         DD   DSN=IMS.RESLIB,DISP=SHR
//         DD   DSN=IMS.RESLIB,DISP=SHR
//         DD   DSN=IMS.RESLIB,DISP=SHR
//         DD   DSN=IMS.RESLIB,DISP=SHR
//         DD   DSN=IMS.RESLIB,DISP=SHR
//         DD   DSN=IMS.RESLIB,DISP=SHR
//         DD   DSN=IMS.RESLIB,DISP=SHR
//         DD   DSN=IMS.RESLIB,DISP=SHR
//         DD   DSN=IMS.RESLIB,DISP=SHR
//         DD   DSN=IMS.RESLIB,DISP=SHR
//         DD   DSN=IMS.RESLIB,DISP=SHR
//         DD   DSN=IMS.RESLIB,DISP=SHR
//         DD   DSN=IMS.RESLIB,DISP=SHR
//         DD   DSN=IMS.RESLIB,DISP=SHR
//         DD   DSN=IMS.RESLIB,DISP=SHR
//         DD   DSN=IMS.RESLIB,DISP=SHR
//         DD   DSN=IMS.RESLIB,DISP=SHR
//         DD   DSN=IMS.RESLIB,DISP=SHR
//         DD   DSN=IMS.RESLIB,DISP=SHR
//         DD   DSN=IMS.RESLIB,DISP=SHR
//         DD   DSN=IMS.RESLIB,DISP=SHR
//         DD   DSN=IMS.RESLIB,DISP=SHR
//         DD   DSN=IMS.RESLIB,DISP=SHR
//         DD   DSN=IMS.RESLIB,DISP=SHR
//         DD   DSN=IMS.RESLIB,DISP=SHR
//         DD   DSN=IMS.RESLIB,DISP=SHR
//         DD   DSN=IMS.RESLIB,DISP=SHR
//         DD   DSN=IMS.RESLIB,DISP=SHR
//         DD   DSN=IMS.RESLIB,DISP=SHR
//         DD   DSN=IMS.RESLIB,DISP=SHR
//         DD   DSN=IMS.RESLIB,DISP=SHR
//         DD   DSN=IMS.RESLIB,DISP=SHR
//         DD   DSN=IMS.RESLIB,DISP=SHR
//         DD   DSN=IMS.RESLIB,DISP=SHR
//         DD   DSN=IMS.RESLIB,DISP=SHR
//         DD   DSN=IMS.RESLIB,DISP=SHR
//         DD   DSN=IMS.RESLIB,DISP=SHR
//         DD   DSN=IMS.RESLIB,DISP=SHR
//         DD   DSN=IMS.RESLIB,DISP=SHR
//         DD   DSN=IMS.RESLIB,DISP=SHR
//         DD   DSN=IMS.RESLIB,DISP=SHR
//         DD   DSN=IMS.RESLIB,DISP=SHR
//         DD   DSN=IMS.RESLIB,DISP=SHR
//         DD   DSN=IMS.RESLIB,DISP=SHR
//         DD   DSN=IMS.RESLIB,DISP=SHR
//         DD   DSN=IMS.RESLIB,DISP=SHR
//         DD   DSN=IMS.RESLIB,DISP=SHR
//         DD   DSN=IMS.RESLIB,DISP=SHR
//         DD   DSN=IMS.RESLIB,DISP=SHR
//         DD   DSN=IMS.RESLIB,DISP=SHR
//         DD   DSN=IMS.RESLIB,DISP=SHR
//         DD   DSN=IMS.RESLIB,DISP=SHR
//         DD   DSN=IMS.RESLIB,DISP=SHR
//         DD   DSN=IMS.RESLIB,DISP=SHR
//         DD   DSN=IMS.RESLIB,DISP=SHR
//         DD   DSN=IMS.RESLIB,DISP=SHR
//         DD   DSN=IMS.RESLIB,DISP=SHR
//         DD   DSN=IMS.RESLIB,DISP=SHR
//         DD   DSN=IMS.RESLIB,DISP=SHR
//         DD   DSN=IMS.RESLIB,DISP=SHR
//         DD   DSN=IMS.RESLIB,DISP=SHR
//         DD   DSN=IMS.RESLIB,DISP=SHR
//         DD   DSN=IMS.RESLIB,DISP=SHR
//         DD   DSN=IMS.RESLIB,DISP=SHR
//         DD   DSN=IMS.RESLIB,DISP=SHR
//         DD   DSN=IMS.RESLIB,DISP=SHR

```

Example: restarting a DL/I batch job with JCL

Operational procedures can restart a DL/I batch job step for an application program using IMS XRST and symbolic CHKP calls.

You cannot restart a BMP application program in a Db2 DL/I batch environment. The symbolic checkpoint records are not accessed, causing an IMS user abend U0102.

To restart a batch job that terminated abnormally or prematurely, find the checkpoint ID for the job on the z/OS system log or from the SYSOUT listing of the failing job. Before you restart the job step, place the checkpoint ID in the CKPTID=value option of the DLIBATCH procedure, submit the job. If the default connection name is used (that is, you did not specify the connection name option in the DDITV02 input data set), the job name of the restart job must be the same as the failing job. Refer to the following skeleton example, in which the last checkpoint ID value was IVP80002:

```
//ISOCS04  JOB 3000,OJALA,MSGLEVEL=(1,1),NOTIFY=OJALA,
   MSGCLASS=T,CLASS=A
// ** ************** The following step restarts COBOL program IVP8CP22, **
//   which updates ** both DB2 and DL/I databases, from CKPTID=IVP80002. **
// ** ************** **
// RSTRT  EXEC DLIBATCH,DBRC=Y,COND=EVEN,LOGT=SYSDA,
//       MBR=DSNMTV01,PSB=IVP8CA,BKO=Y,IRLM=N,CKPTID=IVP80002
//         G.STEPLIB DD
//         DD   DSN=prefix.SDSNLOAD,DISP=SHR
//         DD   DSN=prefix.RUNLIB.LOAD,DISP=SHR
//         DD   DSN=SYS1.COB2LIB,DISP=SHR
//         DD   other program libraries
```
Finding the DL/I batch checkpoint ID

When an application program issues an IMS CHKP call, IMS sends the checkpoint ID to the z/OS console and the SYSOUT listing in message DFS0540I.

About this task

IMS also records the checkpoint ID in the type X’41’ IMS log record. Symbolic CHKP calls also create one or more type X’18’ records on the IMS log. XRST uses the type X’18’ log records to reposition DL/I databases and return information to the application program.

During the commit process the application program checkpoint ID is passed to Db2. If a failure occurs during the commit process, creating an indoubt work unit, Db2 remembers the checkpoint ID. You can use the following techniques to find the last checkpoint ID:

- Look at the SYSOUT listing for the job step to find message DFS0540I, which contains the checkpoint IDs that are issued. Use the last listed checkpoint ID.
- Look at the z/OS console log to find message DFS0540I that contains the checkpoint ID that is issued for this batch program. Use the last listed checkpoint ID.
- Submit the IMS Batch Backout utility to back out the DL/I databases to the last (default) checkpoint ID. When the batch backout finishes, message DFS395I provides the last valid IMS checkpoint ID. Use this checkpoint ID on restart.
- When restarting Db2, issue the command -DISPLAY THREAD(*) TYPE(INDOUBT) to obtain a possible indoubt unit of work (connection name and checkpoint ID). If you restarted the application program from this checkpoint ID, the program could work because the checkpoint is recorded on the IMS log; however, the program could fail with an IMS user abend U102 because IMS did not finish logging the information before the failure. In that case, restart the application program from the previous checkpoint ID.

Db2 performs one of two actions automatically when restarted, if the failure occurs outside the indoubt period: it either backs out the work unit to the prior checkpoint, or it commits the data without any assistance. If the operator then issues the following command, no work unit information is displayed:

- DISPLAY THREAD(*) TYPE(INDOUBT)

Running stored procedures from the command line processor

As an alternative to calling a stored procedure from an application program, you can use the command line processor to invoke stored procedures.

Procedure

To run a stored procedure from the command line processor:
1. Invoke the command line processor and connect to the appropriate Db2 subsystem. For more information about how to perform these tasks, see The Db2 command line processor (Db2 Commands).

2. Specify the CALL statement in the form that is acceptable for the command line processor.

Related tasks
Calling a stored procedure from your application
To run a stored procedure, you can either call it from a client program or invoke it from the command line processor.

Implementing Db2 stored procedures (Db2 Administration Guide)

Command line processor CALL statement

Use the command line processor CALL statement to invoke stored procedures from the command line processor.

Use the following syntax for the command line processor CALL statement.

```
CALL procedure-name (parameter1, parameter2, parameter3, parameter4)
```

Notes:

1. If you specify an unqualified stored procedure name, Db2 searches the schema list in the CURRENT PATH special register. Db2 searches this list for a stored procedure with the specified number of input and output parameters.
2. Specify a question mark (?) as a placeholder for each output parameter.
3. For non-numeric, BLOB, or CLOB input parameters, enclose each value in single quotation marks ('). The exception is if the data is a BLOB or CLOB value that is to be read from a file. In that case, use the notation `file://fully qualified file name`.
4. Specify the input and output parameters in the order that they are specified in the signature for the stored procedure.

Examples

Example 1
Assume that the TEST.DEPT_MEDIAN stored procedure was created with the following statement:

```
CREATE PROCEDURE TEST.DEPT_MEDIAN
  (IN DEPTNUMBER SMALLINT, OUT MEDIANSALARY INT)
```

To invoke the stored procedure from the command line processor, you can specify the following CALL statement:

```
CALL TEST.DEPT_MEDIAN(51, ?)
```

Assume that the stored procedure returns a value of 25,000. The following information is displayed by the command line processor:

```
Value of output parameters
--------------------------
Parameter Name : MEDIANSALARY
Parameter Value : 25000
```

Example 2
Suppose that stored procedure TEST.BLOBSP is defined with one input parameter of type BLOB and one output parameter. You can invoke this stored procedure from the command line processor with the following statement:
CALL TEST.BLOBSP(file:///tmp/photo.bmp,?)

The command line processor reads the contents from /tmp/photo.bmp as the input parameter. Alternatively, you can invoke this stored procedure by specifying the input parameter in the CALL statement itself, as in the following example:

CALL TEST.BLOBSP('abcdef',?)

**Example of running a batch Db2 application in TSO**

Most application programs that are written for the batch environment run under the TSO Terminal Monitor Program (TMP) in background mode.

The following figure shows the JCL statements that you need in order to start such a job. The list that follows explains each statement.

```
//jobname  JOB USER=MY DB2ID
//GO EXEC PGM=IKJEFT01,DYNAMNBR=20
//STEPLIB DD DSN=prefix.SDSNEXIT,DISP=SHR
//        DD DSN=prefix.SDSNLOAD,DISP=SHR
⋮
//SYSTSPRT DD SYSOUT=A
//SYSTSIN DD *
DSN SYSTEM (ssid)
RUN PROG (SAMPPGM) -
   PLAN (SAMPLAN) -
   LIB (SAMPPROJ.SAMPLIB) -
   PARMS ('/D01 D02 D03')
END
/*
```

- The **JOB** option identifies this as a job card. The **USER** option specifies the Db2 authorization ID of the user.
- The **EXEC** statement calls the TSO Terminal Monitor Program (TMP).
- The **STEPLIB** statement specifies the library in which the DSN Command Processor load modules and DSNHDECP or a user-specified application defaults module reside. It can also reference the libraries in which user applications, exit routines, and the customized DSNHDECP module reside. The customized DSNHDECP module is created during installation.
- Subsequent DD statements define additional files that are needed by your program.
- The **DSN** command connects the application to a particular Db2 subsystem.
- The **RUN** subcommand specifies the name of the application program to run.
- The **PLAN** keyword specifies plan name.
- The **LIB** keyword specifies the library that the application should access.
- The **PARMS** keyword passes parameters to the run time processor and the application program.
- **END** ends the DSN command processor.

**Usage notes**

- Keep DSN job steps short.
- **Recommendation:** Do not use DSN to call the EXEC command processor to run CLISTs that contain ISPEXEC statements; results are unpredictable.
- If your program abends or gives you a non-zero return code, DSN terminates.
- You can use a group attachment or subgroup attachment name instead of a specific ssid to connect to a member of a data sharing group.

**Related tasks**

Running TSO application programs (Db2 Administration Guide)
Example of calling applications in a command procedure

As an alternative to foreground or batch calls to an application, you can run a TSO or batch application by using a command procedure (CLIST).

The following CLIST calls a Db2 application program named MYPROG. ssid represents the Db2 subsystem name, or group attachment or subgroup attachment name.

```clist
PROC 0                        /* INVOCATION OF DSN FROM A CLIST      */
DSN SYSTEM(ssid)            /* INVOKE DB2 SUBSYSTEM ssid          */
IF &LASTCC = 0 THEN         /* BE SURE DSN COMMAND WAS SUCCESSFUL  */
   DATA                    /* IF SO THEN DO DSN RUN SUBCOMMAND    */
   RUN PROGRAM(MYPROG)    /* ELSE OMIT THE FOLLOWING:              */
   ENDDATA                 /* THE RUN AND THE END ARE FOR DSN     */
END                        /* THE END OF A COMMAND PROCEDURE       */
EXIT
```

IMS: To run a message-driven program

First, ensure that you can respond to the program's interactive requests for data and that you can recognize the expected results. Then, enter the transaction code that is associated with the program. Users of the transaction code must be authorized to run the program.

To run a non-message-driven program

CICSTo run a program

First, ensure that the corresponding entries in the SNT and RACF control areas allow run authorization for your application. The system administrator is responsible for these functions.

Submit the job control statements that are needed to run the program.

Also, be sure to define to CICS the transaction code that is assigned to your program and the program itself.

Make a new copy of the program

Issue the NEWCOPY command if CICS has not been reinitialized since the program was last bound and compiled.
Chapter 11. Testing and debugging an application program on Db2 for z/OS

Depending on the situation, testing your application program might involve setting up a test environment, testing SQL statements, debugging your programs, and reading output from the precompiler.

Related tasks
Modeling a production environment on a test subsystem (Db2 Performance)
Modeling your production system statistics in a test subsystem (Db2 Performance)

Designing a test data structure

When you test an application that accesses Db2 data, you should have Db2 data available for testing. To do this, you can create test tables and views.

About this task

• **Test views of existing tables:** If your application does not change a set of Db2 data and the data exists in one or more production-level tables, you might consider using a view of existing tables.

• **Test tables:** To create a test table, you need a database and table space. Talk with your DBA to make sure that a database and table spaces are available for your use.

  If the data that you want to change already exists in a table, consider using the LIKE clause of CREATE TABLE. If you want others besides yourself to have ownership of a table for test purposes, you can specify a secondary ID as the owner of the table. You can do this with the SET CURRENT SQLID statement.

If your location has a separate Db2 system for testing, you can create the test tables and views on the test system. This information assumes that you do all testing on a separate system, and that the person who created the test tables and views has an authorization ID of TEST. The table names are TEST.EMP, TEST.PROJ and TEST.DEPT.

Related concepts
Authorization IDs (Managing Security)

Related tasks
Modeling a production environment on a test subsystem (Db2 Performance)
Modeling your production system statistics in a test subsystem (Db2 Performance)

Related reference
SET CURRENT SQLID (Db2 SQL)

Analyzing application data needs

To design tests of an application, you need to determine the type of data that the application uses and how the application accesses that data.

About this task

This information assumes that you do all testing on a separate system, and that the person who created the test tables and views has an authorization ID of TEST. The table names are TEST.EMP, TEST.PROJ and TEST.DEPT.

To design test tables and views, first analyze the data needs of your application.

Procedure

To analyze the data needs of your application:
1. List the data that your application accesses and describe how it accesses each data item.
For example, suppose that you are testing an application that accesses the DSN8C10.EMP,
DSN8C10.DEPT, and DSN8C10.PROJ tables. You might record the information about the data as
shown in Table 152 on page 948.

Table 152. Description of the application data

<table>
<thead>
<tr>
<th>Table or view name</th>
<th>Insert rows?</th>
<th>Delete rows?</th>
<th>Column name</th>
<th>Data type</th>
<th>Update access?</th>
</tr>
</thead>
<tbody>
<tr>
<td>DSN8C10.EMP</td>
<td>No</td>
<td>No</td>
<td>EMPNO</td>
<td>CHAR(6)</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>LASTNAME</td>
<td>VARCHAR(15)</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>WORKDEPT</td>
<td>CHAR(3)</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>PHONENO</td>
<td>CHAR(4)</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>JOB</td>
<td>DECIMAL(3)</td>
<td>Yes</td>
</tr>
<tr>
<td>DSN8C10.DEPT</td>
<td>No</td>
<td>No</td>
<td>DEPTNO</td>
<td>CHAR(3)</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>MGRNO</td>
<td>CHAR(6)</td>
<td>No</td>
</tr>
<tr>
<td>DSN8C10.PROJ</td>
<td>Yes</td>
<td>Yes</td>
<td>PROJNO</td>
<td>CHAR(6)</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>DEPTNO</td>
<td>CHAR(3)</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>RESPEMP</td>
<td>CHAR(6)</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>PRSTAFF</td>
<td>DECIMAL(5,2)</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>PRSTDATE</td>
<td>DECIMAL(6)</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>PRENDATE</td>
<td>DECIMAL(6)</td>
<td>Yes</td>
</tr>
</tbody>
</table>

2. Determine the test tables and views that you need to test your application.
Create a test table on your list when either of the following conditions exists:
• The application modifies data in the table.
• You need to create a view that is based on a test table because your application modifies data in the
  view.

To continue the example, create these test tables:
• TEST.EMP, with the following format:

<table>
<thead>
<tr>
<th>EMPNO</th>
<th>LASTNAME</th>
<th>WORKDEPT</th>
<th>PHONENO</th>
<th>JOB</th>
</tr>
</thead>
</table>

• TEST.PROJ, with the same columns and format as DSN8C10.PROJ, because the application inserts
  rows into the DSN8C10.PROJ table.

To support the example, create a test view of the DSN8C10.DEPT table.
• TEST.DEPT view, with the following format:

<table>
<thead>
<tr>
<th>DEPTNO</th>
<th>MGRNO</th>
</tr>
</thead>
</table>

Because the application does not change any data in the DSN8C10.DEPT table, you can base the view
on the table itself (rather than on a test table). However, a safer approach is to have a complete set of
test tables and to test the program thoroughly using only test data.
Authorization for test tables and applications

Before you can create a table, you need to be authorized to create tables and to use the table space in which the table is to reside. You must also have authority to bind and run programs that you want to test.

Your DBA can grant you the necessary authorization to create and access tables and to bind and run programs.

If you intend to use existing tables and views (either directly or as the basis for a view), you need privileges to access those tables and views. Your DBA can grant those privileges.

To create a view, you must have authorization for each table and view on which you base the view. You then have the same privileges over the view that you have over the tables and views on which you based the view. Before trying the examples, have your DBA grant you the privileges to create new tables and views and to access existing tables. Obtain the names of tables and views that you are authorized to access (as well as the privileges you have for each table) from your DBA.

Example SQL statements to create a comprehensive test structure

You need to create a storage group, database, table space, and table to use as a test structure for your SQL application.

The following SQL statements show how to create a complete test structure to contain a small table named SPUFINUM. The test structure consists of:

- A storage group named SPUFISG
- A database named SPUFIDB
- A table space named SPUFITS in database SPUFIDB and using storage group SPUFISG
- A table named SPUFINUM within the table space SPUFITS

```
CREATE STOGROUP SPUFISG
    VOLUMES (user-volume-number)
    VCAT DSNCAT ;
CREATE DATABASE SPUFIDB ;
CREATE TABLESPACE SPUFITS
    IN SPUFIDB
    USING STOGROUP SPUFISG ;
CREATE TABLE SPUFINUM
    ( XVAL CHAR(12) NOT NULL,
    ISFLOAT FLOAT,
    DEC30 DECIMAL(3,0),
    DEC31 DECIMAL(3,1),
    DEC32 DECIMAL(3,2),
    DEC33 DECIMAL(3,3),
    DEC10 DECIMAL(1,0),
    DEC11 DECIMAL(1,1),
    DEC150 DECIMAL(15,0),
    DEC151 DECIMAL(15,1),
    DEC1515 DECIMAL(15,15) )
    IN SPUFIDB.SPUFITS ;
```

Related reference

CREATE DATABASE (Db2 SQL)
CREATE STOGROUP (Db2 SQL)
CREATE TABLE (Db2 SQL)
CREATE TABLESPACE (Db2 SQL)
Populating the test tables with data

To populate test tables, use SQL INSERT statements or the LOAD utility.

About this task

You can put test data into a table in several ways:

- INSERT ... VALUES (an SQL statement) puts one row into a table each time the statement executes.
- INSERT ... SELECT (an SQL statement) obtains data from an existing table (based on a SELECT clause) and puts it into the table that is identified in the INSERT statement.
- MERGE (an SQL statement) puts new data into a table and updates existing data.
- The LOAD utility obtains data from a sequential file (a non-Db2 file), formats it for a table, and puts it into a table.
- The Db2 sample UNLOAD program (DSNTIAUL) can unload data from a table or view and build control statements for the LOAD utility.
- The UNLOAD utility can unload data from a table and build control statements for the LOAD utility.
- Redirected recovery, the RECOVER utility with the FROM option, can recover data from a production table to a test table with no impact to the availability of the production data and applications. The production data can be recovered to the test object to a point in time or to the current state with transactional consistency. SQL or the UNLOAD utility can then be used on the data in the test table.

Related concepts

Sample applications supplied with Db2 for z/OS
Db2 provides sample applications to help you with Db2 programming techniques and coding practices within each of the four environments: batch, TSO, IMS, and CICS. The sample applications contain various applications that might apply to managing a company.

Related tasks

Inserting rows by using the INSERT statement
One way to insert data into tables is to use the SQL INSERT statement. This method is useful for inserting small amounts of data or inserting data from another table or view.

Inserting rows into a table from another table
You can copy one or more rows from one table into another table.

Inserting data and updating data in a single operation
You can update existing data and insert new data in a single operation. This operation is useful when you want to update a table with a set of rows, some of which are changes to existing rows and some of which are new rows.

Running a redirected recovery (Db2 Utilities)

Related reference

LOAD (Db2 Utilities)
UNLOAD (Db2 Utilities)

Methods for testing SQL statements

You can test your SQL statements by using SQL Processing Using File Input (SPUFI) or the command line processor.

Test with SPUFI: You can use SPUFI (an interface between ISPF and Db2) to test SQL statements in a TSO/ISPF environment. With SPUFI panels, you can put SQL statements into a data set that Db2 subsequently executes. The SPUFI Main panel has several functions that enable you to:

- Name an input data set to hold the SQL statements that are passed to Db2 for execution
- Name an output data set to contain the results of executing the SQL statements
- Specify SPUFI processing options
Test with the command line processor: You can use the command line processor to test SQL statements from UNIX System Services on z/OS.

SQL statements that are executed under SPUFI or the command line processor operate on actual tables (in this case, the tables that you created for testing). Consequently, before you access Db2 data:

- Make sure that all tables and views that your SQL statements refer to exist.
- If the tables or views do not exist, create them (or have your database administrator create them). You can use SPUFI or the command line processor to issue the CREATE statements that are used to create the tables and views that you need for testing.

Related concepts
The Db2 command line processor (Db2 Commands)

Related tasks
Executing SQL by using SPUFI

You can execute SQL statements dynamically in a TSO session by using the SPUFI (SQL processor using file input) facility.

**Executing SQL by using SPUFI**

You can execute SQL statements dynamically in a TSO session by using the SPUFI (SQL processor using file input) facility.

**Before you begin**

Before you use SPUFI, allocate an input data set to store the SQL statements that you want to execute, if such a data set does not already exist.

Before you begin this task, you can specify whether TSO message IDs are displayed by using the TSO PROFILE command. To view message IDs, type `TSO PROFILE MSGID` on the ISPF command line. To suppress message IDs, type `TSO PROFILE NOMSGID`.

These instructions assume that ISPF is available to you.

**About this task**

Important: Ensure that the TSO terminal CCSID matches the Db2 CCSID. If these CCSIDs do not match, data corruption can occur. If SPUFI issues the warning message DSNE345I, terminate your SPUFI session and notify the system administrator.

SPUFI can execute SQL statements that retrieve Unicode UTF-16 graphic data. However, SPUFI might not be able to display some characters, if those characters have no mapping in the target SBCS EBCDIC CCSID.

**Procedure**

To execute SQL by using SPUFI:

1. Open SPUFI and specify the initial options. To open SPUFI and specify initial options:
   
   a) Select SPUFI from the DB2I Primary Option Menu as shown in [The DB2I primary option menu (Introduction to Db2 for z/OS)].

   The SPUFI panel is displayed.

   b) Specify the input data set name and output data set name.

   An example of a SPUFI panel in which an input data set and output data set have been specified is shown in the following figure.
c) Specify new values in any of the other fields on the SPUFI panel.
For more information about these fields, see “The SPUFI panel” on page 955.

2. Optional: Change the SPUFI defaults, as described in “Changing SPUFI defaults” on page 956.

3. Enter SQL statements in SPUFI.
If the input data set that you specified on the SPUFI panel already contains all of the SQL statements that you want to execute, you can bypass this editing step by specifying NO for the EDIT INPUT field on the SPUFI panel. To enter SQL statements by using SPUFI:

a) If the EDIT panel is not already open, on the SPUFI panel, specify Y in the EDIT INPUT field and press ENTER. If the input data set that you specified is empty, an empty EDIT panel opens. Otherwise, if the input data set contained SQL statements, those SQL statements are displayed in an EDIT panel.

b) On the EDIT panel, use the ISPF EDIT program to enter or edit any SQL statements that you want to execute. Move the cursor to the first blank input line, and enter the first part of an SQL statement. If the input data set that you specified on the SPUFI panel already contains all of the SQL statements that you want to execute, you can bypass this editing step by specifying NO for the EDIT INPUT field on the SPUFI panel.

You can enter the rest of the SQL statement on subsequent lines, as shown in the following figure:

```
EDIT --------userid.EXAMPLES(XMP1) -------------- COLUMNS 001 072
COMMAND INPUT ===> SAVE                       SCROLL ===> PAGE
******************************************************************************
000100 SELECT LASTNAME, FIRSTNME, PHONENO
000200   FROM DSN8C10.EMP
000300   WHERE WORKDEPT= 'D11'
000400   ORDER BY LASTNAME;
******************************************************************************
```

Figure 65. The edit panel: After entering an SQL statement

Consider the following rules and recommendations when editing this input data set:

- Indent your lines and enter your statements on several lines to make your statements easier to read. Entering your statements on multiple lines does not change how your statements are processed.

- Do not put more than one SQL statement on a single line. If you do, the first statement executes, but Db2 ignores the other SQL statements on the same line. You can put more than one SQL statement in the input data set. Db2 executes the statements in the order in which you placed them in the data set.
If the length of an SQL statement is greater than 71 bytes for an input data set with record length 79, or 72 bytes for an input data set with record length 80, you need to continue the SQL statement on additional lines of the SPUFI input data set. SPUFI concatenates the text on multiple lines without adding extra spaces at the end of any line. Therefore, if an SQL statement contains two values with a space between them, and the first value ends at the last allowed input position (71 or 72), you need to add an extra space on the next line before the second value.

For example, suppose that the record length of your SPUFI input data set is 80, so the maximum length of an input line is 72. Also suppose that the SQL statement that you wish to enter is 81 bytes long. Bytes 69 through 81 contain FROM MYTABLE; . If you split the SQL statement after FROM, the first line of the statement ends in column 72, so you need to include a space in column 1 of the next line, before MYTABLE; . Otherwise, when SPUFI concatenates the two lines, the result is FROMMYTABLE;. When SPUFI runs the SQL statement, the SQL statement fails with SQLCODE -104.

End each SQL statement with the statement terminator that you specified on the CURRENT SPUFI DEFAULTS panel.

Save the data set every 10 minutes or so by entering the SAVE command.

c) Press the END PF key.

The data set is saved, and the SPUFI panel is displayed.

4. Process SQL statements with SPUFI.

You can use SPUFI to submit the SQL statements in a data set to Db2. To process SQL statements by using SPUFI:

a) On the SPUFI panel, specify YES in the EXECUTE field.

b) If you did not just finish using the EDIT panel to edit the input data set as described in "Entering SQL statements in SPUFI," specify NO in the EDIT INPUT field.

c) Press Enter.

SPUFI passes the input data set to Db2 for processing. Db2 executes the SQL statement in the input data set and sends the output to the output data set.

The output data set opens.

Your SQL statement might take a long time to execute, depending on how large a table Db2 must search, or on how many rows Db2 must process. In this case, you can interrupt the processing by pressing the PA1 key. Then respond to the message that asks you if you really want to stop processing. This action cancels the executing SQL statement. Depending on how much of the input data set Db2 was able to process before you interrupted its processing, Db2 might not have opened the output data set yet, or the output data set might contain all or part of the results data that are produced so far.

Results

SQL statements that exceed resource limit thresholds

Your system administrator might use the Db2 resource limit facility (governor) to set time limits for processing SQL statements in SPUFI. Those limits can be error limits or warning limits.

If you execute an SQL statement through SPUFI that runs longer than this error time limit, SPUFI terminates processing of that SQL statement and all statements that follow in the SPUFI input data set. SPUFI displays a panel that lets you commit or roll back the previously uncommitted changes that you have made. That panel is shown in the following figure.
The following SQL statement has encountered an SQLCODE of -985 or -495:

Statement text

Your SQL statement has exceeded the resource utilization threshold set by your site administrator.

You must ROLLBACK or COMMIT all the changes made since the last COMMIT. SPUFI processing for the current input file will terminate immediately after the COMMIT or ROLLBACK is executed.

1  NEXT ACTION ===>           (Enter COMMIT or ROLLBACK)

PRESS: ENTER to process                             HELP for more information

Figure 66. The resource limit facility error panel

If you execute an SQL statement through SPUFI that runs longer than the warning time limit for predictive governing, SPUFI displays the SQL STATEMENT RESOURCE LIMIT EXCEEDED panel. On this panel, you can tell Db2 to continue executing that statement, or stop processing that statement and continue to the next statement in the SPUFI input data set. That panel is shown in the following figure.

The following SQL statement has encountered an SQLCODE of 495:

Statement text

You can now either CONTINUE executing this statement or BYPASS the execution of this statement. SPUFI processing for the current input file will continue after the CONTINUE or BYPASS processing is completed.

1  NEXT ACTION ===>           (Enter CONTINUE or BYPASS)

PRESS: ENTER to process                             HELP for more information

Figure 67. The resource limit facility warning panel

Related tasks
Setting limits for system resource usage by using the resource limit facility (Db2 Performance)

Related reference
Using ISPF/PDF to Allocate Data Sets (z/OS ISPF User's Guide Vol II)

Related information
Lesson 1.1: Querying data interactively (Introduction to Db2 for z/OS)

Content of a SPUFI input data set

A SPUFI input data set can contain SQL statements, comments, and SPUFI control statements.

You can put comments about SQL statements either on separate lines or on the same line. In either case, use two hyphens (--) to begin a comment. Specify any text other than #SET TERMINATOR or #SET TOLWARN after the comment marker. Db2 ignores everything else to the right of the two hyphens.
The SPUFI panel

The SPUFI panel is the first panel that you need to fill out to run the SPUFI application.

After you complete any fields on the SPUFI panel and press Enter, those settings are saved. When the SPUFI panel displays again, the data entry fields on the panel contain the values that you previously entered. You can specify data set names and processing options each time the SPUFI panel is displayed, as needed. Values that you do not change remain in effect.

The following descriptions explain the fields that are available on the SPUFI panel.

1,2,3 INPUT DATA SET NAME

Identify the input data set in fields 1 through 3. This data set contains one or more SQL statements that you want to execute. Allocate this data set before you use SPUFI, if one does not already exist. Consider the following rules:

• The name of the data set must conform to standard TSO naming conventions.
• The data set can be empty before you begin the session. You can then add the SQL statements by editing the data set from SPUFI.
• The data set can be either sequential or partitioned, but it must have the following DCB characteristics:
  – A record format (RECFM) of either F or FB.
  – A logical record length (LRECL) of either 79 or 80. Use 80 for any data set that the EXPORT command of QMF did not create.
• Data in the data set can begin in column 1. It can extend to column 71 if the logical record length is 79, and to column 72 if the logical record length is 80. SPUFI assumes that the last 8 bytes of each record are for sequence numbers.

If you use this panel a second time, the name of the data set you previously used displays in the field DATA SET NAME. To create a new member of an existing partitioned data set, change only the member name.

4 OUTPUT DATA SET NAME

Enter the name of a data set to receive the output of the SQL statement. You do not need to allocate the data set before you do this.

If the data set exists, the new output replaces its content. If the data set does not exist, Db2 allocates a data set on the device type specified on the CURRENT SPUFI DEFAULTS panel and then catalogs the new data set. The device must be a direct-access storage device, and you must be authorized to allocate space on that device.

Attributes required for the output data set are:

• Organization: sequential
• Record format: F, FB, FBA, V, VB, or VBA
• Record length: 80 to 32768 bytes, not less than the input data set

“Executing SQL by using SPUFI” on page 951 shows the simplest choice, entering RESULT. SPUFI allocates a data set named userid.RESULT and sends all output to that data set. If a data set named userid.RESULT already exists, SPUFI sends Db2 output to it, replacing all existing data.

5 CHANGE DEFAULTS

Enables you to change control values and characteristics of the output data set and format of your SPUFI session. If you specify Y (YES) you can look at the SPUFI defaults panel. See “Changing SPUFI defaults” on page 956 for more information about the values you can specify and how they affect SPUFI processing and output characteristics. You do not need to change the SPUFI defaults for this example.

6 EDIT INPUT

To edit the input data set, leave Y (YES) on line 6. You can use the ISPF editor to create a new member of the input data set and enter SQL statements in it. (To process a data set that already contains a set
of SQL statements you want to execute immediately, enter N (NO). Specifying N bypasses the step 3 described in “Executing SQL by using SPUFI” on page 951.)

7 EXECUTE
To execute SQL statements contained in the input data set, leave Y(YES) on line 7.
SPUFI handles the SQL statements that can be dynamically prepared.

8 AUTOCOMMIT
To make changes to the Db2 data permanent, leave Y(YES) on line 8. Specifying Y makes SPUFI issue COMMIT if all statements execute successfully. If all statements do not execute successfully, SPUFI issues a ROLLBACK statement, which deletes changes already made to the file (back to the last commit point).

If you specify N, Db2 displays the SPUFI COMMIT OR ROLLBACK panel after it executes the SQL in your input data set. That panel prompts you to COMMIT, ROLLBACK, or DEFER any updates made by the SQL. If you enter DEFER, you neither commit nor roll back your changes.

9 BROWSE OUTPUT
To look at the results of your query, leave Y(YES) on line 9. SPUFI saves the results in the output data set. You can look at them at any time, until you delete or write over the data set.

10 CONNECT LOCATION
Specify the name of the database server, if applicable, to which you want to submit SQL statements. SPUFI then issues a type 2 CONNECT statement to this server.

SPUFI is a locally bound package. SQL statements in the input data set can process only if the CONNECT statement is successful. If the connect request fails, the output data set contains the resulting SQL return codes and error messages.

Related reference
Characteristics of SQL statements in Db2 for z/OS (Db2 SQL)
COMMIT (Db2 SQL)
ROLLBACK (Db2 SQL)

Changing SPUFI defaults

Before you execute SQL statements in SPUFI, you can change the default execution behavior, such as the SQL terminator and the isolation level.

About this task
SPUFI provides default values the first time that you use SPUFI for all options except the Db2 subsystem name. Any changes that you make to these values remain in effect until you change the values again.

Procedure
To change the SPUFI defaults:
1. On the SPUFI panel, specify YES in the CHANGE DEFAULTS field.
2. Press Enter.
   The CURRENT SPUFI DEFAULTS panel opens. The following figure shows the initial default values.
Enter the following to control your SPUFI session:
1. SQL TERMINATOR .. ===> ; (SQL Statement Terminator)
2. ISOLATION LEVEL ===> RR (RR=Repeatable Read, CS=Cursor Stability)
   (UR=Uncommitted Read)
3. MAX SELECT LINES ===> 250 (Maximum lines to be returned from a SELECT)
4. ALLOW SQL WARNINGS ===> NO (Continue fetching after SQL warning)
5. CHANGE PLAN NAMES ===> NO (Change the plan names used by SPUFI)
6. SQL FORMAT ...... ===> SQL (SQL, SQLCOMNT, or SQLPL)

Output data set characteristics:
7. SPACE UNIT ...... ===> TRK (TRK or CYL)
8. PRIMARY SPACE ... ===> 5 (Primary space allocation 1-999)
9. SECONDARY SPACE : ===> 6 (Secondary space allocation 0-999)
10. RECORD LENGTH ... ===> 4092 (LRECL= logical record length)
11. BLOCKSIZE ........ ===> 4096 (Size of one block)
12. RECORD FORMAT.... ===> VB (RECFM= F, FB, FBA, V, VB, or VB)
13. DEVICE TYPE ...... ===> SYSDA (Must be a DASD unit name)

Output format characteristics:
14. MAX NUMERIC FIELD ===> 33 (Maximum width for numeric field)
15. MAX CHAR FIELD .. ===> 80 (Maximum width for character field)
16. COLUMN HEADING .. ===> NAMES (NAMES, LABELS, ANY, or BOTH)

Figure 68. The SPUFI defaults panel

3. Specify any new values in the fields of this panel. All fields must contain a value.
4. Press Enter.
   SPUFI saves your changes and one of the following panels or data sets open:

   - The CURRENT SPUFI DEFAULTS - PANEL 2 panel. This panel opens if you specified YES in the
     CHANGE PLAN NAMES field.
   - EDIT panel. This panel opens if you specified YES in the EDIT INPUT field on the SPUFI panel.
   - Output data set. This data set opens if you specified NO in the EDIT INPUT field on the SPUFI panel.
   - SPUFI panel. This panel opens if you specified NO for all of the processing options on the SPUFI
     panel.

   If you press the END key on the CURRENT SPUFI DEFAULTS panel, the SPUFI panel is displayed, and
you lose all the changes that you made on the CURRENT SPUFI DEFAULTS panel.
5. If the CURRENT SPUFI DEFAULTS - PANEL 2 panel opens, specify values for the fields on that panel
and press Enter. All fields must contain a value.

   Important: If you specify an invalid or incorrect plan name, SPUFI might experience operational errors
or your data might be contaminated.

   SPUFI saves your changes and one of the following panels or data sets open:

   - EDIT panel. This panel opens if you specified YES in the EDIT INPUT field on the SPUFI panel.
   - Output data set. This data set opens if you specified NO in the EDIT INPUT field on the SPUFI panel.
   - SPUFI panel. This panel opens if you specified NO for all of the processing options on the SPUFI
     panel.

Results

Next, continue with one of the following tasks:

- If you want to add SQL statements to the input data set or edit the SQL statements in the input data set,
  enter SQL statements in SPUFI.
- Otherwise if the input data set already contains the SQL statements that you want to execute, process
  SQL statements with SPUFI.

Related reference
CURRENT SPUFI DEFAULTS panel
Use the CURRENT SPUFI DEFAULTS panel to specify SPUFI default values.

CURRENT SPUFI DEFAULTS - PANEL 2 panel
Use the CURRENT SPUFI DEFAULTS - PANEL 2 panel to specify default plan name information.

CURRENT SPUFI DEFAULTS panel
Use the CURRENT SPUFI DEFAULTS panel to specify SPUFI default values.

The following descriptions explain the information on the CURRENT SPUFI DEFAULTS panel.

1 SQL TERMINATOR
Specify the character that you use to end each SQL statement. You can specify any character except the characters listed in the following table. A semicolon (;) is the default SQL terminator.

<table>
<thead>
<tr>
<th>Name</th>
<th>Character</th>
<th>Hexadecimal representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>blank</td>
<td></td>
<td>X'40'</td>
</tr>
<tr>
<td>comma</td>
<td></td>
<td>X'5E'</td>
</tr>
<tr>
<td>double quote</td>
<td></td>
<td>X'7F'</td>
</tr>
<tr>
<td>left parenthesis</td>
<td></td>
<td>X'4D'</td>
</tr>
<tr>
<td>right parenthesis</td>
<td></td>
<td>X'5D'</td>
</tr>
<tr>
<td>single quote</td>
<td></td>
<td>X'7D'</td>
</tr>
<tr>
<td>underscore</td>
<td></td>
<td>X'6D'</td>
</tr>
</tbody>
</table>

Use a character other than a semicolon if you plan to execute a statement that contains embedded semicolons. For example, suppose you choose the character # as the statement terminator. Then a CREATE TRIGGER statement with embedded semicolons looks like the following statement:

```
CREATE TRIGGER NEW_HIRE
    AFTER INSERT ON EMP
    FOR EACH ROW MODE DB2SQL
    BEGIN ATOMIC
        UPDATE COMPANY_STATS SET NBEMP = NBEMP + 1;
    END#
```

A CREATE PROCEDURE statement with embedded semicolons looks like the following statement:

```
CREATE PROCEDURE PROC1 (IN PARM1 INT, OUT SCODE INT)
    LANGUAGE SQL
    BEGIN
        DECLARE SQLCODE INT;
        DECLARE EXIT HANDLER FOR SQLEXCEPTION
            SET SCODE = SQLCODE;
        UPDATE TBL1 SET COL1 = PARM1;
    END #
```

Be careful to choose a character for the SQL terminator that is not used within the statement.

You can also set or change the SQL terminator within a SPUFI input data set by using the --#SET TERMINATOR statement.

2 ISOLATION LEVEL
Specify the isolation level for your SQL statements.

3 MAX SELECT LINES
The maximum number of rows that a SELECT statement can return. To limit the number of rows retrieved, enter an integer greater than 0.
4 ALLOW SQL WARNINGS
Enter YES or NO to indicate whether SPUFI will continue to process an SQL statement after receiving SQL warnings:

**YES**
If a warning occurs when SPUFI executes an OPEN or FETCH for a SELECT statement, SPUFI continues to process the SELECT statement.

**NO**
If a warning occurs when SPUFI executes an OPEN or FETCH for a SELECT statement, SPUFI stops processing the SELECT statement. If SQLCODE +802 occurs when SPUFI executes a FETCH for a SELECT statement, SPUFI continues to process the SELECT statement.

You can also specify how SPUFI pre-processes the SQL input by using the `--#SET TOLWARN` statement.

5 CHANGE PLAN NAMES
If you enter YES in this field, you can change plan names on a subsequent SPUFI defaults panel, DSNESP07. Enter YES in this field only if you are certain that you want to change the plan names that are used by SPUFI. Consult with your Db2 system administrator if you are uncertain whether you want to change the plan names. Using an invalid or incorrect plan name might cause SPUFI to experience operational errors or it might cause data contamination.

6 SQL FORMAT
Specify how SPUFI pre-processes the SQL input before passing it to Db2. Select one of the following options:

**SQL**
This is the preferred mode for SQL statements other than SQL procedural language. When you use this option, which is the default, SPUFI collapses each line of an SQL statement into a single line before passing the statement to Db2. SPUFI also discards all SQL comments.

**SQLCOMNT**
This mode is suitable for all SQL, but it is intended primarily for SQL procedural language processing. When this option is in effect, behavior is similar to SQL mode, except that SPUFI does not discard SQL comments. Instead, it automatically terminates each SQL comment with a line feed character (hex 25), unless the comment is already terminated by one or more line formatting characters. Use this option to process SQL procedural language with minimal modification by SPUFI.

**SQLPL**
This mode is suitable for all SQL, but it is intended primarily for SQL procedural language processing. When this option is in effect, SPUFI retains SQL comments and terminates each line of an SQL statement with a line feed character (hex 25) before passing the statement to Db2. Lines that end with a split token are not terminated with a line feed character. Use this mode to obtain improved diagnostics and debugging of SQL procedural language.

You can also specify how SPUFI pre-processes the SQL input by using the `--#SET SQLFORMAT` statement.

7 SPACE UNIT
Specify how space for the SPUFI output data set is to be allocated.

**TRK**
Track

**CYL**
Cylinder

8 PRIMARY SPACE
Specify how many tracks or cylinders of primary space are to be allocated.

9 SECONDARY SPACE
Specify how many tracks or cylinders of secondary space are to be allocated.
10 RECORD LENGTH
The record length must be at least 80 bytes. The maximum record length depends on the device type you use. The default value allows a 32756-byte record.

Each record can hold a single line of output. If a line is longer than a record, the output is truncated, and SPUFI discards fields that extend beyond the record length.

11 BLOCKSIZE
Follow the normal rules for selecting the block size. For record format F, the block size is equal to the record length. For FB and FBA, choose a block size that is an even multiple of LRECL. For VB and VBA only, the block size must be 4 bytes larger than the block size for FB or FBA.

12 RECORD FORMAT
Specify F, FB, FBA, V, VB, or VBA. FBA and VBA formats insert a printer control character after the number of lines specified in the LINES/PAGE OF LISTING field on the DB2I Defaults panel. The record format default is VB (variable-length blocked).

13 DEVICE TYPE
Specify a standard z/OS name for direct-access storage device types. The default is SYSDA. SYSDA specifies that z/OS is to select an appropriate direct access storage device.

14 MAX NUMERIC FIELD
The maximum width of a numeric value column in your output. Choose a value greater than 0. The default is 33.

15 MAX CHAR FIELD
The maximum width of a character value column in your output. DATETIME and GRAPHIC data strings are externally represented as characters, and SPUFI includes their defaults with the default values for character fields. Choose a value greater than 0. The IBM-supplied default is 250.

16 COLUMN HEADING
You can specify NAMES, LABELS, ANY, or BOTH for column headings.

- NAMES uses column names only.
- LABELS (default) uses column labels. Leave the title blank if no label exists.
- ANY uses existing column labels or column names.
- BOTH creates two title lines, one with names and one with labels.

Column names are the column identifiers that you can use in SQL statements. If an SQL statement has an AS clause for a column, SPUFI displays the contents of the AS clause in the heading, rather than the column name. You define column labels with LABEL statements.

Related concepts
Output from SPUFI
SPUFI formats and displays the output data set using the ISPF Browse program.

Related tasks
Changing SPUFI defaults
Before you execute SQL statements in SPUFI, you can change the default execution behavior, such as the SQL terminator and the isolation level.

Executing SQL by using SPUFI
You can execute SQL statements dynamically in a TSO session by using the SPUFI (SQL processor using file input) facility.

CURRENT SPUFI DEFAULTS - PANEL 2 panel
Use the CURRENT SPUFI DEFAULTS - PANEL 2 panel to specify default plan name information.

This panel opens if you specify YES in the CHANGE PLAN NAMES field of the CURRENT SPUFI DEFAULTS panel.

Figure 69 on page 961 shows the initial default values.
Enter the following to control your SPUFI session:
1 CS ISOLATION PLAN ===> DSNESPCS (Name of plan for CS isolation level)
2 RR ISOLATION PLAN ===> DSNESPRR (Name of plan for RR isolation level)
3 UR ISOLATION PLAN ===> DSNESPUR (Name of plan for UR isolation level)

Indicate warning message status:
4 BLANK CCSID WARNING ===> YES (Show warning if terminal CCSID is blank)

PRESS: ENTER to process     END to exit    HELP for more information

Figure 69. CURRENT SPUFI DEFAULTS - PANEL 2

The following descriptions explain the information on the CURRENT SPUFI DEFAULTS - PANEL 2 panel.

1 CS ISOLATION PLAN
   Specify the name of the plan that SPUFI uses when you specify an isolation level of cursor stability (CS). By default, this name is DSNESPCS.

2 RR ISOLATION PLAN
   Specify the name of the plan that SPUFI uses when you specify an isolation level of repeatable read (RR). By default, this name is DSNESPRR.

3 UR ISOLATION PLAN
   Specify the name of the plan that SPUFI uses when you specify an isolation level of uncommitted read (UR). By default, this name is DSNESPUR.

4 BLANK CCSID ALERT
   Indicate whether to receive message DSNE345I when the terminal CCSID setting is blank. A blank terminal CCSID setting occurs when the terminal code page and character set cannot be queried or if they are not supported by ISPF.

   Recommendation: To avoid possible data contamination use the default setting of YES, unless you are specifically directed by your Db2 system administrator to use NO.

Setting the SQL terminator character in a SPUFI input data set

In the SPUFI input data set, you can override the SQL terminator character that is specified on the CURRENT SPUFI DEFAULTS panel. The default SQL terminator is a semicolon (;).

About this task

Overriding the default SQL termination character is useful if you need to use a different SQL terminator character for one particular SQL statement.

To set the SQL terminator character in a SPUFI input data set, specify the text --#SET TERMINATOR character before that SQL statement to which you want this character to apply. This text specifies that SPUFI is to interpret character as a statement terminator. You can specify any single-byte character except the characters that are listed in Table 154 on page 962. Choose a character for the SQL terminator that is not used within the statement. The terminator that you specify overrides a terminator that you specified in option 1 of the CURRENT SPUFI DEFAULTS panel or in a previous --#SET TERMINATOR statement.
### Table 154. Invalid special characters for the SQL terminator

<table>
<thead>
<tr>
<th>Name</th>
<th>Character</th>
<th>Hexadecimal representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>blank</td>
<td></td>
<td>X'40'</td>
</tr>
<tr>
<td>comma</td>
<td>,</td>
<td>X'5E'</td>
</tr>
<tr>
<td>double quote</td>
<td>&quot;</td>
<td>X'7F'</td>
</tr>
<tr>
<td>left parenthesis</td>
<td>(</td>
<td>X'4D'</td>
</tr>
<tr>
<td>right parenthesis</td>
<td>)</td>
<td>X'5D'</td>
</tr>
<tr>
<td>single quote</td>
<td>'</td>
<td>X'7D'</td>
</tr>
<tr>
<td>underscore</td>
<td>_</td>
<td>X'6D'</td>
</tr>
</tbody>
</table>

Use a character other than a semicolon if you plan to execute a statement that contains embedded semicolons. For example, suppose that you choose the character # as the statement terminator. In this case, a CREATE TRIGGER statement with embedded semicolons looks like this:

```sql
CREATE TRIGGER NEW_HIRE
AFTER INSERT ON EMP
FOR EACH ROW MODE DB2SQL
BEGIN ATOMIC
   UPDATE COMPANY_STATS SET NBEMP = NBEMP + 1;
END#
```

### Controlling toleration of warnings in SPUFI

When you use SPUFI, you can specify the action that SPUFI is to take when a warning occurs.

#### About this task

To control the toleration of warnings, specify one of the following TOLWARN control statements:

--**SET TOLWARN NO**

If a warning occurs when SPUFI executes an OPEN or FETCH for SELECT statement, SPUFI stops processing the SELECT statement. If SQLCODE +802 occurs when SPUFI executes a FETCH for a SELECT statement, SPUFI continues to process the SELECT statement.

--**SET TOLWARN YES**

If a warning occurs when SPUFI executes an OPEN or FETCH for SELECT statement, SPUFI continues to process the SELECT statement.

--**SET TOLWARN QUIET**

The same as YES, except that SPUFI suppresses all SQL warning messages from OPEN or FETCH if the SQLCODE is 0 or greater.

#### Example

The following example activates and then deactivates toleration of SQL warnings:

```sql
SELECT * FROM MY.T1;
--SET TOLWARN YES
SELECT * FROM YOUR.T1;
--SET TOLWARN NO
```

### Output from SPUFI

SPUFI formats and displays the output data set using the ISPF Browse program.

An output data set contains the following items for each SQL statement that Db2 executes:
• The executed SQL statement, copied from the input data set
• The results of executing the SQL statement.
• The SQLCODE, and if the execution is unsuccessful, the formatted SQLCA.

At the end of the data set are summary statistics that describe the processing of the input data set as a whole.

For SELECT statements that are executed with SPUFI, the message "SQLCODE IS 100" indicates an error-free result. If the message SQLCODE IS 100 is the only result, Db2 is unable to find any rows that satisfy the condition that is specified in the statement.

For all other types of SQL statements that are executed with SPUFI, the message "SQLCODE IS 0" indicates an error-free result.

**Example SPUFI output**

For example, the sample program returns the following output.

```
BROWSE-- userid.RESULT                        COLUMNS 001 072
COMMAND INPUT ===>                                           SCROLL ===> PAGE
--------+---------+---------+---------+---------+---------+---------+---------+
SELECT LASTNAME, FIRSTNME, PHONENO                                    00010000
FROM DSN8C10.EMP                                                     00020000
WHERE WORKDEPT = 'D11'                                               00030000
ORDER BY LASTNAME;                                                   00040000
---------+---------+---------+---------+---------+---------+---------+---------+
LASTNAME         FIRSTNME      PHONENO
ADAMSON          BRUCE         4510
BROWN            DAVID         4501
JOHN             REBA          0672
JONES            WILLIAM       0942
LUTZ             JENNIFER      0672
PIANKA           ELIZABETH     3782
SCOUTTEN         MARILYN       1682
STERN            IRVING        6423
WALKER           JAMES         2986
YAMAMOTO         KIYOSHI       2890
YOSHIMURA        MASATOSHI     2890
DSNE610I NUMBER OF ROWS DISPLAYED IS 11
DSNE616I STATEMENT EXECUTION WAS SUCCESSFUL, SQLCODE IS 100
---------+---------+---------+---------+---------+---------+---------+---------+
DSNE616I STATEMENT EXECUTION WAS SUCCESSFUL, SQLCODE IS 100
---------+---------+---------+---------+---------+---------+---------+---------+
DSNE617I COMMIT PERFORMED, SQLCODE IS 0
---------+---------+---------+---------+---------+---------+---------+---------+
DSNE661I SQL STATEMENTS ASSUMED TO BE BETWEEN COLUMNS 1 AND 72
DSNE620I NUMBER OF SQL STATEMENTS PROCESSED IS 1
DSNE621I NUMBER OF INPUT RECORDS READ IS 4
DSNE622I NUMBER OF OUTPUT RECORDS WRITTEN IS 30
```

Figure 70. Result data set from the sample problem

**Formatting rules for SELECT statement results in SPUFI**

The results of SELECT statements follow these rules:

• If numeric or character data of a column cannot be displayed completely:
  – Character values and binary values that are too wide truncate on the right.
  – Numeric values that are too wide display as asterisks (*).
  – For columns other than LOB and XML columns, if truncation occurs, the output data set contains a warning message. Because LOB and XML columns are generally longer than the value you choose for field MAX CHAR FIELD on panel CURRENT SPUFI DEFAULTS, SPUFI displays no warning message when it truncates LOB or XML column output.

You can change the amount of data that is displayed for numeric and character columns by changing values on the CURRENT SPUFI DEFAULTS panel, as described in “Changing SPUFI defaults” on page 956.
• A null value is displayed as a series of hyphens (-).
• A ROWID, BLOB, BINARY, or VARBINARY column value is displayed in hexadecimal.
• A CLOB column value is displayed in the same way as a VARCHAR column value.
• A DBCLOB column value is displayed in the same way as a VARGRAPHIC column value.
• An XML column is displayed in the same way as a LOB column.
• A heading identifies each selected column, and is repeated at the top of each output page. The contents of the heading depend on the value that you specified in the COLUMN HEADING field of the CURRENT SPUFI DEFAULTS panel.

Content of the messages from SPUFI

Messages in the SPUFI output contain the following information:

• The SQLCODE, and if the execution is unsuccessful, the formatted SQLCA. If multiple SQL conditions occur, SPUFI returns the error SQLCODE first, followed by any previous SQLCODE conditions that occurred as the input SQL statement was executed. If the final SQLCODE is an error, most prior SQLCODE warnings can be ignored.
• What character positions of the input data set that SPUFI scanned to find SQL statements. This information helps you check the assumptions that SPUFI made about the location of line numbers (if any) in your input data set.
• Some overall statistics:
  - Number of SQL statements that are processed
  - Number of input records that are read (from the input data set)
  - Number of output records that are written (to the output data set).

Other messages that you could receive from the processing of SQL statements include:

• The number of rows that Db2 processed, that either:
  - Your select operation retrieved
  - Your update operation modified
  - Your insert operation added to a table
  - Your delete operation deleted from a table
• Which columns display truncated data because the data was too wide

Testing an external user-defined function

Some commonly used debugging tools, such as TSO TEST, are not available in the environment where user-defined functions run. You need to use alternative testing strategies.

Testing a user-defined function by using the Debug Tool for z/OS

You can use the Debug Tool for z/OS to test Db2 for z/OS user-defined functions that are written in any of the supported languages. The Debug Tool for z/OS works with Language Environment.

About this task

You can use the Debug Tool either interactively or in batch mode. To test your user-defined function using the Debug Tool, you must have the Debug Tool installed on the z/OS system where the user-defined function runs.

Procedure

To test a user-defined function by using the Debug Tool for z/OS, choose one of the following approaches:
• To use the Debug Tool interactively:
a) Compile the user-defined function with the TEST option.
   This places information in the program that the Debug Tool uses.

b) Invoke the Debug Tool. One way to do that is to specify the Language Environment run time TEST option. The TEST option controls when and how the Debug Tool is invoked. The most convenient place to specify run time options is with the RUN OPTIONS clause of CREATE FUNCTION or ALTER FUNCTION.
   For example, suppose that you code this option:

   ```
   TEST(ALL,*,PROMPT,JBJONES%SESSNA:)
   ```

   The parameter values cause the following things to happen:

   **ALL**
   The Debug Tool gains control when an attention interrupt, abend, or program or Language Environment condition of Severity 1 and above occurs.

   ***  
   Debug commands will be entered from the terminal.

   **PROMPT**
   The Debug Tool is invoked immediately after Language Environment initialization.

   **JBJONES%SESSNA:**
   The Debug Tool initiates a session on a workstation identified to APPC as JBJONES with a session ID of SESSNA.

c) If you want to save the output from your debugging session, issue a command that names a log file. For example, the following command starts logging to a file on the workstation called dbgtool.log.

   ```
   SET LOG ON FILE dbgtool.log;
   ```

   This should be the first command that you enter from the terminal or include in your commands file.

• To use the Debug Tool in batch mode:

  a) If you plan to use the Language Environment run time TEST option to invoke the Debug Tool, compile the user-defined function with the TEST option.
   This places information in the program that the Debug Tool uses during a debugging session.

  b) Allocate a log data set to receive the output from the Debug Tool. Put a DD statement for the log data set in the startup procedure for the stored procedures address space.

  c) Enter commands in a data set that you want the Debug Tool to execute. Put a DD statement for that data set in the startup procedure for the stored procedures address space. To define the data set that contains the commands to the Debug Tool, specify its data set name or DD name in the TEST run time option.
   For example, this option tells the Debug Tool to look for the commands in the data set that is associated with DD name TESTDD:

   ```
   TEST(ALL,TESTDD,PROMPT,*)
   ```

   The first command in the commands data set should be:

   ```
   SET LOG ON FILE ddbname;
   ```

   This command directs output from your debugging session to the log data set you defined in step 2. For example, if you defined a log data set with DD name INSPLOG in the start-up procedure for the stored procedures address space, the first command should be:

   ```
   SET LOG ON FILE INSPLOG;
   ```

d) Invoke the Debug Tool.
   The following are two possible methods for invoking the Debug Tool:
- Specify the Language Environment run time TEST option. The most convenient place to do that is in the RUN OPTIONS parameter of CREATE FUNCTION or ALTER FUNCTION.

- Put CEETEST calls in the user-defined function source code. If you use this approach for an existing user-defined function, you must compile, link-edit, and bind the user-defined function again. Then you must issue the STOP FUNCTION SPECIFIC and START FUNCTION SPECIFIC commands to reload the user-defined function.

You can combine the Language Environment run time TEST option with CEETEST calls. For example, you might want to use TEST to name the commands data set but use CEETEST calls to control when the Debug Tool takes control.

You can combine the Language Environment run time TEST option with CEETEST calls. For example, you might want to use TEST to name the commands data set but use CEETEST calls to control when the Debug Tool takes control.

Related reference
CREATE FUNCTION (Db2 SQL)
Debug Tool for z/OS

Testing a user-defined function by routing the debugging messages to SYSPRINT

You can include simple print statements in your user-defined function code that you route to SYSPRINT. Then use System Display and Search Facility (SDSF) to examine the SYSPRINT contents while the WLM-established stored procedure address space is running.

About this task
You can serialize I/O by running the WLM-established stored procedure address space with NUMTCB=1.

Testing a user-defined function by using driver applications

You can write a small driver application that calls a user-defined function as a subprogram and passes the parameter list for the user-defined function. You can then test and debug the user-defined function as a normal Db2 application under TSO.

About this task
You can then use TSO TEST and other commonly used debugging tools.

Testing a user-defined function by using SQL INSERT statements

You can use SQL to insert debugging information into a Db2 table. This allows other machines in the network (such as workstations) to easily access the data in the table by using DRDA access.

About this task
Db2 discards the debugging information if the application executes the ROLLBACK statement. To prevent the loss of the debugging data, code the calling application so that it retrieves the diagnostic data before executing the ROLLBACK statement.
Debugging stored procedures

When debugging stored procedures, you might need to use different techniques than you would use for regular application programs. For example, some commonly used debugging tools, such as TSO TEST, are not available in the environment where stored procedures run.

Procedure

To debug a stored procedure, perform one or more of the following actions:

- Take one or more of the following general actions, which are appropriate in many situations with stored procedures:
  - Ensure that all stored procedures are written to handle any SQL errors.
  - Debug stored procedures as stand-alone programs on a workstation.
    If you have debugging tools on a workstation, consider doing most of your development and testing on a workstation before installing a stored procedure on z/OS. This technique results in very little debugging activity on z/OS.
  - Record stored procedure debugging messages to a disk file or JES spool file.
  - Store debugging information in a table. This technique is especially useful for remote stored procedures.
  - Use the DISPLAY command to view information about particular stored procedures, including statistics and thread information.
  - In the stored procedure that you are debugging, issue DISPLAY commands. You can view the DISPLAY results in the SDSF output. The DISPLAY results can help you find information about the started task that is associated with the address space for the WLM application environment.
  - If necessary, use the STOP PROCEDURE command to stop calls to one or more problematic stored procedures. You can restart them later.
  - If your stored procedures address space has the CEEDUMP data set allocated, look at the diagnostic information in the CEEDUMP output.
  - For COBOL, C, and C++ stored procedures, use the Debug Tool for z/OS.
  - For COBOL stored procedures, compile the stored procedure with the option TEST(SYM) if you want a formatted local variable dump to be included in the CEEDUMP output.
  - For native SQL procedures, external SQL procedures, and Java stored procedures, use the Unified Debugger.
  - For external stored procedures, consider taking one or both of the following actions:
    - Use a driver application.
    - Create or alter the stored procedure definition to include the PARAMETER STYLE SQL option. This option enables the stored procedure to share any error information with the calling application. Ensure that your procedure follows linkage conventions for stored procedures.
  - If you changed a stored procedure or a startup JCL procedure for a WLM application environment, determine whether you need to refresh the WLM environment. You must refresh the WLM environment before certain stored procedure changes take effect.

Related tasks

Handling SQL conditions in an SQL procedure
Displaying information about stored procedures with Db2 commands (Db2 Administration Guide)
Refreshing WLM application environments for stored procedures (Db2 Administration Guide)
Implementing Db2 stored procedures (Db2 Administration Guide)

Related reference

Linkage conventions for external stored procedures
The linkage convention for a stored procedure can be either GENERAL, GENERAL WITH NULLS, or SQL. These linkage conventions apply to only external stored procedures.

-START PROCEDURE (Db2) (Db2 Commands)
-STOP PROCEDURE (Db2) (Db2 Commands)

Related information
Db2 for z/OS Stored Procedures: Through the CALL and Beyond (IBM Redbooks)

Debugging stored procedures with the Debug Tool and IBM VisualAge COBOL

If you have VisualAge® COBOL installed on your workstation and the Debug Tool installed on your z/OS system, you can use the VisualAge COBOL Edit/Compile/Debug component with the Debug Tool to debug COBOL stored procedures that run in a WLM-established stored procedures address space.

About this task
Before you begin debugging, write your COBOL stored procedure and set up the WLM environment.

Procedure
To debug with the Debug Tool and IBM VisualAge COBOL:
1. When you compile the stored procedure, specify the TEST and SOURCE options. Ensure that the source listing is stored in a permanent data set. VisualAge COBOL displays the source listing during the debug session.
2. When you define the stored procedure, include run time option TEST with the suboption VADTCPPIP&ipaddr in your RUN OPTIONS argument.
   VADTCPPIP& tells the Debug Tool that it is interfacing with a workstation that runs VisualAge COBOL and is configured for TCP/IP communication with your z/OS system. ipaddr is the IP address of the workstation on which you display your debug information. For example, the RUN OPTIONS value in the following stored procedure definition indicates that debug information should go to the workstation with IP address 9.63.51.17:

   ```
   CREATE PROCEDURE WLMCOB
   (IN INTEGER, INOUT VARCHAR(3000), INOUT INTEGER)
   MODIFIES SQL DATA
   LANGUAGE COBOL EXTERNAL
   PROGRAM TYPE MAIN
   WLM ENVIRONMENT WLMENV1
   RUN OPTIONS 'POSIX(ON),TEST(,,,VADTCPPIP&9.63.51.17:*)'
   ```
3. In the JCL startup procedure for WLM-established stored procedures address space, add the data set name of the Debug Tool load library to the STEPLIB concatenation. For example, suppose that ENV1PROC is the JCL procedure for application environment WLMENV1. The modified JCL for ENV1PROC might look like this:

   ```
   //DSNWLM PROC RGN=0K,APPLENV=WLMENV1,DB2SSN=DSN,NUMTCB=8
   //IEFPROC EXEC PGM=DSNX9WLM,REGION=&RGN,TIME=NOLIMIT,
   // STEPLIB DD DISP=SHR,DSN=DSN1210.RUNLIB_LOAD
   // DD DISP=SHR,DSN=CEE.SCEERUN
   // DD DISP=SHR,DSN=DSN1210.SDSNLOAD
   // DD DISP=SHR,DSN=EQA.W.SEQAMOD <= DEBUG TOOL
   ```
4. On the workstation, start the VisualAge Remote Debugger daemon.
   This daemon waits for incoming requests from TCP/IP.
5. Call the stored procedure.
   When the stored procedure starts, a window that contains the debug session is displayed on the workstation. You can then execute Debug Tool commands to debug the stored procedure.

Related reference
Debug Tool for z/OS
Debugging a C language stored procedure with the Debug Tool and C/C++ Productivity Tools for z/OS

You can debug a C or C++ stored procedure that runs in a WLM-established stored procedures address space. You must have the C/C++ Productivity Tools for z/OS installed on your workstation and the Debug Tool installed on your z/OS system.

About this task

The code against which you run the debug tools is the C source program that is produced by the program preparation process for the stored procedure.

Before you begin debugging, write your C++ stored procedure and set up the WLM environment.

Procedure

To test the stored procedure with the Distributed Debugger feature of the C/C++ Productivity Tools for z/OS and the Debug Tool:

1. When you define the stored procedure, include run time option TEST with the suboption VADTCPIP&ipaddr in your RUN OPTIONS argument.
   
   VADTCPIP& tells the Debug Tool that it is interfacing with a workstation that runs VisualAge C++ and is configured for TCP/IP communication with your z/OS system. ipaddr is the IP address of the workstation on which you display your debug information. For example, this RUN OPTIONS value in a stored procedure definition indicates that debug information should go to the workstation with IP address 9.63.51.17:

   ```
   RUN OPTIONS 'POSIX(ON),TEST(,,,VADTCPIP&9.63.51.17:*)' 
   ```

2. Precompile the stored procedure. Ensure that the modified source program that is the output from the precompile step is in a permanent, catalogued data set.

3. Compile the output from the precompile step. Specify the TEST, SOURCE, and OPT(0) compiler options.

4. In the JCL startup procedure for the stored procedures address space, add the data set name of the Debug Tool load library to the STEPLIB concatenation. For example, suppose that ENV1PROC is the JCL procedure for application environment WLMENV1. The modified JCL for ENV1PROC might look like this:

   ```
   //DSNWLM   PROC RGN=0K,APPLENV=WLMENV1,DB2SSN=DSN,NUMTCB=8
   //IEFPROC EXEC PGM=DSNX9WLM,REGION=&RGN,TIME=NOLIMIT,
   //      PARM='&DB2SSN,&NUMTCB,&APPLENV'
   //STEPLIB   DD  DISP=SHR,DSN=DSN1210.RUNLIB.LOAD
   //         DD  DISP=SHR,DSN=CEE.SCEERUN
   //         DD  DISP=SHR,DSN=DSN1210.SDSNLOAD
   //         DD  DISP=SHR,DSN=EQAW.SEQAMOD <== DEBUG TOOL
   ```

5. On the workstation, start the Distributed Debugger daemon.
   This daemon waits for incoming requests from TCP/IP.

6. Call the stored procedure.
   When the stored procedure starts, a window that contains the debug session is displayed on the workstation. You can then execute Debug Tool commands to debug the stored procedure.

Related reference

- Debug Tool for z/OS
Debugging stored procedures by using the Unified Debugger

You can use the Unified Debugger to remotely debug native SQL procedures, external SQL procedures, and Java stored procedures that execute on Db2 for z/OS servers. The Unified Debugger also supports debugging nested stored procedure calls.

About this task

With the Unified Debugger, you can observe the execution of the procedure code, set breakpoints for lines, and view or modify variable values.

Procedure

To debug stored procedures by using the Unified Debugger:

1. Set up the Unified Debugger by performing the following steps:
   a) Ensure that job DSNTIJRT successfully created the stored procedures that provide server support for the Unified Debugger. This job is run during the installation and migration process. The stored procedures that this job creates must run in WLM environments.
      
      **Recommendation:** Initially, define and use the Db2 core WLM environment DSNWLM_GENERAL to run the SYSPROC.DBG_RUNSESSIONMANAGER stored procedure and core WLM environment DSNWLM_DEBUGGER to run the other stored procedures for Unified debugger.
   b) Define the debug mode characteristics for the stored procedure that you want to debug by completing one of the following actions:
      - For a native SQL procedure, define the procedure with the ALLOW DEBUG MODE option and the WLM ENVIRONMENT FOR DEBUG MODE option. If the procedure already exists, you can use the ALTER PROCEDURE statement to specify these options.
      - For an external SQL procedure, use DSNTPSMP or IBM Data Studio to build the SQL procedure with the BUILD_DEBUG option.
      - For a Java stored procedure, define the procedure with the ALLOW DEBUG MODE option, select an appropriate WLM environment for Java debugging, and compile the Java code with the -G option.
   c) Grant the DEBUGSESSION privilege to the user who runs the debug client.

2. Include breakpoints in your routines or executable files.

3. Follow the instructions for debugging stored procedures in the information for IBM Data Studio.

Related concepts

Java stored procedures and user-defined functions (Db2 Application Programming for Java)

Related tasks

Creating an external SQL procedure by using DSNTPSMP

The SQL procedure processor, DSNTPSMP, is one of several methods that you can use to create and prepare an external SQL procedure. DSNTPSMP is a REXX stored procedure that you can invoke from your application program.

Developing database routines (IBM Data Studio, IBM Optim Database Administrator, IBM infoSphere Data Architect, IBM Optim Development Studio)

Installing the Unified Debugger session manager on a z/OS system (Db2 Installation and Migration)

Related reference

Sample programs to help you prepare and run external SQL procedures

Db2 provides sample jobs to help you prepare and run external SQL procedures. All samples are in data set DSN1210.SDSNSAMP. Before you can run the samples, you must customize them for your installation.

ALTER PROCEDURE (SQL - native) (Db2 SQL)
CREATE PROCEDURE (SQL - native) (Db2 SQL)

Related information

Db2 for z/OS Stored Procedures: Through the CALL and Beyond (IBM Redbooks)
Debugging stored procedures with the Debug Tool for z/OS

You can use the Debug Tool to test z/OS stored procedures that are written in any of the compiled languages that the Debug Tool supports. You can test these stored procedures either interactively or in batch mode.

About this task

Using Debug Tool interactively: To test a stored procedure interactively using the Debug Tool, you must have the Debug Tool installed on the z/OS system where the stored procedure runs.

Procedure

To debug your stored procedure using the Debug Tool:

1. Compile the stored procedure with option TEST. This places information in the program that the Debug Tool uses during a debugging session.

2. Invoke the Debug Tool. One way to do that is to specify the Language Environment run time option TEST. The TEST option controls when and how the Debug Tool is invoked. The most convenient place to specify run time options is in the RUN OPTIONS parameter of the CREATE PROCEDURE or ALTER PROCEDURE statement for the stored procedure.

For example, you can code the TEST option using the following parameters:

```
TEST(ALL,*,PROMPT,JBJONES%SESSNA:)
```

The following table lists the effects that each parameter has on the Debug Tool:

<table>
<thead>
<tr>
<th>Parameter value</th>
<th>Effect on the Debug Tool</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALL</td>
<td>The Debug Tool gains control when an attention interrupt, ABEND, or program or Language Environment condition of Severity 1 and above occurs. Debug commands will be entered from the terminal.</td>
</tr>
<tr>
<td>PROMPT</td>
<td>The Debug Tool is invoked immediately after Language Environment initialization.</td>
</tr>
<tr>
<td>JBJONES%SESSNA:</td>
<td>The Debug Tool initiates a session on a workstation identified to APPC/MVS as JBJONES with a session ID of SESSNA.</td>
</tr>
</tbody>
</table>

3. If you want to save the output from your debugging session, issue the following command:

```
SET LOG ON FILE dbgtool.log;
```

This command saves a log of your debugging session to a file on the workstation called dbgtool.log. This should be the first command that you enter from the terminal or include in your commands file.

Results

Using Debug Tool in batch mode: To test your stored procedure in batch mode, you must have the Debug Tool installed on the z/OS system where the stored procedure runs. To debug your stored procedure in batch mode using the Debug Tool, complete the following steps:

- Compile the stored procedure with option TEST, if you plan to use the Language Environment run time option TEST to invoke the Debug Tool. This places information in the program that the Debug Tool uses during a debugging session.
• Allocate a log data set to receive the output from the Debug Tool. Put a DD statement for the log data set in the start-up procedure for the stored procedures address space.

• Enter commands in a data set that you want the Debug Tool to execute. Put a DD statement for that data set in the start-up procedure for the stored procedures address space. To define the commands data set to the Debug Tool, specify the commands data set name or DD name in the TEST run time option. For example, to specify that the Debug Tool use the commands that are in the data set that is associated with the DD name TESTDD, include the following parameter in the TEST option:

  TEST(ALL,TESTDD,PROMPT,*)

The first command in the commands data set should be:

  SET LOG ON FILE ddname;

This command directs output from your debugging session to the log data set that you defined in the previous step. For example, if you defined a log data set with DD name INSPLOG in the stored procedures address space start-up procedure, the first command should be the following command:

  SET LOG ON FILE INSPLOG;

• Invoke the Debug Tool. The following are two possible methods for invoking the Debug Tool:

  – Specify the run time option TEST. The most convenient place to do that is in the RUN OPTIONS parameter of the CREATE PROCEDURE or ALTER PROCEDURE statement for the stored procedure.

  – Put CEETEST calls in the stored procedure source code. If you use this approach for an existing stored procedure, you must recompile, re-link, and bind it, and issue the STOP PROCEDURE and START PROCEDURE commands to reload the stored procedure.

  You can combine the run time option TEST with CEETEST calls. For example, you might want to use TEST to name the commands data set but use CEETEST calls to control when the Debug Tool takes control.

Related reference

Debug Tool for z/OS

Recording stored procedure debugging messages in a file

You can debug external stored procedures and external SQL procedures by recording debugging messages in a disk file or in a JES spool file. You cannot use this debugging technique for native SQL procedures or Java stored procedures.

Procedure

To record stored procedure debugging messages in a file:

  1. Specify the Language Environment (LE) MSGFILE run time option for the stored procedure. This option identifies where LE is to write the debugging messages. To specify this option, include the RUN OPTIONS clause in either the CREATE PROCEDURE statement or an ALTER PROCEDURE statement.

     Specify the following MSGFILE parameters:

     • Use the first MSGFILE parameter to specify the JCL DD statement that identifies the data set for the debugging messages. You can direct debugging messages to a disk file or JES spool file. To prevent multiple procedures from sharing a data set, ensure that you specify a unique DD statement.

     • Use the ENQ option to serialize I/O to the message file. This action is necessary, because multiple TCBs can be active in the stored procedure address space. Alternatively, if you debug your applications infrequently or on a Db2 test system, you can serialize I/O by temporarily running the stored procedures address space with NUMTCB=1 in the stored procedures address space start-up procedure.

     2. For each instance of MSGFILE that you specify, add a DD statement to the JCL procedure that is used to start the stored procedures address space.
Driver applications for debugging procedures

You can write a small driver application that calls the stored procedure as a subprogram and passes the parameter list that the stored procedure supports. You can then test and debug the stored procedure as a normal Db2 application under TSO.

Using this method, you can use TSO TEST and other commonly used debugging tools.

Restriction: You cannot use this technique for SQL procedures

Db2 tables that contain debugging information

You can use SQL statements to insert debugging information into a Db2 table. Inserting this information into a table enables other machines in the network (such as a workstation) to easily access the data in the table by using DRDA access.

Db2 discards the debugging information if the application executes the ROLLBACK statement. To prevent the loss of the debugging data, code the calling application so that it retrieves the diagnostic data before executing the ROLLBACK statement.

Debugging an application program

Many sites have guidelines regarding what to do if a program abnormally terminates.

About this task

For information about the compiler or assembler test facilities, see the publications for the compiler or CODE/370. The compiler publications include information about the appropriate debugger for the language you are using.

You can also use ISPF Dialog Test to debug your program. You can run all or portions of your application, examine the results, make changes, and rerun it.

Related reference
Dialog test (option 7) (z/OS ISPF User’s Guide Vol II)

Locating the problem in an application

If your program does not run correctly, you need to isolate the problem. You should check several items.

About this task

Those items are:

• Output from the precompiler, which consists of errors and warnings. Ensure that you have resolved all errors and warnings.
• Output from the compiler or assembler. Ensure that you have resolved all error messages.
• Output from the linkage editor.
  – Have you resolved all external references?
  – Have you included all necessary modules in the correct order?
– Did you include the correct language interface module? The correct language interface module is:
  - DSNELI or DSNULI for TSO
  - DFSLI000 for IMS
  - DSNCLI or DSNULI for CICS
  - DSNALI or DSNULI for the call attachment facility
  - DSNRLI or DSNULI for the Resource Recovery Services attachment facility
– Did you specify the correct entry point to your program?
  • Output from the bind process.
    – Have you resolved all error messages?
    – Did you specify a plan name? If not, the bind process assumes that you want to process the DBRM for diagnostic purposes, but that you do not want to produce an application plan.
    – Have you specified all the packages that are associated with the programs that make up the application and their partitioned data set (PDS) names in a single application plan?
  • Your JCL.

IMS
– If you are using IMS, have you included the DL/I option statement in the correct format?
– Have you included the region size parameter in the EXEC statement? Does it specify a region size that is large enough for the required storage for the Db2 interface, the TSO, IMS, or CICS system, and your program?
– Have you included the names of all data sets (Db2 and non_Db2) that the program requires?
  • Your program.

You can also use dumps to help localize problems in your program. For example, one of the more common error situations occurs when your program is running and you receive a message that it abended. In this situation, your test procedure might be to capture a TSO dump. To do so, you must allocate a SYSUDUMP or SYSABEND dump data set before calling Db2. When you press the ENTER key (after the error message and READY message), the system requests a dump. You then need to use the FREE command to deallocate the dump data set.

Error and warning messages from the precompiler
In some circumstances, the statements that the Db2 precompiler generates might produce compiler or assembly error messages. You need to know why the messages occur when you compile Db2-produced source statements.

SYSTERM output from the precompiler
The SYSTERM output provides a brief summary of the results from the precompiler, all error messages that the precompiler generated, and the statement that is in error, when possible.

The Db2 precompiler provides SYSTERM output when you allocate the DD name SYSTERM. If you use the program preparation panels to prepare and run your program, DB2I allocates SYSTERM according to the TERM option that you specify.

You can use the line number that is provided in each error message in the SYSTERM output to locate the failing source statement.

Figure 71 on page 975 shows the format of SYSTERM output.
Figure 71. Db2 precompiler SYSTEP output

Notes:
1. Error message.
2. Source SQL statement.
3. Summary statements of source statistics.
4. Summary statement of the number of errors that were detected.
5. Summary statement that indicates the number of errors that were detected but not printed. This situation might occur if you specify a FLAG option other than I.
6. Storage requirement statement that indicates how many bytes of working storage that the Db2 precompiler actually used to process your source statements. That value helps you determine the storage allocation requirements for your program.
7. Return code: 0 = success, 4 = warning, 8 = error, 12 = severe error, and 16 = unrecoverable error.

SYSPRINT output from the precompiler

SYSPRINT output from the Db2 precompiler shows the results of the precompile operation. This output can also include a list of the options that were used, a source code listing, and a host variable cross-reference listing.

When you use the program preparation panels to prepare and run your program, Db2 allocates SYSPRINT according to TERM option that you specify (on line 12 of the PROGRAM PREPARATION: COMPILE, PRELINK, LINK, AND RUN panel). As an alternative, when you use the DSNH command procedure (CLIST), you can specify PRINTTERM to obtain SYSPRINT output at your terminal, or you can specify PRINT(qualifier) to place the SYSPRINT output into a data set named authorizationID.qualifier.PCLIST. Assuming that you do not specify PRINT as LEAVE, NONE, or TERM, Db2 issues a message when the precompiler finishes, telling you where to find your precompiler listings. This helps you locate your diagnostics quickly and easily.

The SYSPRINT output can provide information about your precompiled source module if you specify the options SOURCE and XREF when you start the Db2 precompiler.

The format of SYSPRINT output is as follows:

- A list of the Db2 precompiler options that are in effect during the precompilation (if you did not specify NOOPTIONS).
- A list of your source statements (only if you specified the SOURCE option). An example is shown in Figure 72 on page 976.
- A list of the symbolic names used in SQL statements (this listing appears only if you specify the XREF option). An example is show in Figure 73 on page 977.
- A summary of the errors that are detected by the Db2 precompiler and a list of the error messages that are generated by the precompiler. An example is shown in
The following code shows an example list of Db2 precompiler options as it is displayed in the SYSPRINT output.

```
DB2 SQL PRECOMPILER VERSION 11 REL. 1.0
OPTIONS SPECIFIED: HOST(PLI),SOURCE,XREF,STDSQL(NO),TWOPASS
DSNHDECP LOADED FROM - (USER99.RELM.TESTLIB(DSNHDECP))
OPTIONS USED - SPECIFIED OR DEFAULTED
APOST
APOSTSQL
ATTACH(TSO)
CCSID(37)
CONNECT(2)
DEC(15)
FLAG(I)
FLOAT(S390)
HOST(PLI)
LINECOUNT(60)
MARGINS(2,72)
NEWFUN(V11)
OPTIONS
PERIOD
SOURCE
SQL(DB2)
STDSQL(NO)
TWOPASS
XREF
```

**Notes:**

1. This section lists the options that are specified at precompilation time. This list does not appear if one of the precompiler option is NOOPTIONS.

2. This section lists the options that are in effect, including defaults, forced values, and options that you specified. The Db2 precompiler overrides or ignores any options that you specify that are inappropriate for the host language.

The following figure shows an example list of source statements as it is displayed in the SYSPRINT output.

```
1        TMN5P40:PROCEDURE OPTIONS (MAIN) ;                       00000100
2        /**********************************************************00000200
3        *      program description and prologue                       00000300
⋮
1324        /*********************************************************/     00132400
1325        /*  GET INFORMATION ABOUT THE PROJECT FROM THE   */     00132500
1326        /*  PROJECT TABLE.                               */     00132600
1327        /*********************************************************/     00132700
1328               EXEC SQL SELECT ACTNO, PREQPROJ, PREQACT         00132800
1329                    INTO PROJ_DATA                              00132900
1330                    FROM TPREREQ                                00133000
1331                         WHERE PROJNO = :PROJ_NO;               00133100
1332                                                                00133200
1333        /*********************************************************/     00133300
1334        /*  PROJECT IS FINISHED. DELETE IT.              */     00133400
1335        /*********************************************************/     00133500
1336                                                                00133600
1337                EXEC SQL DELETE FROM PROJ                       00133700
1338                     WHERE PROJNO = :PROJ_NO;                   00133800
⋮
1523        END;                                                    00152300
```

*Figure 72. Db2 precompiler SYSPRINT output: Source statements section*

**Notes:**

- The left column of sequence numbers, which the Db2 precompiler generates, is for use with the symbol cross-reference listing, the precompiler error messages, and the BIND error messages.
- The right column shows sequence numbers that come from the sequence numbers that are supplied with your source statements.

The following figure shows an example list of symbolic names as it is displayed in the SYSPRINT output.
Figure 73. Db2 precompiler SYSPRINT output: Symbol cross-reference section

Notes:

DATA NAMES
Identifies the symbolic names that are used in source statements. Names enclosed in double quotation marks ("\) or apostrophes (\') are names of SQL entities such as tables, columns, and authorization IDs. Other names are host variables.

DEFN
Is the number of the line that the precompiler generates to define the name. **** means that the object was not defined, or the precompiler did not recognize the declarations.

REFERENCE
Contains two kinds of information: the symbolic name, which the source program defines, and which lines refer to the symbolic name. If the symbolic name refers to a valid host variable, the list also identifies the data type or the word STRUCTURE.

The following code shows an example summary report of errors as it is displayed in the SYSPRINT output.

Notes:
1. Summary statement that indicates the number of source lines.
2. Summary statement that indicates the number of symbolic names in the symbol table (SQL names and host names).
3. Storage requirement statement that indicates the number of bytes for the symbol table.
4. Summary statement that indicates the number of messages that are printed.
5. Summary statement that indicates the number of errors that are detected but not printed. You might get this statement if you specify the option FLAG.
6. Storage requirement statement that indicates the number of bytes of working storage that are actually used by the Db2 precompiler to process your source statements.
7. Return code 0 = success, 4 = warning, 8 = error, 12 = severe error, and 16 = unrecoverable error.
8. Error messages (this example detects only one error).

**Techniques for debugging programs in TSO**

Documenting the errors that are identified during testing of a TSO application helps you investigate and correct problems in the program.

The following information can be useful:

- The application plan name of the program
- The input data that is being processed
- The failing SQL statement and its function
- The contents of the SQLCA (SQL communication area) and, if your program accepts dynamic SQL statements, the SQLDA (SQL descriptor area)
- The date and time of day
- The abend code and any error messages

When your program encounters an error that does not result in an abend, it can pass all the required error information to a standard error routine. Online programs might also send an error message to the terminal.

**The TSO TEST command**

The TSO TEST command is especially useful for debugging assembler programs.

The following example is a command procedure (CLIST) that runs a Db2 application named MYPROG under TSO TEST, and sets an address stop at the entry to the program. The Db2 subsystem name in this example is DB4.

```plaintext
PROC 0  
TEST 'prefix.SDSNLOAD(DSN)' CP  
DSN SYSTEM(DB4)  
AT MYPROG.MYPROG.+0 DEFER  
GO  
RUN PROGRAM(MYPROG) LIBRARY('L186331.RUNLIB.LOAD(MYPROG)')
```

**Related reference**

TEST command (TSO/E Command Reference)

**Techniques for debugging programs in IMS**

Documenting the errors that are identified during testing of an IMS application helps you investigate and correct problems in the program.

The following information can be useful:

- The application plan name for the program
- The input message that is being processed
- The name of the originating logical terminal
- The failing statement and its function
- The contents of the SQLCA (SQL communication area) and, if your program accepts dynamic SQL statements, the SQLDA (SQL descriptor area)
- The date and time of day
- The PSB name for the program
- The transaction code that the program was processing
- The call function (that is, the name of a DL/I function)
- The contents of the PCB that the program call refers to
- If a DL/I database call was running, the SSAs, if any, that the call used
• The abend completion code, abend reason code, and any dump error messages

When your program encounters an error, it can pass all the required error information to a standard error routine. Online programs can also send an error message to the originating logical terminal.

An interactive program also can send a message to the master terminal operator (MTO) operator giving information about the termination of the program. To do that, the program places the logical terminal name of the master terminal in an express PCB and issues one or more ISRT calls.

Some organizations run a BMP at the end of the day to list all the errors that occurred during the day. If your organization does this, you can send a message by using an express PCB that has its destination set for that BMP.

**Batch Terminal Simulator:** The Batch Terminal Simulator (BTS) enables you to test IMS application programs. BTS traces application program DL/I calls and SQL statements, and it simulates data communication functions. It can make a TSO terminal appear as an IMS terminal to the terminal operator, which enables the user to interact with the application as though it were an online application. The user can use any application program that is under the user’s control to access any database (whether DL/I or Db2) that is under the user’s control. Access to Db2 databases requires BTS to operate in batch BMP or TSO BMP mode.

**Techniques for debugging programs in CICS**

Documenting the errors that are identified during testing of a CICS application helps you investigate and correct problems in the program.

The following information can be useful:

• The application plan name of the program
• The input data that is being processed
• The ID of the originating logical terminal
• The failing SQL statement and its function
• The contents of the SQLCA (SQL communication area) and, if your program accepts dynamic SQL statements, the SQLDA (SQL descriptor area)
• The date and time of day
• Data that is peculiar to CICS that you should record
• Abend code and dump error messages
• Transaction dump, if produced

Using CICS facilities, you can have a printed error record; you can also print the SQLCA and SQLDA contents.

**Debugging aids for CICS**

CICS provides the following aids to the testing, monitoring, and debugging of application programs:

• **Execution (Command Level) Diagnostic Facility (EDF).** EDF shows CICS commands for all releases of CICS.
• **Abend recovery.** You can use the HANDLE ABEND command to deal with abend conditions. You can use the ABEND command to cause a task to abend.
• **Trace facility.** A trace table can contain entries showing the execution of various CICS commands, SQL statements, and entries that are generated by application programs; you can have these entries written to main storage and, optionally, to an auxiliary storage device.
• **Dump facility.** You can specify areas of main storage to dump onto a sequential data set, either tape or disk, for subsequent offline formatting and printing with a CICS utility program.
- **Journals.** For statistical or monitoring purposes, facilities can create entries in special data sets called journals. The system log is a journal.

- **Recovery.** When an abend occurs, CICS restores certain resources to their original state so that the operator can easily resubmit a transaction for restart. You can use the SYNCPOINT command to subdivide a program so that you only need to resubmit the uncompleted part of a transaction.

### CICS execution diagnostic facility

The CICS execution diagnostic facility (EDF) traces SQL statements in an interactive debugging mode, enabling application programmers to test and debug programs online without changing the program or the program preparation procedure.

EDF intercepts the running application program at various points and displays helpful information about the statement type, input and output variables, and any error conditions after the statement executes.

It also displays any screens that the application program sends, so that you can converse with the application program during testing just as a user would on a production system.

EDF displays essential information before and after an SQL statement runs, while the task is in EDF mode. This can be a significant aid in debugging CICS transaction programs that contains SQL statements. The SQL information that EDF displays is helpful for debugging programs and for error analysis after an SQL error or warning. Using this facility reduces the amount of work that you need to do to write special error handlers.

### EDF before execution

The following figure shows an example of an EDF screen before it executes an SQL statement. The names of the key information fields on this panel are in **boldface**.

![Figure 74: EDF screen before a Db2 SQL statement](image)

The Db2 SQL information in this screen is as follows:

- **EXEC SQL statement type**

  This is the type of SQL statement to execute. The SQL statement can be any valid SQL statement.

- **DBRM=dbrm name**

  The name of the database request module (DBRM) that is currently processing. The DBRM, created by the Db2 precompiler, contains information about an SQL statement.

- **STMT=statement number**
This is the Db2 precompiler-generated statement number. The source and error message listings from the precompiler use this statement number, and you can use the statement number to determine which statement is processing. This number is a source line counter that includes host language statements. A statement number that is greater than 32767 displays as 0.

- **SECT=section number**
  The section number of the plan that the SQL statement uses.

**SQL statements that contain input host variables**

The IVAR (input host variables) section and its attendant fields appear only when the executing statement contains input host variables.

The host variables section includes the variables from predicates, the values used for inserting or updating, and the text of dynamic SQL statements that are being prepared. The address of the input variable is AT X'nnnnnnnn'.

Additional host variable information:

- **TYPE=data type**
  Specifies the data type for this host variable. The basic data types include character string, graphic string, binary integer, floating-point, decimal, date, time, and timestamp.

- **LEN=length**
  Specifies the length of the host variable.

- **IND=indicator variable status number**
  Specifies the indicator variable that is associated with this particular host variable. A value of zero indicates that no indicator variable exists. If the value for the selected column is null, Db2 puts a negative value in the indicator variable for this host variable.

- **DATA=host variable data**
  Specifies the data, displayed in hexadecimal format, that is associated with this host variable. If the data exceeds what can display on a single line, three periods (...) appear at the far right to indicate that more data is present.

**EDF after execution**

The following figure shows an example of the first EDF screen that is displayed after the executing an SQL statement. The names of the key information fields on this panel are in **boldface**.
Figure 75. EDF screen after a Db2 SQL statement

The Db2 SQL information in this screen is as follows:

- **P.AUTH=primary authorization ID**
  
  The primary Db2 authorization ID.

- **S.AUTH=secondary authorization ID**
  
  The secondary authorization ID. If the RACF list of group options is not active, Db2 uses the connected group name that the CICS attachment facility supplies as the secondary authorization ID. If the RACF list of group options is active, Db2 ignores the connected group name that the CICS attachment facility supplies, but the value is displayed in the Db2 list of secondary authorization IDs.

- **PLAN=plan name**
  
  The name of the plan that is currently running. The PLAN represents the control structure that is produced during the bind process and that is used by Db2 to process SQL statements that are encountered while the application is running.

- **SQL Communication Area (SQLCA)**
  
  Information in the SQLCA. The SQLCA contains information about errors, if any occur. Db2 uses the SQLCA to give an application program information about the executing SQL statements.

Plus signs (+) on the left of the screen indicate that you can see additional EDF output by using PF keys to scroll the screen forward or back.

The OVAR (output host variables) section and its attendant fields are displayed only when the executing statement returns output host variables.

The following figure contains the rest of the EDF output for this example.
The attachment facility automatically displays SQL information while in the EDF mode. (You can start EDF as outlined in the appropriate CICS application programmer's reference manual.) If this information is not displayed, contact the person that is responsible for installing and migrating Db2.

Related concepts
Data types of columns
When you create a Db2 table, you define each column to have a specific data type. The data type of a column determines what you can and cannot do with the column.

Indicator variables, arrays, and structures
An indicator variable is associated with a particular host variable. Each indicator variable contains a small integer value that indicates some information about the associated host variable. Indicator arrays and structures serve the same purpose for host-variable arrays and structures.

Related information
CICS debugging aids (CICS Transaction Server for z/OS)

Finding a violated referential or check constraint
When you receive an SQL error because of a constraint violation, look at the SQLCA for specific information.

About this task

Question: When a referential or check constraint has been violated, how do I determine which one it is?

Answer: When you receive an SQL error because of a constraint violation, print out the SQLCA. You can use the DSNTIAR routine to format the SQLCA for you. Check the SQL error message insertion text (SQLERRM) for the name of the constraint. For information about possible violations, see SQLCODEs -530 through -548.

Related concepts
SQL error codes (Db2 Codes)

Related tasks
Displaying SQLCA fields by calling DSNTIAR
If you use the SQLCA to check whether an SQL statement executed successfully, your program needs to read the data in the appropriate SQLCA fields. One easy way to read these fields is to use the assembler subroutine DSNTIAR.
Chapter 12. Sample data and applications supplied with Db2 for z/OS

You can use sample applications that are included with Db2 for z/OS to learn about how to program applications that take advantage Db2 capabilities. Db2 also provides models for your own situations.

To prepare and run the supplied sample applications, use the JCL in prefix.SDSNSAMP as a model:

**Related reference**
Db2 sample tables (Introduction to Db2 for z/OS)

Db2 sample tables

Much of the Db2 information refers to or relies on the Db2 sample tables. As a group, the tables include information that describes employees, departments, projects, and activities, and they make up a sample application that exemplifies many of the features of Db2.

The sample storage group, databases, table spaces, tables, and views are created when you run the installation sample jobs DSNTEJ1 and DSNTEJ7. Db2 sample objects that include LOBs are created in job DSNTEJ7. All other sample objects are created in job DSNTEJ1. The CREATE INDEX statements for the sample tables are not shown here; they, too, are created by the DSNTEJ1 and DSNTEJ7 sample jobs.

Authorization on all sample objects is given to PUBLIC in order to make the sample programs easier to run. You can review the contents of any table by executing an SQL statement, for example SELECT * FROM DSN8C10.PROJ. For convenience in interpreting the examples, the department and employee tables are listed in full.

**Related concepts**
Phase 1: Creating and loading sample tables (Db2 Installation and Migration)

Activity table (DSN8C10.ACT)

The activity table describes the activities that can be performed during a project.

The activity table resides in database DSN8D12A and is created with the following statement:

```sql
CREATE TABLE DSN8C10.ACT
    (ACTNO SMALLINT NOT NULL,
     ACTKWD CHAR(6) NOT NULL,
     ACTDESC VARCHAR(20) NOT NULL,
     PRIMARY KEY (ACTNO)
    )
    IN DSN8D12A.DSN8S12P
    CCSID EBCDIC;
```

**Content of the activity table**

The following table shows the content of the columns in the activity table.

<table>
<thead>
<tr>
<th>Column</th>
<th>Column name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>ACTNO</td>
<td>Activity ID (the primary key)</td>
</tr>
</tbody>
</table>
Table 156. Columns of the activity table (continued)

<table>
<thead>
<tr>
<th>Column</th>
<th>Column name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>ACTKWD</td>
<td>Activity keyword (up to six characters)</td>
</tr>
<tr>
<td>3</td>
<td>ACTDESC</td>
<td>Activity description</td>
</tr>
</tbody>
</table>

The activity table has the following indexes.

Table 157. Indexes of the activity table

<table>
<thead>
<tr>
<th>Name</th>
<th>On column</th>
<th>Type of index</th>
</tr>
</thead>
<tbody>
<tr>
<td>DSN8C10.XACT1</td>
<td>ACTNO</td>
<td>Primary, ascending</td>
</tr>
<tr>
<td>DSN8C10.XACT2</td>
<td>ACTKWD</td>
<td>Unique, ascending</td>
</tr>
</tbody>
</table>

Relationship to other tables

The activity table is a parent table of the project activity table, through a foreign key on column ACTNO.

Department table (DSN8C10.DEPT)

The department table describes each department in the enterprise and identifies its manager and the department to which it reports.

The department table resides in table space DSN8D12A.DSN8S12D and is created with the following statement:

```sql
CREATE TABLE DSN8C10.DEPT
(DEPTNO    CHAR(3)           NOT NULL,
 DEPTNAME  VARCHAR(36)       NOT NULL,
 MGRNO     CHAR(6)                   ,
 ADMRDEPT  CHAR(3)           NOT NULL,
 LOCATION  CHAR(16)                  ,
 PRIMARY KEY (DEPTNO)                )
IN DSN8D12A.DSN8S12D
CCSID EBCDIC;
```

Because the department table is self-referencing, and also is part of a cycle of dependencies, its foreign keys must be added later with the following statements:

```sql
ALTER TABLE DSN8C10.DEPT
FOREIGN KEY RDD (ADMRDEPT) REFERENCES DSN8C10.DEPT
ON DELETE CASCADE;

ALTER TABLE DSN8C10.DEPT
FOREIGN KEY RDE (MGRNO) REFERENCES DSN8C10.EMP
ON DELETE SET NULL;
```

Content of the department table

The following table shows the content of the columns in the department table.

Table 158. Columns of the department table

<table>
<thead>
<tr>
<th>Column</th>
<th>Column name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>DEPTNO</td>
<td>Department ID, the primary key.</td>
</tr>
</tbody>
</table>
Table 158. Columns of the department table (continued)

<table>
<thead>
<tr>
<th>Column</th>
<th>Column name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>DEPTNAME</td>
<td>A name that describes the general activities of the department.</td>
</tr>
<tr>
<td>3</td>
<td>MGRNO</td>
<td>Employee number (EMPNO) of the department manager.</td>
</tr>
<tr>
<td>4</td>
<td>ADMRDEPT</td>
<td>ID of the department to which this department reports; the department at the highest level reports to itself.</td>
</tr>
<tr>
<td>5</td>
<td>LOCATION</td>
<td>The remote location name.</td>
</tr>
</tbody>
</table>

The following table shows the indexes of the department table.

Table 159. Indexes of the department table

<table>
<thead>
<tr>
<th>Name</th>
<th>On column</th>
<th>Type of index</th>
</tr>
</thead>
<tbody>
<tr>
<td>DSN8C10.XDEPT1</td>
<td>DEPTNO</td>
<td>Primary, ascending</td>
</tr>
<tr>
<td>DSN8C10.XDEPT2</td>
<td>MGRNO</td>
<td>Ascending</td>
</tr>
<tr>
<td>DSN8C10.XDEPT3</td>
<td>ADMRDEPT</td>
<td>Ascending</td>
</tr>
</tbody>
</table>

The following table shows the content of the department table.

Table 160. DSN8C10.DEPT: department table

<table>
<thead>
<tr>
<th>DEPTNO</th>
<th>DEPTNAME</th>
<th>MGRNO</th>
<th>ADMRDEPT</th>
<th>LOCATION</th>
</tr>
</thead>
<tbody>
<tr>
<td>A00</td>
<td>SPIFFY COMPUTER SERVICE DIV.</td>
<td>000010</td>
<td>A00</td>
<td>---------</td>
</tr>
<tr>
<td>B01</td>
<td>PLANNING</td>
<td>000020</td>
<td>A00</td>
<td>---------</td>
</tr>
<tr>
<td>C01</td>
<td>INFORMATION CENTER</td>
<td>000030</td>
<td>A00</td>
<td>---------</td>
</tr>
<tr>
<td>D01</td>
<td>DEVELOPMENT CENTER</td>
<td>------</td>
<td>A00</td>
<td>---------</td>
</tr>
<tr>
<td>E01</td>
<td>SUPPORT SERVICES</td>
<td>000050</td>
<td>A00</td>
<td>---------</td>
</tr>
<tr>
<td>D11</td>
<td>MANUFACTURING SYSTEMS</td>
<td>000060</td>
<td>D01</td>
<td>---------</td>
</tr>
<tr>
<td>D21</td>
<td>ADMINISTRATION SYSTEMS</td>
<td>000070</td>
<td>D01</td>
<td>---------</td>
</tr>
<tr>
<td>E11</td>
<td>OPERATIONS</td>
<td>000090</td>
<td>E01</td>
<td>---------</td>
</tr>
<tr>
<td>E21</td>
<td>SOFTWARE SUPPORT</td>
<td>000100</td>
<td>E01</td>
<td>---------</td>
</tr>
<tr>
<td>F22</td>
<td>BRANCH OFFICE F2</td>
<td>------</td>
<td>E01</td>
<td>---------</td>
</tr>
<tr>
<td>G22</td>
<td>BRANCH OFFICE G2</td>
<td>------</td>
<td>E01</td>
<td>---------</td>
</tr>
<tr>
<td>H22</td>
<td>BRANCH OFFICE H2</td>
<td>------</td>
<td>E01</td>
<td>---------</td>
</tr>
<tr>
<td>I22</td>
<td>BRANCH OFFICE I2</td>
<td>------</td>
<td>E01</td>
<td>---------</td>
</tr>
<tr>
<td>J22</td>
<td>BRANCH OFFICE J2</td>
<td>------</td>
<td>E01</td>
<td>---------</td>
</tr>
</tbody>
</table>

The LOCATION column contains null values until sample job DSNTEJ6 updates this column with the location name.

**Relationship to other tables**

The department table is self-referencing: the value of the administering department must be a valid department ID.

The department table is a parent table of the following:
• The employee table, through a foreign key on column WORKDEPT
• The project table, through a foreign key on column DEPTNO

The department table is a dependent of the employee table, through its foreign key on column MGRNO.

**Employee table (DSN8C10.EMP)**

The sample employee table identifies all employees by an employee number and lists basic personnel information.

The employee table resides in the partitioned table space DSN8D12A.DSN8S12E. Because this table has a foreign key that references DEPT, that table and the index on its primary key must be created first. Then EMP is created with the following statement:

```
CREATE TABLE DSN8C10.EMP
(EMPNO     CHAR(6)                          NOT NULL,
  FIRSTNME  VARCHAR(12)                      NOT NULL,
  MIDINIT   CHAR(1)                          NOT NULL,
  LASTNAME  VARCHAR(15)                      NOT NULL,
  WORKDEPT  CHAR(3)                                  ,
  PHONENO   CHAR(4)         CONSTRAINT NUMBER CHECK
  (PHONENO >= '0000' AND
   PHONENO <= '9999')                      ,
  HIREDATE  DATE                                     ,
  JOB       CHAR(8)                                  ,
  EDLEVEL   SMALLINT                                 ,
  SEX       CHAR(1)                                  ,
  BIRTHDATE DATE                                     ,
  SALARY    DECIMAL(9,2)                             ,
  BONUS     DECIMAL(9,2)                             ,
  COMM      DECIMAL(9,2)                             ,
  PRIMARY KEY (EMPNO)                                ,
  FOREIGN KEY RED (WORKDEPT) REFERENCES DSN8C10.DEPT
  ON DELETE SET NULL                       )
```

**Content of the employee table**

The following table shows the type of content of each of the columns in the employee table. The table has a check constraint, NUMBER, which checks that the four-digit phone number is in the numeric range 0000 to 9999.

<table>
<thead>
<tr>
<th>Column</th>
<th>Column name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>EMPNO</td>
<td>Employee number (the primary key)</td>
</tr>
<tr>
<td>2</td>
<td>FIRSTNME</td>
<td>First name of employee</td>
</tr>
<tr>
<td>3</td>
<td>MIDINIT</td>
<td>Middle initial of employee</td>
</tr>
<tr>
<td>4</td>
<td>LASTNAME</td>
<td>Last name of employee</td>
</tr>
<tr>
<td>5</td>
<td>WORKDEPT</td>
<td>ID of department in which the employee works</td>
</tr>
<tr>
<td>6</td>
<td>PHONENO</td>
<td>Employee telephone number</td>
</tr>
<tr>
<td>7</td>
<td>HIREDATE</td>
<td>Date of hire</td>
</tr>
<tr>
<td>8</td>
<td>JOB</td>
<td>Job held by the employee</td>
</tr>
<tr>
<td>9</td>
<td>EDLEVEL</td>
<td>Number of years of formal education</td>
</tr>
<tr>
<td>10</td>
<td>SEX</td>
<td>Sex of the employee (M or F)</td>
</tr>
</tbody>
</table>
Table 161. Columns of the employee table (continued)

<table>
<thead>
<tr>
<th>Column</th>
<th>Column name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td>BIRTHDATE</td>
<td>Date of birth</td>
</tr>
<tr>
<td>12</td>
<td>SALARY</td>
<td>Yearly salary in dollars</td>
</tr>
<tr>
<td>13</td>
<td>BONUS</td>
<td>Yearly bonus in dollars</td>
</tr>
<tr>
<td>14</td>
<td>COMM</td>
<td>Yearly commission in dollars</td>
</tr>
</tbody>
</table>

The following table shows the indexes of the employee table.

Table 162. Indexes of the employee table

<table>
<thead>
<tr>
<th>Name</th>
<th>On column</th>
<th>Type of index</th>
</tr>
</thead>
<tbody>
<tr>
<td>DSN8C10.XEMP1</td>
<td>EMPNO</td>
<td>Primary, partitioned, ascending</td>
</tr>
<tr>
<td>DSN8C10.XEMP2</td>
<td>WORKDEPT</td>
<td>Ascending</td>
</tr>
</tbody>
</table>

The following table shows the first half (left side) of the content of the employee table. (Table 164 on page 990 shows the remaining content (right side) of the employee table.)

Table 163. Left half of DSN8C10.EMP: employee table. Note that a blank in the MIDINIT column is an actual value of " " rather than null.

<table>
<thead>
<tr>
<th>EMPNO</th>
<th>FIRSTNAME</th>
<th>MIDINIT</th>
<th>LASTNAME</th>
<th>WORKDEPT</th>
<th>PHONENO</th>
<th>HIREDATE</th>
</tr>
</thead>
<tbody>
<tr>
<td>000010</td>
<td>CHRISTINE</td>
<td>I</td>
<td>HAAS</td>
<td>A00</td>
<td>3978</td>
<td>1965-01-01</td>
</tr>
<tr>
<td>000020</td>
<td>MICHAEL</td>
<td>L</td>
<td>THOMPSON</td>
<td>B01</td>
<td>3476</td>
<td>1973-10-10</td>
</tr>
<tr>
<td>000030</td>
<td>SALLY</td>
<td>A</td>
<td>Kwan</td>
<td>C01</td>
<td>4738</td>
<td>1975-04-05</td>
</tr>
<tr>
<td>000050</td>
<td>JOHN</td>
<td>B</td>
<td>GEYER</td>
<td>E01</td>
<td>6789</td>
<td>1949-08-17</td>
</tr>
<tr>
<td>000060</td>
<td>IRVING</td>
<td>F</td>
<td>STERN</td>
<td>D11</td>
<td>6423</td>
<td>1973-09-14</td>
</tr>
<tr>
<td>000070</td>
<td>EVA</td>
<td>D</td>
<td>PULASKI</td>
<td>D21</td>
<td>7831</td>
<td>1980-09-30</td>
</tr>
<tr>
<td>000090</td>
<td>EILEEN</td>
<td>W</td>
<td>HENDERSON</td>
<td>E11</td>
<td>5498</td>
<td>1970-08-15</td>
</tr>
<tr>
<td>000100</td>
<td>THEODORE</td>
<td>Q</td>
<td>SPENSER</td>
<td>E21</td>
<td>0972</td>
<td>1980-06-19</td>
</tr>
<tr>
<td>000110</td>
<td>VINCENZO</td>
<td>G</td>
<td>LUCCHESI</td>
<td>A00</td>
<td>3490</td>
<td>1958-05-16</td>
</tr>
<tr>
<td>000120</td>
<td>SEAN</td>
<td></td>
<td>O'CONNELL</td>
<td>A00</td>
<td>2167</td>
<td>1963-12-05</td>
</tr>
<tr>
<td>000130</td>
<td>DOLORES</td>
<td>M</td>
<td>QUINTANA</td>
<td>C01</td>
<td>4578</td>
<td>1971-07-28</td>
</tr>
<tr>
<td>000140</td>
<td>HEATHER</td>
<td>A</td>
<td>NICHOLLS</td>
<td>C01</td>
<td>1793</td>
<td>1976-12-15</td>
</tr>
<tr>
<td>000150</td>
<td>BRUCE</td>
<td></td>
<td>ADAMSON</td>
<td>D11</td>
<td>4510</td>
<td>1972-02-12</td>
</tr>
<tr>
<td>000160</td>
<td>ELIZABETH</td>
<td>R</td>
<td>PIANKA</td>
<td>D11</td>
<td>3782</td>
<td>1977-10-11</td>
</tr>
<tr>
<td>000170</td>
<td>MASATOSHI</td>
<td>J</td>
<td>YOSHIURA</td>
<td>D11</td>
<td>2890</td>
<td>1978-09-15</td>
</tr>
<tr>
<td>000180</td>
<td>MARILYN</td>
<td>S</td>
<td>SCOUTEN</td>
<td>D11</td>
<td>1682</td>
<td>1973-07-07</td>
</tr>
<tr>
<td>000190</td>
<td>JAMES</td>
<td>H</td>
<td>WALKER</td>
<td>D11</td>
<td>2986</td>
<td>1974-07-26</td>
</tr>
<tr>
<td>000200</td>
<td>DAVID</td>
<td></td>
<td>BROWN</td>
<td>D11</td>
<td>4501</td>
<td>1966-03-03</td>
</tr>
<tr>
<td>000210</td>
<td>WILLIAM</td>
<td>T</td>
<td>JONES</td>
<td>D11</td>
<td>0942</td>
<td>1979-04-11</td>
</tr>
<tr>
<td>000220</td>
<td>JENNIFER</td>
<td>K</td>
<td>LUTZ</td>
<td>D11</td>
<td>0672</td>
<td>1968-08-29</td>
</tr>
<tr>
<td>000230</td>
<td>JAMES</td>
<td>J</td>
<td>JEFFERSON</td>
<td>D21</td>
<td>2094</td>
<td>1966-11-21</td>
</tr>
</tbody>
</table>
Table 163. Left half of DSN8C10.EMP: employee table. Note that a blank in the MIDINIT column is an actual value of " " rather than null. (continued)

<table>
<thead>
<tr>
<th>EMPNO</th>
<th>FIRSTNME</th>
<th>MIDINIT</th>
<th>LASTNAME</th>
<th>WORKDEPT</th>
<th>PHONENO</th>
<th>HIREDATE</th>
</tr>
</thead>
<tbody>
<tr>
<td>000240</td>
<td>SALVATORE</td>
<td>M</td>
<td>MARINO</td>
<td>D21</td>
<td>3780</td>
<td>1979-12-05</td>
</tr>
<tr>
<td>000250</td>
<td>DANIEL</td>
<td>S</td>
<td>SMITH</td>
<td>D21</td>
<td>0961</td>
<td>1969-10-30</td>
</tr>
<tr>
<td>000260</td>
<td>SYBIL</td>
<td>P</td>
<td>JOHNSON</td>
<td>D21</td>
<td>8953</td>
<td>1975-09-11</td>
</tr>
<tr>
<td>000270</td>
<td>MARIA</td>
<td>L</td>
<td>PEREZ</td>
<td>D21</td>
<td>9001</td>
<td>1980-09-30</td>
</tr>
<tr>
<td>000280</td>
<td>ETHEL</td>
<td>R</td>
<td>SCHNEIDER</td>
<td>E11</td>
<td>8997</td>
<td>1967-03-24</td>
</tr>
<tr>
<td>000290</td>
<td>JOHN</td>
<td>R</td>
<td>PARKER</td>
<td>E11</td>
<td>4502</td>
<td>1980-05-30</td>
</tr>
<tr>
<td>000300</td>
<td>PHILIP</td>
<td>X</td>
<td>SMITH</td>
<td>E11</td>
<td>2095</td>
<td>1972-06-19</td>
</tr>
<tr>
<td>000310</td>
<td>MAUDE</td>
<td>F</td>
<td>SETRIGHT</td>
<td>E11</td>
<td>3332</td>
<td>1964-09-12</td>
</tr>
<tr>
<td>000320</td>
<td>RAMLAL</td>
<td>V</td>
<td>MEHTA</td>
<td>E21</td>
<td>9990</td>
<td>1965-07-07</td>
</tr>
<tr>
<td>000330</td>
<td>WING</td>
<td></td>
<td>LEE</td>
<td>E21</td>
<td>2103</td>
<td>1976-02-23</td>
</tr>
<tr>
<td>000340</td>
<td>JASON</td>
<td>R</td>
<td>GOUNOT</td>
<td>E21</td>
<td>5698</td>
<td>1947-05-05</td>
</tr>
<tr>
<td>200110</td>
<td>DIAN</td>
<td>J</td>
<td>HEMMINGER</td>
<td>A00</td>
<td>3978</td>
<td>1965-01-01</td>
</tr>
<tr>
<td>200120</td>
<td>GREG</td>
<td></td>
<td>ORLANDO</td>
<td>A00</td>
<td>2167</td>
<td>1972-05-05</td>
</tr>
<tr>
<td>200140</td>
<td>KIM</td>
<td>N</td>
<td>NATZ</td>
<td>C01</td>
<td>1793</td>
<td>1976-12-15</td>
</tr>
<tr>
<td>200170</td>
<td>KIYOSHI</td>
<td></td>
<td>YAMAMOTO</td>
<td>D11</td>
<td>2890</td>
<td>1978-09-15</td>
</tr>
<tr>
<td>200220</td>
<td>REBA</td>
<td>K</td>
<td>JOHN</td>
<td>D11</td>
<td>0672</td>
<td>1968-08-29</td>
</tr>
<tr>
<td>200240</td>
<td>ROBERT</td>
<td>M</td>
<td>MONTEVERDE</td>
<td>D21</td>
<td>3780</td>
<td>1979-12-05</td>
</tr>
<tr>
<td>200280</td>
<td>EILEEN</td>
<td>R</td>
<td>SCHWARTZ</td>
<td>E11</td>
<td>8997</td>
<td>1967-03-24</td>
</tr>
<tr>
<td>200310</td>
<td>MICHELLE</td>
<td>F</td>
<td>SPRINGER</td>
<td>E11</td>
<td>3332</td>
<td>1964-09-12</td>
</tr>
<tr>
<td>200330</td>
<td>HELENA</td>
<td></td>
<td>WONG</td>
<td>E21</td>
<td>2103</td>
<td>1976-02-23</td>
</tr>
<tr>
<td>200340</td>
<td>ROY</td>
<td>R</td>
<td>ALONZO</td>
<td>E21</td>
<td>5698</td>
<td>1947-05-05</td>
</tr>
</tbody>
</table>

(Table 163 on page 989 shows the first half (right side) of the content of employee table.)

Table 164. Right half of DSN8C10.EMP: employee table

<table>
<thead>
<tr>
<th>(EMPNO)</th>
<th>JOB</th>
<th>EDLEVEL</th>
<th>SEX</th>
<th>BIRTHDATE</th>
<th>SALARY</th>
<th>BONUS</th>
<th>COMM</th>
</tr>
</thead>
<tbody>
<tr>
<td>(000010)</td>
<td>PRES</td>
<td>18</td>
<td>F</td>
<td>1933-08-14</td>
<td>52750.00</td>
<td>1000.00</td>
<td>4220.00</td>
</tr>
<tr>
<td>(000020)</td>
<td>MGR</td>
<td>18</td>
<td>M</td>
<td>1948-02-02</td>
<td>41250.00</td>
<td>800.00</td>
<td>3300.00</td>
</tr>
<tr>
<td>(000030)</td>
<td>MGR</td>
<td>20</td>
<td>F</td>
<td>1941-05-11</td>
<td>38250.00</td>
<td>800.00</td>
<td>3060.00</td>
</tr>
<tr>
<td>(000050)</td>
<td>MGR</td>
<td>16</td>
<td>M</td>
<td>1925-09-15</td>
<td>40175.00</td>
<td>800.00</td>
<td>3214.00</td>
</tr>
<tr>
<td>(000060)</td>
<td>MGR</td>
<td>16</td>
<td>M</td>
<td>1945-07-07</td>
<td>32250.00</td>
<td>600.00</td>
<td>2580.00</td>
</tr>
<tr>
<td>(000070)</td>
<td>MGR</td>
<td>16</td>
<td>F</td>
<td>1953-05-26</td>
<td>36170.00</td>
<td>700.00</td>
<td>2893.00</td>
</tr>
<tr>
<td>(000090)</td>
<td>MGR</td>
<td>16</td>
<td>F</td>
<td>1941-05-15</td>
<td>29750.00</td>
<td>600.00</td>
<td>2380.00</td>
</tr>
<tr>
<td>(000100)</td>
<td>MGR</td>
<td>14</td>
<td>M</td>
<td>1956-12-18</td>
<td>26150.00</td>
<td>500.00</td>
<td>2092.00</td>
</tr>
<tr>
<td>(000110)</td>
<td>SALES</td>
<td>19</td>
<td>M</td>
<td>1929-11-05</td>
<td>46500.00</td>
<td>900.00</td>
<td>3720.00</td>
</tr>
<tr>
<td>(000120)</td>
<td>CLERK</td>
<td>14</td>
<td>M</td>
<td>1942-10-18</td>
<td>29250.00</td>
<td>600.00</td>
<td>2340.00</td>
</tr>
<tr>
<td>(000130)</td>
<td>ANALYST</td>
<td>16</td>
<td>F</td>
<td>1925-09-15</td>
<td>23800.00</td>
<td>500.00</td>
<td>1904.00</td>
</tr>
</tbody>
</table>
### Table 164. Right half of DSN8C10.EMP: employee table (continued)

<table>
<thead>
<tr>
<th>(EMPNO)</th>
<th>JOB</th>
<th>EDLEVEL</th>
<th>SEX</th>
<th>BIRTHDATE</th>
<th>SALARY</th>
<th>BONUS</th>
<th>COMM</th>
</tr>
</thead>
<tbody>
<tr>
<td>(000140)</td>
<td>ANALYST</td>
<td>18</td>
<td>F</td>
<td>1946-01-19</td>
<td>28420.00</td>
<td>600.00</td>
<td>2274.00</td>
</tr>
<tr>
<td>(000150)</td>
<td>DESIGNER</td>
<td>16</td>
<td>M</td>
<td>1947-05-17</td>
<td>25280.00</td>
<td>500.00</td>
<td>2022.00</td>
</tr>
<tr>
<td>(000160)</td>
<td>DESIGNER</td>
<td>17</td>
<td>F</td>
<td>1955-04-12</td>
<td>22250.00</td>
<td>400.00</td>
<td>1780.00</td>
</tr>
<tr>
<td>(000170)</td>
<td>DESIGNER</td>
<td>16</td>
<td>M</td>
<td>1951-01-05</td>
<td>24680.00</td>
<td>500.00</td>
<td>1974.00</td>
</tr>
<tr>
<td>(000180)</td>
<td>DESIGNER</td>
<td>17</td>
<td>F</td>
<td>1949-02-21</td>
<td>21340.00</td>
<td>500.00</td>
<td>1707.00</td>
</tr>
<tr>
<td>(000190)</td>
<td>DESIGNER</td>
<td>16</td>
<td>M</td>
<td>1952-06-25</td>
<td>20450.00</td>
<td>500.00</td>
<td>1636.00</td>
</tr>
<tr>
<td>(000200)</td>
<td>DESIGNER</td>
<td>17</td>
<td>F</td>
<td>1955-04-12</td>
<td>22250.00</td>
<td>400.00</td>
<td>1780.00</td>
</tr>
<tr>
<td>(000210)</td>
<td>DESIGNER</td>
<td>16</td>
<td>M</td>
<td>1951-01-05</td>
<td>24680.00</td>
<td>500.00</td>
<td>1974.00</td>
</tr>
<tr>
<td>(000220)</td>
<td>DESIGNER</td>
<td>17</td>
<td>F</td>
<td>1949-02-21</td>
<td>21340.00</td>
<td>500.00</td>
<td>1707.00</td>
</tr>
<tr>
<td>(000230)</td>
<td>CLERK</td>
<td>14</td>
<td>M</td>
<td>1935-05-30</td>
<td>22180.00</td>
<td>400.00</td>
<td>1774.00</td>
</tr>
<tr>
<td>(000240)</td>
<td>CLERK</td>
<td>17</td>
<td>M</td>
<td>1954-03-31</td>
<td>28760.00</td>
<td>600.00</td>
<td>2301.00</td>
</tr>
<tr>
<td>(000250)</td>
<td>CLERK</td>
<td>15</td>
<td>M</td>
<td>1939-11-12</td>
<td>19180.00</td>
<td>400.00</td>
<td>1534.00</td>
</tr>
<tr>
<td>(000260)</td>
<td>CLERK</td>
<td>16</td>
<td>F</td>
<td>1936-10-05</td>
<td>17250.00</td>
<td>300.00</td>
<td>1380.00</td>
</tr>
<tr>
<td>(000270)</td>
<td>CLERK</td>
<td>15</td>
<td>F</td>
<td>1953-05-26</td>
<td>27380.00</td>
<td>500.00</td>
<td>2190.00</td>
</tr>
<tr>
<td>(000280)</td>
<td>OPERATOR</td>
<td>17</td>
<td>F</td>
<td>1936-03-28</td>
<td>26250.00</td>
<td>500.00</td>
<td>2100.00</td>
</tr>
<tr>
<td>(000290)</td>
<td>OPERATOR</td>
<td>12</td>
<td>M</td>
<td>1946-07-09</td>
<td>15340.00</td>
<td>300.00</td>
<td>1227.00</td>
</tr>
<tr>
<td>(000300)</td>
<td>OPERATOR</td>
<td>14</td>
<td>M</td>
<td>1936-10-27</td>
<td>17750.00</td>
<td>400.00</td>
<td>1420.00</td>
</tr>
<tr>
<td>(000310)</td>
<td>OPERATOR</td>
<td>12</td>
<td>F</td>
<td>1931-04-21</td>
<td>15900.00</td>
<td>300.00</td>
<td>1272.00</td>
</tr>
<tr>
<td>(000320)</td>
<td>FIELDREP</td>
<td>16</td>
<td>M</td>
<td>1932-08-11</td>
<td>19950.00</td>
<td>400.00</td>
<td>1596.00</td>
</tr>
<tr>
<td>(000330)</td>
<td>FIELDREP</td>
<td>14</td>
<td>M</td>
<td>1941-07-18</td>
<td>25370.00</td>
<td>500.00</td>
<td>2030.00</td>
</tr>
<tr>
<td>(000340)</td>
<td>FIELDREP</td>
<td>16</td>
<td>M</td>
<td>1926-05-17</td>
<td>23840.00</td>
<td>500.00</td>
<td>1907.00</td>
</tr>
<tr>
<td>(200010)</td>
<td>SALESREP</td>
<td>18</td>
<td>F</td>
<td>1933-08-14</td>
<td>46500.00</td>
<td>1000.00</td>
<td>4220.00</td>
</tr>
<tr>
<td>(200120)</td>
<td>CLERK</td>
<td>14</td>
<td>M</td>
<td>1942-10-18</td>
<td>29250.00</td>
<td>600.00</td>
<td>2340.00</td>
</tr>
<tr>
<td>(200140)</td>
<td>ANALYST</td>
<td>18</td>
<td>F</td>
<td>1946-01-19</td>
<td>28420.00</td>
<td>600.00</td>
<td>2274.00</td>
</tr>
<tr>
<td>(200170)</td>
<td>DESIGNER</td>
<td>16</td>
<td>M</td>
<td>1951-01-05</td>
<td>24680.00</td>
<td>500.00</td>
<td>1974.00</td>
</tr>
<tr>
<td>(200220)</td>
<td>DESIGNER</td>
<td>18</td>
<td>F</td>
<td>1948-03-19</td>
<td>29840.00</td>
<td>600.00</td>
<td>2387.00</td>
</tr>
<tr>
<td>(200240)</td>
<td>CLERK</td>
<td>17</td>
<td>M</td>
<td>1954-03-31</td>
<td>28760.00</td>
<td>600.00</td>
<td>2301.00</td>
</tr>
<tr>
<td>(200280)</td>
<td>OPERATOR</td>
<td>17</td>
<td>F</td>
<td>1936-03-28</td>
<td>26250.00</td>
<td>500.00</td>
<td>2100.00</td>
</tr>
<tr>
<td>(200310)</td>
<td>OPERATOR</td>
<td>12</td>
<td>F</td>
<td>1931-04-21</td>
<td>15900.00</td>
<td>300.00</td>
<td>1272.00</td>
</tr>
<tr>
<td>(200330)</td>
<td>FIELDREP</td>
<td>14</td>
<td>F</td>
<td>1941-07-18</td>
<td>25370.00</td>
<td>500.00</td>
<td>2030.00</td>
</tr>
<tr>
<td>(200340)</td>
<td>FIELDREP</td>
<td>16</td>
<td>M</td>
<td>1926-05-17</td>
<td>23840.00</td>
<td>500.00</td>
<td>1907.00</td>
</tr>
</tbody>
</table>

### Relationship to other tables

The employee table is a parent table of:
- The department table, through a foreign key on column MGRNO
- The project table, through a foreign key on column RESPEMP

Chapter 12. Sample data and applications supplied with Db2 for z/OS 991
The employee table is a dependent of the department table, through its foreign key on column WORKDEPT.

**Employee photo and resume table (DSN8C10.EMP_PHOTO_RESUME)**

The sample employee photo and resume table complements the employee table.

Each row of the photo and resume table contains a photo of the employee, in two formats, and the employee's resume. The photo and resume table resides in table space DSN8S12B. The following statement creates the table:

```sql
CREATE TABLE DSN8C10.EMP_PHOTO_RESUME
(EMPNO CHAR(06) NOT NULL,
EMP_ROWID ROWID NOT NULL GENERATED ALWAYS,
PSEG_PHOTO BLOB(500K),
BMP_PHOTO BLOB(100K),
RESUME CLOB(5K))
PRIMARY KEY (EMPNO)
IN DSN8D12L.DSN8S12B
CCSID EBCDIC;
```

Db2 requires an auxiliary table for each LOB column in a table. The following statements define the auxiliary tables for the three LOB columns in DSN8C10.EMP_PHOTO_RESUME:

```sql
CREATE AUX TABLE DSN8C10.AUX_BMP_PHOTO
IN DSN8D12L.DSN8S12M
STORES DSN8C10.EMP_PHOTO_RESUME
COLUMN BMP_PHOTO;

CREATE AUX TABLE DSN8C10.AUX_PSEG_PHOTO
IN DSN8D12L.DSN8S12L
STORES DSN8C10.EMP_PHOTO_RESUME
COLUMN PSEG_PHOTO;

CREATE AUX TABLE DSN8C10.AUX_EMP_RESUME
IN DSN8D12L.DSN8S12N
STORES DSN8C10.EMP_PHOTO_RESUME
COLUMN RESUME;
```

**Content of the employee photo and resume table**

The following table shows the content of the columns in the employee photo and resume table.

<table>
<thead>
<tr>
<th>Table 165. Columns of the employee photo and resume table</th>
</tr>
</thead>
<tbody>
<tr>
<td>Column</td>
</tr>
<tr>
<td>--------</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>5</td>
</tr>
</tbody>
</table>

The following table shows the indexes for the employee photo and resume table.

<table>
<thead>
<tr>
<th>Table 166. Indexes of the employee photo and resume table</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
</tr>
<tr>
<td>------</td>
</tr>
<tr>
<td>DSN8C10.XEMP_PHOTO_RESUME</td>
</tr>
</tbody>
</table>
The following table shows the indexes for the auxiliary tables that support the employee photo and resume table.

### Table 167. Indexes of the auxiliary tables for the employee photo and resume table

<table>
<thead>
<tr>
<th>Name</th>
<th>On table</th>
<th>Type of index</th>
</tr>
</thead>
<tbody>
<tr>
<td>DSN8C10.XAUX_BMP_PHOTO</td>
<td>DSN8C10.AUX_BMP_PHOTO</td>
<td>Unique</td>
</tr>
<tr>
<td>DSN8C10.XAUX_PSEG_PHOTO</td>
<td>DSN8C10.AUX_PSEG_PHOTO</td>
<td>Unique</td>
</tr>
<tr>
<td>DSN8C10.XAUX_EMP_RESUME</td>
<td>DSN8C10.AUX_EMP_RESUME</td>
<td>Unique</td>
</tr>
</tbody>
</table>

### Relationship to other tables

The employee photo and resume table is a parent table of the project table, through a foreign key on column RESPEMP.

### Project table (DSN8C10.PROJ)

The sample project table describes each project that the business is currently undertaking. Data that is contained in each row of the table includes the project number, name, person responsible, and schedule dates.

The project table resides in database DSN8D12A. Because this table has foreign keys that reference DEPT and EMP, those tables and the indexes on their primary keys must be created first. Then PROJ is created with the following statement:

```sql
CREATE TABLE DSN8C10.PROJ
(PROJNO CHAR(6) PRIMARY KEY NOT NULL,
PROJNAME VARCHAR(24) NOT NULL WITH DEFAULT
'PROJECT NAME UNDEFINED',
DEPTNO CHAR(3) NOT NULL REFERENCES DSN8C10.DEPT ON DELETE RESTRICT,
RESPEMP CHAR(6) NOT NULL REFERENCES DSN8C10.EMP ON DELETE RESTRICT,
PRSTAFF DECIMAL(5, 2),
PRSTDATE DATE,
PRENDATE DATE,
MAJPROJ CHAR(6))
IN DSN8D12A.DSN8S12P
CCSID EBCDIC;
```

Because the project table is self-referencing, the foreign key for that constraint must be added later with the following statement:

```sql
ALTER TABLE DSN8C10.PROJ
FOREIGN KEY RPP (MAJPROJ) REFERENCES DSN8C10.PROJ
ON DELETE CASCADE;
```

### Content of the project table

The following table shows the content of the columns of the project table.

### Table 168. Columns of the project table

<table>
<thead>
<tr>
<th>Column</th>
<th>Column name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>PROJNO</td>
<td>Project ID (the primary key)</td>
</tr>
<tr>
<td>2</td>
<td>PROJNAME</td>
<td>Project name</td>
</tr>
<tr>
<td>3</td>
<td>DEPTNO</td>
<td>ID of department responsible for the project</td>
</tr>
</tbody>
</table>
Table 168. Columns of the project table (continued)

<table>
<thead>
<tr>
<th>Column</th>
<th>Column name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>RESPEMP</td>
<td>ID of employee responsible for the project</td>
</tr>
<tr>
<td>5</td>
<td>PRSTAFF</td>
<td>Estimated mean number of persons that are needed between PRSTDATE and PRENDATE to complete the whole project, including any subprojects</td>
</tr>
<tr>
<td>6</td>
<td>PRSTDATE</td>
<td>Estimated project start date</td>
</tr>
<tr>
<td>7</td>
<td>PRENDATE</td>
<td>Estimated project end date</td>
</tr>
<tr>
<td>8</td>
<td>MAJPROJ</td>
<td>ID of any project of which this project is a part</td>
</tr>
</tbody>
</table>

The following table shows the indexes for the project table:

Table 169. Indexes of the project table

<table>
<thead>
<tr>
<th>Name</th>
<th>On column</th>
<th>Type of index</th>
</tr>
</thead>
<tbody>
<tr>
<td>DSN8C10.XPROJ1</td>
<td>PROJNO</td>
<td>Primary, ascending</td>
</tr>
<tr>
<td>DSN8C10.XPROJ2</td>
<td>RESPEMP</td>
<td>Ascending</td>
</tr>
</tbody>
</table>

Relationship to other tables

The table is self-referencing: a non-null value of MAJPROJ must be a valid project number. The table is a parent table of the project activity table, through a foreign key on column PROJNO. It is a dependent of the following tables:

- The department table, through its foreign key on DEPTNO
- The employee table, through its foreign key on RESPEMP

Project activity table (DSN8C10.PROJACT)

The sample project activity table lists the activities that are performed for each project.

The project activity table resides in database DSN8D12A. Because this table has foreign keys that reference PROJ and ACT, those tables and the indexes on their primary keys must be created first. Then PROJACT is created with the following statement:

```
CREATE TABLE DSN8C10.PROJACT
  (PROJNO  CHAR(6)     NOT NULL,
   ACTNO   SMALLINT    NOT NULL,
   ACSTAFF DECIMAL(5,2) NOT NULL,
   ACSTDATE DATE       NOT NULL,
   ACENDATE DATE       NOT NULL,
   PRIMARY KEY (PROJNO, ACTNO, ACSTDATE),
   FOREIGN KEY RPAP (PROJNO) REFERENCES DSN8C10.PROJ
     ON DELETE RESTRICT,
   FOREIGN KEY RPAA (ACTNO) REFERENCES DSN8C10.ACT
     ON DELETE RESTRICT)
IN DSN8D12A.DSN8S12P
CCSID EBCDIC;
```

Content of the project activity table

The following table shows the content of the columns of the project activity table.
Table 170. Columns of the project activity table

<table>
<thead>
<tr>
<th>Column</th>
<th>Column name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>PROJNO</td>
<td>Project ID</td>
</tr>
<tr>
<td>2</td>
<td>ACTNO</td>
<td>Activity ID</td>
</tr>
<tr>
<td>3</td>
<td>ACSTAFF</td>
<td>Estimated mean number of employees that are needed to staff the activity</td>
</tr>
<tr>
<td>4</td>
<td>ACSTDATE</td>
<td>Estimated activity start date</td>
</tr>
<tr>
<td>5</td>
<td>ACENDATE</td>
<td>Estimated activity completion date</td>
</tr>
</tbody>
</table>

The following table shows the index of the project activity table:

Table 171. Index of the project activity table

<table>
<thead>
<tr>
<th>Name</th>
<th>On columns</th>
<th>Type of index</th>
</tr>
</thead>
<tbody>
<tr>
<td>DSN8C10.XPROJAC1</td>
<td>PROJNO, ACTNO, ACSTDATE</td>
<td>primary, ascending</td>
</tr>
</tbody>
</table>

Relationship to other tables

The project activity table is a parent table of the employee to project activity table, through a foreign key on columns PROJNO, ACTNO, and EMSTDATE. It is a dependent of the following tables:

- The activity table, through its foreign key on column ACTNO
- The project table, through its foreign key on column PROJNO

Related reference

Activity table (DSN8C10.ACT) (Introduction to Db2 for z/OS)
Project table (DSN8C10.PROJ) (Introduction to Db2 for z/OS)

Employee-to-project activity table (DSN8C10.EMPPROJACT)

The sample employee-to-project activity table identifies the employee who performs an activity for a project, tells the proportion of the employee's time that is required, and gives a schedule for the activity.

The employee-to-project activity table resides in database DSN8D12A. Because this table has foreign keys that reference EMP and PROJACT, those tables and the indexes on their primary keys must be created first. Then EMPPROJACT is created with the following statement:

```
CREATE TABLE DSN8C10.EMPPROJACT
(EMPNO CHAR(6) NOT NULL,
PROJNO CHAR(6) NOT NULL,
ACTNO SMALLINT NOT NULL,
EMPTIME DECIMAL(5,2) ,
EMSTDATE DATE ,
EMENDATE DATE ,
FOREIGN KEY REPAPA (PROJNO, ACTNO, EMSTDATE)
REFERENCES DSN8C10.PROJACT
ON DELETE RESTRICT,
FOREIGN KEY REPAE (EMPNO) REFERENCES DSN8C10.EMP
ON DELETE RESTRICT)
IN DSN8D12A.DSN8S12P
CCSID EBCDIC;
```

Content of the employee-to-project activity table

The following table shows the content of the columns in the employee-to-project activity table.
Table 172. Columns of the employee-to-project activity table

<table>
<thead>
<tr>
<th>Column</th>
<th>Column name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>EMPNO</td>
<td>Employee ID number</td>
</tr>
<tr>
<td>2</td>
<td>PROJNO</td>
<td>Project ID of the project</td>
</tr>
<tr>
<td>3</td>
<td>ACTNO</td>
<td>ID of the activity within the project</td>
</tr>
<tr>
<td>4</td>
<td>EMPTIME</td>
<td>A proportion of the employee’s full time (between 0.00 and 1.00) that is to be spent on the activity</td>
</tr>
<tr>
<td>5</td>
<td>EMSTDATE</td>
<td>Date the activity starts</td>
</tr>
<tr>
<td>6</td>
<td>EMENDATE</td>
<td>Date the activity ends</td>
</tr>
</tbody>
</table>

The following table shows the indexes for the employee-to-project activity table:

Table 173. Indexes of the employee-to-project activity table

<table>
<thead>
<tr>
<th>Name</th>
<th>On columns</th>
<th>Type of index</th>
</tr>
</thead>
<tbody>
<tr>
<td>DSN8C10.XEMPPROJACT1</td>
<td>PROJNO, ACTNO, EMSTDATE, EMPNO</td>
<td>Unique, ascending</td>
</tr>
<tr>
<td>DSN8C10.XEMPPROJACT2</td>
<td>EMPNO</td>
<td>Ascending</td>
</tr>
</tbody>
</table>

Relationship to other tables

The employee-to-project activity table is a dependent of the following tables:

- The employee table, through its foreign key on column EMPNO
- The project activity table, through its foreign key on columns PROJNO, ACTNO, and EMSTDATE.

Related reference

- Employee table (DSN8C10.EMP) (Introduction to Db2 for z/OS)
- Project activity table (DSN8C10.PROJACT) (Introduction to Db2 for z/OS)

Unicode sample table (DSN8C10.DEMO_UNICODE)

The Unicode sample table is used to verify that data conversions to and from EBCDIC and Unicode are working as expected.

The table resides in database DSN8D12A, and is defined with the following statement:

```sql
CREATE TABLE DSN8C10.DEMO_UNICODE
  (LOWER_A_TO_Z      CHAR(26)                     ,
   UPPER_A_TO_Z      CHAR(26)                     ,
   ZERO_TO_NINE      CHAR(10)                     ,
   X00_TO_XFF        VARCHAR(256) FOR BIT DATA) IN DSN8D81E.DSN8S81U
  CCSID UNICODE;
```

Content of the Unicode sample table

The following table shows the content of the columns in the Unicode sample table:
### Table 174. Columns of the Unicode sample table

<table>
<thead>
<tr>
<th>Column</th>
<th>Column Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>LOWER_A_TO_Z</td>
<td>Array of characters, 'a' to 'z'</td>
</tr>
<tr>
<td>2</td>
<td>UPPER_A_TO_Z</td>
<td>Array of characters, 'A' to 'Z'</td>
</tr>
<tr>
<td>3</td>
<td>ZERO_TO_NINE</td>
<td>Array of characters, '0' to '9'</td>
</tr>
<tr>
<td>4</td>
<td>X00_TO_XFF</td>
<td>Array of characters, x'00' to x'FF'</td>
</tr>
</tbody>
</table>

This table has no indexes.

**Relationship to other tables**

This table has no relationship to other tables.

**Relationships among the sample tables**

Relationships among the sample tables are established by foreign keys in dependent tables that reference primary keys in parent tables.

The following figure shows relationships among the sample tables. You can find descriptions of the columns with the descriptions of the tables.
Figure 77. Relationships among tables in the sample application
Views on the sample tables

Db2 creates a number of views on the sample tables for use in the sample applications.

The following table indicates the tables on which each view is defined and the sample applications that use the view. All view names have the qualifier DSN8C10.

<table>
<thead>
<tr>
<th>View name</th>
<th>On tables or views</th>
<th>Used in application</th>
</tr>
</thead>
<tbody>
<tr>
<td>VDEPT</td>
<td>DEPT</td>
<td>Organization</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Project</td>
</tr>
<tr>
<td>VHDEPT</td>
<td>DEPT</td>
<td>Distributed organization</td>
</tr>
<tr>
<td>VEMP</td>
<td>EMP</td>
<td>Distributed organization</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Organization</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Project</td>
</tr>
<tr>
<td>VPROJ</td>
<td>PROJ</td>
<td>Project</td>
</tr>
<tr>
<td>VACT</td>
<td>ACT</td>
<td>Project</td>
</tr>
<tr>
<td>VPROJACT</td>
<td>PROJACT</td>
<td>Project</td>
</tr>
<tr>
<td>VEMPPROJACT</td>
<td>EMPPROJACT</td>
<td>Project</td>
</tr>
<tr>
<td>VDEPMG1</td>
<td>DEPT</td>
<td>Organization</td>
</tr>
<tr>
<td></td>
<td>EMP</td>
<td></td>
</tr>
<tr>
<td>VEMPDPT1</td>
<td>DEPT</td>
<td>Organization</td>
</tr>
<tr>
<td></td>
<td>EMP</td>
<td></td>
</tr>
<tr>
<td>VASTRDE1</td>
<td>DEPT</td>
<td></td>
</tr>
<tr>
<td>VASTRDE2</td>
<td>VDEPMG1</td>
<td>Organization</td>
</tr>
<tr>
<td></td>
<td>EMP</td>
<td></td>
</tr>
<tr>
<td>VPROJRE1</td>
<td>PROJ</td>
<td>Project</td>
</tr>
<tr>
<td></td>
<td>EMP</td>
<td></td>
</tr>
<tr>
<td>VPSTRDE1</td>
<td>VPROJRE1</td>
<td>Project</td>
</tr>
<tr>
<td></td>
<td>VPROJRE2</td>
<td></td>
</tr>
<tr>
<td>VPSTRDE2</td>
<td>VPROJRE1</td>
<td>Project</td>
</tr>
</tbody>
</table>
Table 175. Views on sample tables (continued)

<table>
<thead>
<tr>
<th>View name</th>
<th>On tables or views</th>
<th>Used in application</th>
</tr>
</thead>
<tbody>
<tr>
<td>VFORPLA</td>
<td>VPROJRE1, EMPPROJACT</td>
<td>Project</td>
</tr>
<tr>
<td>VSTAFAC1</td>
<td>PROJACT, ACT</td>
<td>Project</td>
</tr>
<tr>
<td>VSTAFAC2</td>
<td>EMPPROJACT, ACT, EMP</td>
<td>Project</td>
</tr>
<tr>
<td>VPHONE</td>
<td>EMP, DEPT</td>
<td>Phone</td>
</tr>
<tr>
<td>VEMPLP</td>
<td>EMP</td>
<td>Phone</td>
</tr>
</tbody>
</table>

The following SQL statement creates the view named VDEPT.

```
CREATE VIEW DSN8C10.VDEPT
  AS SELECT ALL
      DEPTNO  ,
      DEPTNAME,
      MGRNO   ,
      ADMRDEPT
  FROM DSN8C10.DEPT;
```

The following SQL statement creates the view named VHDEPT.

```
CREATE VIEW DSN8C10.VHDEPT
  AS SELECT ALL
      DEPTNO  ,
      DEPTNAME,
      MGRNO   ,
      ADMRDEPT,
      LOCATION
  FROM DSN8C10.DEPT;
```

The following SQL statement creates the view named VEMP.

```
CREATE VIEW DSN8C10.VEMP
  AS SELECT ALL
      EMPNO   ,
      FIRSTNAME,
      MIDINIT ,
      LASTNAME ,
      WORKDEPT
  FROM DSN8C10.EMP;
```

The following SQL statement creates the view named VPROJ.

```
CREATE VIEW DSN8C10.VPROJ
  AS SELECT ALL
      PROJNO, PROJNAME, DEPTNO, RESPEMP, PRSTAFF,
      PRSTDATE, PRENDATE, MAJPROJ
  FROM DSN8C10.PROJ;
```

The following SQL statement creates the view named VACT.

```
CREATE VIEW DSN8C10.VACT
  AS SELECT ALL
      ACTNO   ,
      ACTKWD  ,
      ACTDESC
  FROM DSN8C10.ACT;
```
The following SQL statement creates the view named VPROJACT.

```sql
CREATE VIEW DSN8C10.VPROJACT
AS SELECT ALL
    PROJNO, ACTNO, ACSTAFF, ACSTDATE, ACENDATE
FROM DSN8C10.PROJACT ;
```

The following SQL statement creates the view named VEMPPROJACT.

```sql
CREATE VIEW DSN8C10.VEMPPROJACT
AS SELECT ALL
    EMPNO, PROJNO, ACTNO, EMPTIME, EMSTDATE, EMENDATE
FROM DSN8C10.EMPPROJACT ;
```

The following SQL statement creates the view named VDEPMG1.

```sql
CREATE VIEW DSN8C10.VDEPMG1
    (DEPTNO, DEPTNAME, MGRNO, FIRSTNME, MIDINIT,
     LASTNAME, ADMRDEPT)
AS SELECT ALL
    DEPTNO, DEPTNAME, MGRNO, FIRSTNME, MIDINIT,
    LASTNAME, ADMRDEPT
FROM DSN8C10.DEPT LEFT OUTER JOIN DSN8C10.EMP
    ON MGRNO = EMPNO ;
```

The following SQL statement creates the view named VEMPDPT1.

```sql
CREATE VIEW DSN8C10.VEMPDPT1
    (DEPTNO, DEPTNAME, EMPNO, FRSTINIT, MIDINIT,
     LASTNAME, WORKDEPT)
AS SELECT ALL
    DEPTNO, DEPTNAME, EMPNO, SUBSTR(FIRSTNME, 1, 1), MIDINIT,
    LASTNAME, WORKDEPT
FROM DSN8C10.DEPT RIGHT OUTER JOIN DSN8C10.EMP
    ON WORKDEPT = DEPTNO ;
```

The following SQL statement creates the view named VASTRDE1.

```sql
CREATE VIEW DSN8C10.VASTRDE1
    (DEPT1NO, DEPT1NAM, EMP1NO, EMP1FN, EMP1MI, EMP1LN, TYPE2,
     DEPT2NO, DEPT2NAM, EMP2NO, EMP2FN, EMP2MI, EMP2LN)
AS SELECT ALL
    D1.DEPTNO, D1.DEPTNAME, D1.MGRNO, D1.FIRSTNME, D1.MIDINIT,
    D1.LASTNAME, '1',
    D2.DEPTNO, D2.DEPTNAME, D2.MGRNO, D2.FIRSTNME, D2.MIDINIT,
    D2.LASTNAME
FROM DSN8C10.VDEPMG1 D1, DSN8C10.VDEPMG1 D2
    WHERE D1.DEPTNO = D2.ADMRDEPT ;
```

The following SQL statement creates the view named VASTRDE2.

```sql
CREATE VIEW DSN8C10.VASTRDE2
    (DEPT1NO, DEPT1NAM, EMP1NO, EMP1FN, EMP1MI, EMP1LN, TYPE2,
     DEPT2NO, DEPT2NAM, EMP2NO, EMP2FN, EMP2MI, EMP2LN)
AS SELECT ALL
    D1.DEPTNO, D1.DEPTNAME, D1.MGRNO, D1.FIRSTNME, D1.MIDINIT,
    D1.LASTNAME, '2',
    D1.DEPTNO, D1.DEPTNAME, E2.EMPNO, E2.FIRSTNME, E2.MIDINIT,
    E2.LASTNAME
FROM DSN8C10.VDEPMG1 D1, DSN8C10.EMP E2
    WHERE D1.DEPTNO = E2.WORKDEPT ;
```

The following figure shows the SQL statement that creates the view named VPROJRE1.
CREATE VIEW DSN8C10.VPROJRE1
(PROJNO, PROJNAME, PROJDEP, RESPEMP, FIRSTNME, MIDINIT,
LASTNAME, MAJPROJ)
AS SELECT ALL
  PROJNO, PROJNAME, DEPTNO, EMPNO, FIRSTNME, MIDINIT,
  LASTNAME, MAJPROJ
FROM DSN8C10.PROJ, DSN8C10.EMP
WHERE RESPEMP = EMPNO;

Figure 78. VPROJRE1

The following SQL statement creates the view named VPSTRDE1.

CREATE VIEW DSN8C10.VPSTRDE1
(PROJ1NO, PROJ1NAME, RESP1NO, RESP1FN, RESP1MI, RESP1LN,
PROJ2NO, PROJ2NAME, RESP2NO, RESP2FN, RESP2MI, RESP2LN)
AS SELECT ALL
  P1.PROJNO, P1.PROJNAME, P1.RESPEMP, P1.FIRSTNME, P1.MIDINIT,
  P1.LASTNAME,
  P2.PROJNO, P2.PROJNAME, P2.RESPEMP, P2.FIRSTNME, P2.MIDINIT,
  P2.LASTNAME
FROM DSN8C10.VPROJRE1 P1,
DSN8C10.VPROJRE1 P2
WHERE P1.PROJNO = P2.MAJPROJ;

The following SQL statement creates the view named VPSTRDE2.

CREATE VIEW DSN8C10.VPSTRDE2
(PROJ1NO, PROJ1NAME, RESP1NO, RESP1FN, RESP1MI, RESP1LN,
PROJ2NO, PROJ2NAME, RESP2NO, RESP2FN, RESP2MI, RESP2LN)
AS SELECT ALL
  P1.PROJNO, P1.PROJNAME, P1.RESPEMP, P1.FIRSTNME, P1.MIDINIT,
  P1.LASTNAME,
  P1.PROJNO, P1.PROJNAME, P1.RESPEMP, P1.FIRSTNME, P1.MIDINIT,
  P1.LASTNAME
FROM DSN8C10.VPROJRE1 P1
WHERE NOT EXISTS
  (SELECT * FROM DSN8C10.VPROJRE1 P2
   WHERE P1.PROJNO = P2.MAJPROJ);

The following SQL statement creates the view named VFORPLA.

CREATE VIEW DSN8C10.VFORPLA
(PROJNO, PROJNAME, RESPEMP, PROJDEP, FRSTINIT, MIDINIT, LASTNAME)
AS SELECT ALL
  F1.PROJNO, F1.PROJNAME, RESPEMP, PROJDEP, SUBSTR(FIRSTNME, 1, 1),
  MIDINIT, LASTNAME
FROM DSN8C10.VPROJRE1 F1 LEFT OUTER JOIN DSN8C10.EMPPROJACT F2
  ON F1.PROJNO = F2.PROJNO;

The following SQL statement creates the view named VSTAFAC1.

CREATE VIEW DSN8C10.VSTAFAC1
(PROJNO, ACTNO, ACTDESC, EMPNO, FIRSTNME, MIDINIT, LASTNAME,
EMPTIME, STDATE, ENDATE, TYPE)
AS SELECT ALL
  PA.PROJNO, PA.ACTNO, AC.ACTDESC, ' ', ' ', ' ', ' ',
  PA.ACSTAFF, PA.ACSTDATE,
  PA.ACENDATE,'1'
FROM DSN8C10.PROJACT PA, DSN8C10.ACT AC
WHERE PA.ACTNO = AC.ACTNO;

The following SQL statement creates the view named VSTAFAC2.

CREATE VIEW DSN8C10.VSTAFAC2
(PROJNO, ACTNO, ACTDESC, EMPNO, FIRSTNME, MIDINIT, LASTNAME,
EMPTIME, STDATE, ENDATE, TYPE)
AS SELECT ALL
  EP.PROJNO, EP.ACTNO, AC.ACTDESC, EP.EMPNO, EM.FIRSTNME,
  EM.MIDINIT, EM.LASTNAME, EP.EMPTIME, EP.EMSTDATE,
  EP.EMENDATE,'2'
FROM DSN8C10.EMPPROJACT EP, DSN8C10.ACT AC, DSN8C10.EMP EM
WHERE EP.ACTNO = AC.ACTNO AND EP.EMPNO = EM.EMPNO;
The following SQL statement creates the view named VPHONE.

```sql
CREATE VIEW DSN8C10.VPHONE
(LASTNAME,
FIRSTNAME,
MIDDLEINITIAL,
PHONENUMBER,
EMPLOYEEENUMBER,
DEPTNUMBER,
DEPTNAME)
AS SELECT ALL      LASTNAME,
FIRSTNAME,
MIDDLEINIT,
VALUE(PHONENO,'    '),
EMPNO,
DEPTNO,
DEPTNAME
FROM DSN8C10.EMP, DSN8C10.DEPT
WHERE WORKDEPT = DEPTNO;
```

The following SQL statement creates the view named VEMPLP.

```sql
CREATE VIEW DSN8C10.VEMPLP
(EMPLOYEEENUMBER,
PHONENUMBER)
AS SELECT ALL      EMPNO   ,
PHONENO
FROM DSN8C10.EMP;
```

Storage of sample application tables

Normally, related data is stored in the same database.

The following figure shows how the sample tables are related to databases and storage groups. Two databases are used to illustrate the possibility.
Figure 79. Relationship among sample databases and table spaces

Storage group:

DSNBG<vr>0

Databases:

DSN8D<vr>A application data

DSN8D<vr>L LOB application data

DSN8D<vr>P common for programming tables

Table spaces:

DSN8S<vr>D department table

DSN8S<vr>E employee table

Separate spaces for other application tables

LOB spaces for employee photo and resume table

DSN8S<vr>P common for programming table

<vr> is a 2-digit version identifier.
In addition to the storage group and databases that are shown in the preceding figure, the storage group DSN8G12U and database DSN8D12U are created when you run DSNTEJ2A.

**Storage group for sample application data**

Sample application data is stored in storage group DSN8G120. The default storage group, SYSDEFLT, which is created when Db2 is installed, is not used to store sample application data.

The storage group that is used to store sample application data is defined by the following statement:

```
CREATE STOGROUP DSN8G120
  VOLUMES (DSNV01)
  VCAT  DSNC111;
```

**Databases for sample application data**

Sample application data is stored in several different databases. The default database that is created when Db2 is installed is not used to store the sample application data.

DSN8D12P is the database that is used for tables that are related to programs. The other databases are used for tables that are related to applications. The databases are defined by the following statements:

```
CREATE DATABASE DSN8D12A
  STOGROUP DSN8G120
  BUFFERPOOL BP0
  CCSID EBCDIC;

CREATE DATABASE DSN8D12P
  STOGROUP DSN8G120
  BUFFERPOOL BP0
  CCSID EBCDIC;

CREATE DATABASE DSN8D12L
  STOGROUP DSN8G120
  BUFFERPOOL BP0
  CCSID EBCDIC;

CREATE DATABASE DSN8D12E
  STOGROUP DSN8G120
  BUFFERPOOL BP0
  CCSID UNICODE;

CREATE DATABASE DSN8D12U
  STOGROUP DSN8G12U
  CCSID EBCDIC;
```

**Table spaces for sample application data**

The table spaces that are not explicitly defined are created implicitly in the DSN8D12A database, using the default space attributes.

The following SQL statements explicitly define a series of table spaces.

```
CREATE TABLESPACE DSN8S12D
  IN DSN8D12A
  USING STOGROUP DSN8G120
  PRIQTY 20
  SECQTY 20
  ERASE NO
  LOCKSIZE PAGE LOCKMAX SYSTEM
  BUFFERPOOL BP0
```
CLOSE NO
CCSID EBCDIC;

CREATE TABLESPACE DSN8S12E
IN DSN8D12A
USING STOGROUP DSN8G120
PRIQTY 20
SECQTY 20
ERASE NO
NUMPARTS 4
(PART 1 USING STOGROUP DSN8G120
PRIQTY 12
SECQTY 12,
PART 3 USING STOGROUP DSN8G120
PRIQTY 12
SECQTY 12)
LOCKSIZE PAGE LOCKMAX SYSTEM
BUFFERPOOL BP0
CLOSE NO
COMPRESS YES
CCSID EBCDIC;

CREATE TABLESPACE DSN8S12B
IN DSN8D12L
USING STOGROUP DSN8G120
PRIQTY 20
SECQTY 20
ERASE NO
LOCKSIZE PAGE
LOCKMAX SYSTEM
BUFFERPOOL BP0
CLOSE NO
CCSID EBCDIC;

CREATE LOB TABLESPACE DSN8S12M
IN DSN8D12L
LOG NO;

CREATE LOB TABLESPACE DSN8S12L
IN DSN8D12L
LOG NO;

CREATE LOB TABLESPACE DSN8S12N
IN DSN8D12L
LOG NO;

CREATE TABLESPACE DSN8S12C
IN DSN8D12P
USING STOGROUP DSN8G120
PRIQTY 160
SECQTY 80
SEGSIZE 4
LOCKSIZE TABLE
BUFFERPOOL BP0
CLOSE NO
CCSID EBCDIC;

CREATE TABLESPACE DSN8S12P
IN DSN8D12A
USING STOGROUP DSN8G120
PRIQTY 160
SECQTY 80
SEGSIZE 4
LOCKSIZE ROW
BUFFERPOOL BP0
CLOSE NO
CCSID EBCDIC;

CREATE TABLESPACE DSN8S12R
IN DSN8D12A
USING STOGROUP DSN8G120
PRIQTY 20
SECQTY 20
ERASE NO
LOCKSIZE PAGE LOCKMAX SYSTEM
BUFFERPOOL BP0
CLOSE NO
CCSID EBCDIC;

CREATE TABLESPACE DSN8S1Z
IN DSN8D12A
USING STOGROUP DSN8G120
PRIQTY 20
SECQTY 20
ERASE NO
LOCKSIZE PAGE LOCKMAX SYSTEM
BUFFERPOOL BP0
CLOSE NO
CCSID EBCDIC;

CREATE TABLESPACE DSN8S81Q
IN DSN8D81P
USING STOGROUP DSN8G810
PRIQTY 160
SECQTY 80
SEGSIZE 4
LOCKSIZE PAGE
BUFFERPOOL BP0
CLOSE NO
CCSID EBCDIC;

CREATE TABLESPACE DSN8S81U
IN DSN8D81E
USING STOGROUP DSN8G810
PRIQTY 5
SECQTY 5
ERASE NO
LOCKSIZE PAGE LOCKMAX SYSTEM
BUFFERPOOL BP0
CLOSE NO
CCSID UNICODE;

SYSDUMMYx tables

Db2 for z/OS provides a set of SYSDUMMYx catalog tables.

The last character of the table name identifies the associated encoding scheme as follows:

- SYSIBM.SYSDUMMY1 uses the EBCDIC encoding scheme.
- SYSIBM.SYSDUMMYE uses the EBCDIC encoding scheme.
- SYSIBM.SYSDUMMYA uses the ASCII encoding scheme.
- SYSIBM.SYSDUMMYU uses the UNICODE encoding scheme.

Although the SYSDUMMYx tables are implemented as catalog tables, they are similar to sample tables, and are used in some examples in the Db2 for z/OS documentation.

You can use any of the SYSDUMMYx tables when you need to write a query, but no data from a table is referenced. In any query, you must specify a table reference in the FROM clause, but if the query does not reference any data from the table, it does not matter which table is referenced in the FROM clause. Each of the SYSDUMMYx tables contains one row, so a SYSDUMMYx table can be referenced in a SELECT INTO statement without the need for a predicate to limit the result to a single row of data.

For example, when you want to retrieve the value of a special register or global variable, you can use a query that references a SYSDUMMYx table.

```
SELECT CURRENT PATH -- Retrieve the value of a special register
  INTO :myvar
  FROM SYSIBM.SYSDUMMY1;

SELECT TEMPORAL_LOGICAL_TRANSACTION_TIME -- Retrieve the value of a special register
  INTO :myvar
  FROM SYSIBM.SYSDUMMY1;
```
Sometimes when Db2 for z/OS processes an SQL statement, the statement is rewritten, and a reference to a SYSDUMMYx table is added. For example, some of the internal rewrites that result in adding a reference to a SYSDUMMYx table are for processing the search condition of a trigger, or the SQL PL control statements IF, REPEAT, RETURN, or WHILE. In these situations, the privilege set must include the SELECT privilege on the SYSDUMMYx table that is referenced.

Related concepts
Objects with different CCSIDs in the same SQL statement (Db2 Internationalization Guide)

Related tasks
Avoiding character conversion for LOB locators
In certain situations, Db2 materializes the entire LOB value and converts it to the encoding scheme of a particular SQL statement. This extra processing can degrade performance and should be avoided.

Related reference
SYSDUMMY1 catalog table (Db2 SQL)
SYSDUMMYA catalog table (Db2 SQL)
SYSDUMMYE catalog table (Db2 SQL)
SYSDUMMYU catalog table (Db2 SQL)

Db2 productivity-aid sample programs

Db2 provides four sample programs that many users find helpful as productivity aids. These programs are shipped as source code, so you can modify them to meet your needs.

DSNTIAUL
DSNTIAUL is a sample program for unloading data, as an alternative to the UNLOAD utility. DSNTIAUL is written in the assembler language. DSNTIAUL can unload some or all rows from up to 100 Db2 tables. With DSNTIAUL, you can unload data of any Db2 built-in data type or distinct type. DSNTIAUL unloads the rows in a form that is compatible with the LOAD utility and generates utility control statements for LOAD. You can also used DSNTIAUL to execute any SQL non-SELECT statement that can be executed dynamically.

DSNTIAD
The DSNTIAD sample program can issue any SQL statement that can be executed dynamically, except for SELECT statements. DSNTIAD is written in assembler language.

DSNTEP2
DSNTEP2 is a sample dynamic SQL program that can issue any SQL statement that can be executed dynamically. DSNTEP2 is written in PL/I language and available in two versions: a source version that you can modify to meet your needs or an object code version that you can use without the need for a PL/I compiler.

DSNTEP4
The DSNTEP4 sample program is identical to DSNTEP2, except that it uses multi-row fetch for increased performance. DSNTEP4 is written in PL/I language and available in two versions: a source version that you can modify to meet your needs or an object code version that you can use without the need for a PL/I compiler.

Because these four programs also accept the static SQL statements CONNECT, SET CONNECTION, and RELEASE, you can use the programs to access Db2 tables at remote locations.

Preparing the productivity-aid sample programs

DSNTIAUL and DSNTIAD are shipped only as source code, so you must precompile, assemble, link, and bind them before you can use them. If you want to use the source code version of DSNTEP2 or DSNTEP4, you must precompile, compile, link, and bind it. You need to bind the object code version of DSNTEP2 or DSNTEP4 before you can use them. Usually a system administrator prepares the programs as part of the installation process. The following table indicates the installation jobs that prepare each sample program. All installation jobs are in data set DSN1210.SDSNSAMP.
Table 176. Jobs that prepare DSNTIAUL, DSNTIAD, DSNTEP2, and DSNTEP4

<table>
<thead>
<tr>
<th>Program name</th>
<th>Program preparation job</th>
</tr>
</thead>
<tbody>
<tr>
<td>DSNTIAUL</td>
<td>DSNTEJ2A</td>
</tr>
<tr>
<td>DSNTIAD</td>
<td>DSNTEJ1TM</td>
</tr>
<tr>
<td>DSNTEP2 (source)</td>
<td>DSNTEJ1P</td>
</tr>
<tr>
<td>DSNTEP2 (object)</td>
<td>DSNTEJ1L</td>
</tr>
<tr>
<td>DSNTEP4 (source)</td>
<td>DSNTEJ1P</td>
</tr>
<tr>
<td>DSNTEP4 (object)</td>
<td>DSNTEJ1L</td>
</tr>
</tbody>
</table>

Application compatibility for the productivity-aid sample programs

If you are ready for the productivity-aid sample programs to start using new application capabilities in later Db2 12 function levels, you must rebind the packages for these programs with the corresponding APPLCOMPAT option.

The Db2 12 jobs for preparing the productivity-aid sample programs bind the packages with APPLCOMPAT(V12R1) by default, which is equivalent to APPLCOMPAT(V12R1M500).

For more information, see Controlling Db2 application compatibility (Db2 for z/OS What's New?) and Function levels and related levels in Db2 12 (Db2 for z/OS What's New?).

Running the productivity-aid sample programs

To run the sample programs, use the RUN (DSN) command. For more information, see RUN (DSN) (Db2 Commands) and the following topics.

Retrieval of UTF-16 Unicode data by DSNTEP2, DSNTEP4, and DSNTIAUL

You can use DSNTEP2, DSNTEP4, and DSNTIAUL to retrieve Unicode UTF-16 graphic data. However, these programs might not be able to display some characters, if those characters have no mapping in the target SBCS EBCDIC CCSID.

Related reference
- RUN (DSN) (Db2 Commands)
- Db2 for z/OS Exchange

DSNTIAUL sample program

You can use the DSNTIAUL program to unload data from Db2 tables into sequential data sets. The data is copied to the data sets and is not deleted from the table.

DSNTIAUL is a sample program for unloading data, as an alternative to the UNLOAD utility. DSNTIAUL is written in the assembler language. DSNTIAUL can unload some or all rows from up to 100 Db2 tables. With DSNTIAUL, you can unload data of any Db2 built-in data type or distinct type. DSNTIAUL unloads the rows in a form that is compatible with the LOAD utility and generates utility control statements for LOAD. You can also used DSNTIAUL to execute any SQL non-SELECT statement that can be executed dynamically.

When multi-row fetch is used, parallelism might be disabled in the last parallel group in the top-level query block for a query. For very simple queries, parallelism might be disabled for the entire query when multi-row fetch is used. To obtain full parallelism when running DSNTIAUL, switch DSNTIAUL to single-row fetch mode by specifying 1 for the number-of-rows-per-fetch parameter.

DSNTIAUL uses SQL to access Db2. Operations on a row-level or column-level access control enforced table are subject to the rules specified for the access control. If the table is row-level access control enforced, DSNTIAUL receives and returns only the rows of the table that satisfy the row permissions for
If the table is column-level access control enforced, DSNTIAUL receives and returns the values in the column values as modified by the column masks for the user.

**Important:** To avoid substitution characters in unloaded data, do not use DSNTIAUL to unload an EBCDIC table that contains a Unicode column.

### Preparing the DSNTIAUL sample program

Before you can use the DSNTIAUL sample program, you must precompile, assemble, link, and bind it first. Also, when you are ready for DSNTIAUL to start using to use new capabilities in later Db2 12 function levels, you must rebind the packages at the corresponding APPLCOMPAT level.

For more information, see “Db2 productivity-aid sample programs” on page 1008.

### Running the DSNTIAUL sample program

To run the DSNTIAUL sample program, use the RUN (DSN) command and specify the following load module and plan name.

<table>
<thead>
<tr>
<th>Load module name</th>
<th>Plan name</th>
</tr>
</thead>
<tbody>
<tr>
<td>DSNTIAUL</td>
<td>DSNTIBC1</td>
</tr>
</tbody>
</table>

For more information about the RUN command, see RUN (DSN) (Db2 Commands).

### DSNTIAUL parameters

**PKGSET**(collection)**

Specifies that DSNTIAUL implicitly executes a SET CURRENT PACKAGESET statement to assign a value to the CURRENT PACKAGESET special register before processing the dynamic SQL statements in SYSIN.

- **collection**
  
  The value to assign to the CURRENT PACKAGESET special register. You can specify up to 40 characters.

**SQL**

Specify SQL to indicate that your input data set contains one or more complete SQL statements, each of which ends with a semicolon. You can include any SQL statement that can be executed dynamically in your input data set. In addition, you can include the static SQL statements CONNECT, SET CONNECTION, or RELEASE. Static SQL statements must be uppercase.

DSNTIAUL uses the SELECT statements to determine which tables to unload and dynamically executes all other statements except CONNECT, SET CONNECTION, and RELEASE. DSNTIAUL executes CONNECT, SET CONNECTION, and RELEASE statically to connect to remote locations.

**number-of-rows-per-fetch**

Specify a number from 1 to 32767 to specify the number of rows that DSNTIAUL retrieves for each SQL FETCH operation. If you do not specify this number, DSNTIAUL retrieves 100 rows for each FETCH. This parameter can be specified with the SQL parameter.

If the LOBFILE parameter is also specified, and the result set of a FETCH operation can contain NULL LOB values, **number-of-rows-per-fetch** must be 1.

**TOLWARN**

Specify NO (the default) or YES to indicate whether DSNTIAUL continues to retrieve rows after receiving an SQL warning:

- **(NO)**
  
  If a warning occurs when DSNTIAUL executes an OPEN or FETCH to retrieve rows, DSNTIAUL stops retrieving rows. If the SQLWARN1, SQLWARN2, SQLWARN6, or SQLWARN7 flag is set when DSNTIAUL executes a FETCH to retrieve rows, DSNTIAUL continues to retrieve rows.
If a warning occurs when DSNTIAUL executes an OPEN or FETCH to retrieve rows, DSNTIAUL continues to retrieve rows.

The same as YES except that the program suppresses all SQL warning messages from OPEN or FETCH statements if the SQLCODE is 0 or greater.

Specify LOBFILE to indicate that you want DSNTIAUL to dynamically allocate data sets, each to receive the full content of a LOB cell. (A LOB cell is the intersection of a row and a LOB column.) If you do not specify the LOBFILE option, you can unload up to only 32 KB of data from a LOB column.

Specify a high-level qualifier for these dynamically allocated data sets. You can specify up to 17 characters. The qualifier must conform with the rules for TSO data set names.

DSNTIAUL uses a naming convention for these dynamically allocated data sets of prefix.Qiiiiii.Cjjjjjjj.Rkkkkkkk, where these qualifiers have the following values:

The high-level qualifier that you specify in the LOBFILE option.

The sequence number (starting from 0) of a query that returns one or more LOB columns

The sequence number (starting from 0) of a column in a query that returns one or more LOB columns

The sequence number (starting from 0) of a row of a result set that has one or more LOB columns.

The generated LOAD statement contains LOB file reference variables that can be used to load data from these dynamically allocated data sets.

If you do not specify the SQL parameter, your input data set must contain one or more single-line statements (without a semicolon) that use the following syntax:

```
table or view name [WHERE conditions] [ORDER BY columns]
```

Each input statement must be a valid SQL SELECT statement with the clause SELECT * FROM omitted and with no ending semicolon. DSNTIAUL generates a SELECT statement for each input statement by appending your input line to SELECT * FROM, then uses the result to determine which tables to unload. For this input format, the text for each table specification can be a maximum of 72 bytes and must not span multiple lines.

You can use the input statements to specify SELECT statements that join two or more tables or select specific columns from a table. If you specify columns, you need to modify the LOAD statement that DSNTIAUL generates.

### DSNTIAUL data sets

<table>
<thead>
<tr>
<th>Data set</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SYSIN</td>
<td>Input data set.</td>
</tr>
</tbody>
</table>

If you specify the SQL parameter, you can enter bracketed comments in DSNTIAUL input that includes dynamic SQL statements. Bracketed comments are not supported if the input includes the static SQL statements CONNECT, SET CONNECTION, or RELEASE. Bracketed comments begin with `/*` and end with `*/`.

The record length for the input data set must be at least 72 bytes. DSNTIAUL reads only the first 72 bytes of each record.
**SYSPRINT**
Output data set. DSNTIAUL writes informational and error messages in this data set.

The record length for the SYSPRINT data set is 121 bytes.

**SYSPUNCH**
Output data set. DSNTIAUL writes the LOAD utility control statements in this data set.

**SYSRECnn**
Output data sets. The value \( nn \) ranges from 00 to 99. You can have a maximum of 100 output data sets for a single execution of DSNTIAUL. Each data set contains the data that is unloaded when DSNTIAUL processes a SELECT statement from the input data set. Therefore, the number of output data sets must match the number of SELECT statements (if you specify parameter SQL) or table specifications in your input data set.

Define all data sets as sequential data sets. You can specify the record length and block size of the SYSPUNCH and SYSRECnn data sets. The maximum record length for the SYSPUNCH and SYSRECnn data sets is 32760 bytes.

**DSNTIAUL return codes**

<table>
<thead>
<tr>
<th>Return code</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Successful completion.</td>
</tr>
<tr>
<td>4</td>
<td>An SQL statement received a warning code.</td>
</tr>
<tr>
<td></td>
<td>• If TOLWARN(YES) is specified, and the warning occurred on a FETCH or OPEN during the processing of a SELECT statement, Db2 performs the unload operation.</td>
</tr>
<tr>
<td></td>
<td>• Otherwise if the SQL statement was a SELECT statement, Db2 did not perform the associated unload operation.</td>
</tr>
<tr>
<td></td>
<td>If Db2 returns a +394, which indicates that it is using optimization hints, or a +395, which indicates one or more invalid optimization hints, Db2 performs the unload operation.</td>
</tr>
<tr>
<td>8</td>
<td>An SQL statement received an error code. If the SQL statement was a SELECT statement, Db2 did not perform the associated unload operation or did not complete it.</td>
</tr>
<tr>
<td>12</td>
<td>DSNTIAUL could not open a data set, an SQL statement returned a severe error code (-144, -302, -804, -805, -818, -902, -906, -911, -913, -922, -923, -924, or -927), or an error occurred in the SQL message formatting routine.</td>
</tr>
</tbody>
</table>

**Examples**

**Example: using DSNTIAUL to unload a subset of rows in a table**

Suppose that you want to unload the rows for department D01 from the project table. Because you can fit the table specification on one line, and you do not want to execute any non-SELECT statements, you do not need the SQL parameter. Your invocation looks like the one that is shown in the following figure:

```bash
//UNLOAD EXEC PGM=IKJEFT01, DYNAMNBR=20
//SYSTSPRT DD SYSOUT=*  
//SYSTSIN DD * 
//DSN SYSTEM(DSN) 
//RUN PROGRAM(DSNTIAUL) PLAN(DSNTIBC1) LIB('DSN1210.RUNLIB.LOAD') 
//SYSPRINT DD SYSOUT=*
//SYSUDUMP DD SYSOUT=*
//SYSREC00 DD DSN=DSN8UNLD.SYSREC00,UNIT=SYSDA, SPACE=(32760,(1000,500)), DISP=(,CATLG), VOL=SER=SCR03 
```
Example: using DSNTIAUL to unload rows in more than one table

Suppose that you also want to use DSNTIAUL to perform the following actions:

- Unload all rows from the project table
- Unload only rows from the employee table for employees in departments with department numbers that begin with D, and order the unloaded rows by employee number
- Lock both tables in share mode before you unload them
- Retrieve 250 rows per fetch

For these activities, you must specify the SQL parameter and the number-of-rows-per-fetch parameter when you run DSNTIAUL. Your DSNTIAUL invocation is shown in the following figure:

Example: using DSNTIAUL to obtain LOAD utility control statements

If you want to obtain the LOAD utility control statements for loading rows into a table, but you do not want to unload the rows, you can set the data set names for the SYSRECnn data sets to DUMMY. For example, to obtain the utility control statements for loading rows into the department table, you invoke DSNTIAUL as shown in the following figure:

Example: using DSNTIAUL to unload LOB data

This example uses the sample LOB table with the following structure:

```sql
CREATE TABLE DSN8C10.EMP_PHOTO_RESUME
( EMPNO CHAR(06) NOT NULL,
  EMP_ROWID ROWID NOT NULL GENERATED ALWAYS,
  EMP_PHOTO_RESUME BLOB)
```
The following call to DSNTIAUL unloads the sample LOB table. The parameters for DSNTIAUL indicate the following options:

- The SQL parameter specifies that the input data set (SYSIN) contains SQL.
- The number-of-rows-per-fetch parameter value of 1 specifies that DSNTIAUL is to retrieve one row for each FETCH operation. A value of 1 is necessary if the LOB columns that you unload might contain NULL values.
- The LOBFILE parameter value of LOBFILE(DSN8UNLD) specifies that DSNTIAUL places the LOB data in data sets with a high-level qualifier of DSN8UNLD.

```sql
//UNLOAD   EXEC PGM=IKJEFT01,DYNAMNBR=20
//SYSTSIN   DD *
DSN SYSTEM(DSN)
RUN PROGRAM(DSNTIAUL) PLAN(DSNTIB91) -
PARMS('SQL,1,LOBFILE(DSN8UNLD)') -
LIB('DSN1210.RUNLIB.LOAD')
//SYSPRINT  DD SYSOUT=*  
//SYSUDUMP  DD SYSOUT=*  
//SYSREC00 DD DSN=DSN8UNLD.SYSREC00,
//UNIT=SYSDA,SPACE=(800,(15,15)),DISP=(,CATLG),
//VOL=SER=SCR03,RECFM=FB
//SYSPUNCH DD DSN=DSN8UNLD.SYSPUNCH,
//UNIT=SYSDA,SPACE=(800,(15,15)),DISP=(,CATLG),
//VOL=SER=SCR03,RECFM=FB
//SYSSIN   DD *
SELECT * FROM DSN8C10.EMP_PHOTO_RESUME;
```

Given that the sample LOB table has 4 rows of data, DSNTIAUL produces the following output:

- Data for columns EMPNO and EMP_ROWID are placed in the data set that is allocated according to the SYSREC00 DD statement. The data set name is DSN8UNLD.SYSREC00.
- A generated LOAD statement is placed in the data set that is allocated according to the SYSPUNCH DD statement. The data set name is DSN8UNLD.SYSPUNCH.
- The following data sets are dynamically created to store LOB data:
  - DSN8UNLD.Q0000000.C0000002.R0000000
  - DSN8UNLD.Q0000000.C0000002.R0000001
  - DSN8UNLD.Q0000000.C0000002.R0000002
  - DSN8UNLD.Q0000000.C0000002.R0000003
  - DSN8UNLD.Q0000000.C0000003.R0000000
  - DSN8UNLD.Q0000000.C0000003.R0000001
  - DSN8UNLD.Q0000000.C0000003.R0000002
  - DSN8UNLD.Q0000000.C0000003.R0000003
  - DSN8UNLD.Q0000000.C0000004.R0000000
  - DSN8UNLD.Q0000000.C0000004.R0000001
  - DSN8UNLD.Q0000000.C0000004.R0000002
  - DSN8UNLD.Q0000000.C0000004.R0000003
For example, DSN8UNLD.Q0000000.C0000004.R0000001 means that the data set contains data that is unloaded from the second row (R0000001) and the fifth column (C0000004) of the result set for the first query (Q0000000).
DSNTIAD sample program

You can use the DSNTIAD program to execute dynamic SQL statements other than SELECT statements. The DSNTIAD sample program can issue any SQL statement that can be executed dynamically, except for SELECT statements. DSNTIAD is written in assembler language.

Preparing the DSNTIAD sample program

Before you can use the DSNTIAD sample program, you must precompile, assemble, link, and bind it first. Also, when you are ready for DSNTIAD to start using to use new capabilities in later Db2 12 function levels, you must rebind the packages at the corresponding APPLCOMPAT level. For more information, see “Db2 productivity-aid sample programs” on page 1008.

Running the DSNTIAD sample program

To run the DSNTIAD sample program, use the RUN (DSN) command and specify the following load module and plan name.

<table>
<thead>
<tr>
<th>Load module name</th>
<th>DSNTIAD</th>
</tr>
</thead>
<tbody>
<tr>
<td>Plan name</td>
<td>DSNTIAC1</td>
</tr>
</tbody>
</table>

For more information about the RUN command, see RUN (DSN) (Db2 Commands).

DSNTIAD parameters

**PKGSET(collection)**

Specifies that DSNTIAD implicitly executes a SET CURRENT PACKAGESET statement to assign a value to the CURRENT PACKAGESET special register before processing the dynamic SQL statements in SYSIN.

- **collection**
  - The value to assign to the CURRENT PACKAGESET special register. You can specify up to 40 characters.

**RC0**

If you specify this parameter, DSNTIAD ends with return code 0, even if the program encounters SQL errors. If you do not specify RC0, DSNTIAD ends with a return code that reflects the severity of the errors that occur. Without RC0, DSNTIAD terminates if more than 10 SQL errors occur during a single execution.

**SQLTERM(termchar)**

Specify this parameter to indicate the character that you use to end each SQL statement. You can use any special character except one of those listed in the following table. SQLTERM(;) is the default.

<table>
<thead>
<tr>
<th>Name</th>
<th>Character</th>
<th>Hexadecimal representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>blank</td>
<td>\x20</td>
<td>X'40'</td>
</tr>
<tr>
<td>comma</td>
<td>,</td>
<td>X'6B'</td>
</tr>
<tr>
<td>double quotation mark</td>
<td>&quot;</td>
<td>X'7F'</td>
</tr>
<tr>
<td>left parenthesis</td>
<td>(</td>
<td>X'4D'</td>
</tr>
<tr>
<td>right parenthesis</td>
<td>)</td>
<td>X'5D'</td>
</tr>
<tr>
<td>single quotation mark</td>
<td>'</td>
<td>X'7D'</td>
</tr>
<tr>
<td>underscore</td>
<td>_</td>
<td>X'6D'</td>
</tr>
</tbody>
</table>
Use a character other than a semicolon if you plan to execute a statement that contains embedded semicolons.

For example, suppose that you specify the parameter SQLTERM(#) to indicate that the character # is the statement terminator. Then a CREATE TRIGGER statement with embedded semicolons looks like this:

```sql
CREATE TRIGGER NEW_HIRE
    AFTER INSERT ON EMP
    FOR EACH ROW MODE DB2SQL
    BEGIN ATOMIC
        UPDATE COMPANY_STATS SET NBEMP = NBEMP + 1;
    END#
```

A CREATE PROCEDURE statement with embedded semicolons looks like the following statement:

```sql
CREATE PROCEDURE PROC1 (IN PARM1 INT, OUT SCODE INT)
    LANGUAGE SQL
    BEGIN
        DECLARE SQLCODE INT;
        DECLARE EXIT HANDLER FOR SQLEXCEPTION
            SET SCODE = SQLCODE;
        UPDATE TBL1 SET COL1 = PARM1;
    END #
```

Be careful to choose a character for the statement terminator that is not used within the statement.

**DSNTIAD data sets**

**Data set**

**Description**

**SYSIN**

Input data set. In this data set, you can enter any number of non-SELECT SQL statements. Each SQL statement must be terminated with the SQL termination character. If you specify the SQLTERM(termchar) parameter, termchar is the SQL termination character. Otherwise, the SQL termination character is a semicolon. A statement can span multiple lines, but DSNTIAD reads only the first 72 bytes of each line.

Comments in DSNTIAD input are not supported.

**SYSPRINT**

Output data set. DSNTIAD writes informational and error messages in this data set. DSNTIAD sets the record length of this data set to 121 bytes and the block size to 1210 bytes.

Define all data sets as sequential data sets.

**DSNTIAD return codes**

<table>
<thead>
<tr>
<th>Return code</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Successful completion, or the user-specified parameter RC0.</td>
</tr>
<tr>
<td>4</td>
<td>An SQL statement received a warning code.</td>
</tr>
<tr>
<td>8</td>
<td>An SQL statement received an error code.</td>
</tr>
<tr>
<td>12</td>
<td>DSNTIAD could not open a data set, the length of an SQL statement was more than 2 MB, an SQL statement returned a severe error code (-8nn or -9nn), or an error occurred in the SQL message formatting routine.</td>
</tr>
</tbody>
</table>
Example: invoking the DSNTIAD program

Suppose that you want to execute 20 UPDATE statements, and you do not want DSNTIAD to terminate if more than 10 errors occur. Your invocation looks like the one that is shown in the following figure:

```plaintext
//RUNTIAD EXEC PGM=IKJEFT01,DYNAMNBR=20
//SYSTSPRT DD SYSOUT=*  
//SYSTSPR  DD *  
//DSN SYSTEM(DSN)
  RUN PROGRAM(DSNTIAD) PLAN(DSNTIAC1) PARMS('RC0') -  
  LIB('DSN1210.RUNLIB.LOAD')  
//SYSPRINT DD SYSOUT=*  
//SYSUDUMP DD SYSOUT=*  
//SYSIN    DD *  
UPDATE DSN8C10.PROJ SET DEPTNO='J01' WHERE DEPTNO='A01';  
UPDATE DSN8C10.PROJ SET DEPTNO='J02' WHERE DEPTNO='A02';  
⋮  
UPDATE DSN8C10.PROJ SET DEPTNO='J20' WHERE DEPTNO='A20';
```

DSNTEP2 and DSNTEP4 sample programs

You can use the DSNTEP2 or DSNTEP4 programs to execute SQL statements dynamically.

DSNTEP2 is a sample dynamic SQL program that can issue any SQL statement that can be executed dynamically. DSNTEP2 is written in PL/I language and available in two versions: a source version that you can modify to meet your needs or an object code version that you can use without the need for a PL/I compiler.

The DSNTEP4 sample program is identical to DSNTEP2, except that it uses multi-row fetch for increased performance. DSNTEP4 is written in PL/I language and available in two versions: a source version that you can modify to meet your needs or an object code version that you can use without the need for a PL/I compiler.

When multi-row fetch is used, parallelism might be disabled for the last parallel group in the top-level query block, or entirely disabled for very simple queries. To obtain full parallelism, use DSNTEP2 or specify the control option SET MULT_FETCH 1 for DSNTEP4.

DSNTEP2 and DSNTEP4 write their results to the data set that is defined by the SYSPRINT DD statement. SYSPRINT data must have a logical record length that matches the PAGEWIDTH value in the DSNTEP2 or DSNTEP4 source program. If you use the original version of the program that is shipped with Db2, the logical record length is 133 bytes. If the SYSPRINT data do not have the same logical record length as the PAGEWIDTH value, the program issues return code 12 with abend U4038 and reason code 1. This abend occurs due to the PL/I file exception error IBM0201S ONCODE=81. The following error message is issued:

```
The UNDEFINEDFILE condition was raised because of conflicting DECLARE and OPEN attributes (FILE= SYSPRINT).
```

If you use applications or other automation to process output from DSNTEP2 or DSNTEP4, be aware that minor changes in the format can occur as a result of service or enhancements. Such changes might require you to adjust your processes that use the output of these programs.

**Important:** When you allocate a new data set with the SYSPRINT DD statement, either specify a DCB with RECFM=FBA and LRECL=nnnn, where nnn matches the PAGEWIDTH value in the source program, or do not specify the DCB parameter.

Preparing the DSNTEP2 and DSNTEP4 sample programs

Before you can use the DSNTEP2 and DSNTEP4 sample programs, you must prepare them. If you use the source-code versions of DSNTEP2 or DSNTEP4, you must precompile, compile, link, and bind them first. If you use the object-code versions of DSNTEP2 or DSNTEP4 you must bind them first.

Also, when you are ready for DSNTEP2 or DSNTEP4 to start using to use new capabilities in later Db2 12 function levels, you must rebind the packages at the corresponding APPLCOMPAT level.

For more information, see "Preparing the productivity-aid sample programs" in “Db2 productivity-aid sample programs” on page 1008.
Running the DSNTEP2 and DSNTEP4 sample programs

To run the DSNTEP2 and DSNTEP4 sample programs, use the RUN (DSN) command and specify the following load module and plan name.

DSNTEP2 load module and plan

<table>
<thead>
<tr>
<th>Load module</th>
<th>DSNTEP2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Plan</td>
<td>DSNTEPC1</td>
</tr>
</tbody>
</table>

DSNTEP4 load module and plan

<table>
<thead>
<tr>
<th>Load module</th>
<th>DSNTEP4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Plan</td>
<td>DSNTP412</td>
</tr>
</tbody>
</table>

For more information about the RUN command, see RUN (DSN) (Db2 Commands).

DSNTEP2 and DSNTEP4 parameters

PKGSET(collection)

Specifies that DSNTEP2 or DSNTEP4 implicitly executes a SET CURRENT PACKAGESET statement to assign a value to the CURRENT PACKAGESET special register before processing the dynamic SQL statements in SYSIN.

collection

The value to assign to the CURRENT PACKAGESET special register. You can specify up to 40 characters.

For an example, see “Example: Running dynamic SQL statements at different application compatibility levels in the same SYSIN” on page 1023.

ALIGN(MID) or ALIGN(LHS)

Specifies the alignment.

ALIGN(MID)

Specifies that DSNTEP2 or DSNTEP4 output should be centered. ALIGN(MID) is the default.

ALIGN(LHS)

Specifies that the DSNTEP2 or DSNTEP4 output should be left-justified.

NOMIXED or MIXED

Specifies whether DSNTEP2 or DSNTEP4 contains any DBCS characters.

NOMIXED

Specifies that the DSNTEP2 or DSNTEP4 input contains no DBCS characters. NOMIXED is the default.

MIXED

Specifies that the DSNTEP2 or DSNTEP4 input contains some DBCS characters.

PREPWARN

Specifies that DSNTEP2 or DSNTEP4 is to display details about any SQL warnings that are encountered at PREPARE time.

Regardless of whether you specify PREPWARN, when an SQL warning is encountered at PREPARE time, the program displays the message SQLWARNING ON PREPARE and sets the return code to 4. When you specify PREPWARN, the program also displays the details about any SQL warnings.

SQLFORMAT

Specifies how DSNTEP2 or DSNTEP4 pre-processes SQL statements before passing them to Db2. Select one of the following options:
This is the preferred mode for SQL statements other than SQL procedural language. When you use this option, which is the default, DSNTEP2 or DSNTEP4 collapses each line of an SQL statement into a single line before passing the statement to Db2. DSNTEP2 or DSNTEP4 also discards all SQL comments.

This mode is suitable for all SQL, but it is intended primarily for SQL procedural language processing. When this option is in effect, behavior is similar to SQL mode, except that DSNTEP2 or DSNTEP4 does not discard SQL comments. Instead, it automatically terminates each SQL comment with a line feed character (hex 25), unless the comment is already terminated by one or more line formatting characters. Use this option to process SQL procedural language with minimal modification by DSNTEP2 or DSNTEP4.

This mode is suitable for all SQL, but it is intended primarily for SQL procedural language processing. When this option is in effect, DSNTEP2 or DSNTEP4 retains SQL comments and terminates each line of an SQL statement with a line feed character (hex 25) before passing the statement to Db2. Lines that end with a split token are not terminated with a line feed character. Use this mode to obtain improved diagnostics and debugging of SQL procedural language.

Specifies the character that you use to end each SQL statement. You can use any character except one of those that are listed in Table 178 on page 1015. SQLTERM(;) is the default.

A character other than a semicolon if you plan to execute a statement that contains embedded semicolons.

See “Example: changing the SQL terminator” on page 1023.

Indicates whether DSNTEP2 or DSNTEP4 continues to process SQL SELECT statements after receiving an SQL warning. You can specify one of the following values:

- **NO**
  Indicates that the program stops processing the SELECT statement if a warning occurs when the program executes an OPEN or FETCH for a SELECT statement. NO is the default value for TOLWARN.

  The following exceptions exist:

  - If SQLCODE +445 or SQLCODE +595 occurs when DSNTEP2 or DSNTEP4 executes a FETCH for a SELECT statement, the program continues to process the SELECT statement.
  - If SQLCODE +354 occurs when DSNTEP4 executes a FETCH for a SELECT statement, the program continues to process the SELECT statement.
  - If SQLCODE +802 occurs when DSNTEP2 or DSNTEP4 executes a FETCH for a SELECT statement, the program continues to process the SELECT statement if the TOLARTHWRN control statement is set to YES.

- **YES**
  Indicates that the program continues to process the SELECT statement if a warning occurs when the program executes an OPEN or FETCH for a SELECT statement.

The same as YES except that the program suppresses all SQL warning messages from OPEN or FETCH statements if the SQLCODE is 0 or greater.

If the DSNTEP2 or DSNTEP4 output formatting does not meet your requirements, you can change some variables in the source code to modify the formatting.
For example, if you want to increase the maximum width of a page, the maximum size of a print line, and the maximum number of characters in the output of a character column, you can increase the values of the PAGEWIDTH, MAXPAGWD, and MAXCOLWD variable values.

For a complete description of the settings that you can change, see the information under PROGRAM SIZES in the DSNTEP2 and DSNTEP4 prologs.

**DSNTEP2 and DSNTEP4 data sets**

The following data sets are used by DSNTEP2 and DSNTEP4:

**SYSIN**

Input data set. In this data set, you can enter any number of SQL statements, each terminated with a semicolon. A statement can span multiple lines, but DSNTEP2 or DSNTEP4 reads only the first 72 bytes of each line. You must explicitly commit any SQL statements except the last one.

You can enter comments in DSNTEP2 or DSNTEP4 input with an asterisk (*) in column 1 or two hyphens (-- ) anywhere on a line. Text that follows the asterisk is considered to be comment text. Text that follows two hyphens can be comment text or a control statement. Comments are not considered in dynamic statement caching. Comments and control statements cannot span lines.

You can enter control statements of the following form in the DSNTEP2 and DSNTEP4 input data set:

```
--#SET control-option value
```

You can specify the following control options. If you specify a value of NO for any of the options in this list, the program behaves as if you did not specify the parameter.

--- **--#PKGSET value**

Specifies that DSNTEP2 or DSNTEP4 implicitly executes a SET CURRENT PACKAGESET statement to assign a value to the CURRENT PACKAGESET special register before processing dynamic SQL statements after this control statement in SYSIN. *value* is the value to assign to CURRENT PACKAGESET special register. You can specify up to 40 characters.

For an example, see “Example: Running dynamic SQL statements at different application compatibility levels in the same SYSIN” on page 1023.

--- **--#TERMINATOR value**

The SQL statement terminator. *value* is any single-byte character other than one of those that are listed in “DSNTIAD sample program” on page 1015. The default is the value of the SQLTERM parameter.

See “Example: changing the SQL terminator within a series of SQL statements” on page 1023.

--- **--#ROWS_FETCH value**

The number of rows that are to be fetched from the result table. *value* is a numeric literal between -1 and the number of rows in the result table. -1 means that all rows are to be fetched. The default is -1.

--- **--#ROWS_OUT value**

The number of fetched rows that are to be sent to the output data set. *value* is a numeric literal between -1 and the number of fetched rows. -1 means that all fetched rows are to be sent to the output data set. The default is -1.

--- **--#MULT_FETCH value**

This option is valid only for DSNTEP4. Use MULT_FETCH to specify the number of rows that are to be fetched at one time from the result table. The default fetch amount for DSNTEP4 is 100 rows, but you can specify from 1 to 32676 rows.

--- **--#TOLWARN value**

Indicates whether DSNTEP2 or DSNTEP4 continues to process SQL SELECT statements after receiving an SQL warning. You can specify one of the following values:
NO
Indicates that the program stops processing the SELECT statement if a warning occurs when the program executes an OPEN or FETCH for a SELECT statement. NO is the default value for TOLWARN.

The following exceptions exist:

• If SQLCODE +445 or SQLCODE +595 occurs when DSNTEP2 or DSNTEP4 executes a FETCH for a SELECT statement, the program continues to process the SELECT statement.
• If SQLCODE +354 occurs when DSNTEP4 executes a FETCH for a SELECT statement, the program continues to process the SELECT statement.
• If SQLCODE +802 occurs when DSNTEP2 or DSNTEP4 executes a FETCH for a SELECT statement, the program continues to process the SELECT statement if the TOLARTHWRN control statement is set to YES.

YES
Indicates that the program continues to process the SELECT statement if a warning occurs when the program executes an OPEN or FETCH for a SELECT statement.

---#TOLARTHWRN value
Indicates whether DSNTEP2 and DSNTEP4 continue to process an SQL SELECT statement after an arithmetic SQL warning (SQLCODE +802) is returned. value is either NO (the default) or YES.

---#PREPWARN value
Specifies that DSNTEP2 or DSNTEP4 is to display details about any SQL warnings that are encountered at PREPARE time.

Regardless of whether you specify PREPWARN, when an SQL warning is encountered at PREPARE time, the program displays the message SQLWARNING ON PREPARE and sets the return code to 4. When you specify PREPWARN, the program also displays the details about any SQL warnings.

---#SQLFORMAT value
 Specifies how DSNTEP2 or DSNTEP4 pre-processes SQL statements before passing them to Db2. Select one of the following options:

SQL
This is the preferred mode for SQL statements other than SQL procedural language. When you use this option, which is the default, DSNTEP2 or DSNTEP4 collapses each line of an SQL statement into a single line before passing the statement to Db2. DSNTEP2 or DSNTEP4 also discards all SQL comments.

SQLCOMNT
This mode is suitable for all SQL, but it is intended primarily for SQL procedural language processing. When this option is in effect, behavior is similar to SQL mode, except that DSNTEP2 or DSNTEP4 does not discard SQL comments. Instead, it automatically terminates each SQL comment with a line feed character (hex 25), unless the comment is already terminated by one or more line formatting characters. Use this option to process SQL procedural language with minimal modification by DSNTEP2 or DSNTEP4.

SQLPL
This mode is suitable for all SQL, but it is intended primarily for SQL procedural language processing. When this option is in effect, DSNTEP2 or DSNTEP4 retains SQL comments and terminates each line of an SQL statement with a line feed character (hex 25) before passing the statement to Db2. Lines that end with a split token are not terminated with a line feed character. Use this mode to obtain improved diagnostics and debugging of SQL procedural language.

---#MAXERRORS value
value specifies that number of errors that DSNTEP2 and DSNTEP4 handle before processing stops. The default is 10. Use a value of -1 to indicate that a program is to tolerate an unlimited number of errors.
**SYSPRINT**
Output data set. DSNTEP2 and DSNTEP4 write informational and error messages in this data set. DSNTEP2 and DSNTEP4 write output records of no more than 133 bytes.

Define all data sets as sequential data sets.

**DSNTEP2 and DSNTEP4 return codes**

<table>
<thead>
<tr>
<th>Return code</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Successful completion.</td>
</tr>
<tr>
<td>4</td>
<td>An SQL statement received a warning code.</td>
</tr>
<tr>
<td>8</td>
<td>An SQL statement received an error code.</td>
</tr>
<tr>
<td>12</td>
<td>The length of an SQL statement was more than 2097152 bytes, an SQL statement returned a severe error code (-8nn or -9nn), or an error occurred in the SQL message formatting routine.</td>
</tr>
</tbody>
</table>

**Examples**

**Example: invoking DSNTEP2**
Suppose that you want to use DSNTEP2 to execute SQL SELECT statements that might contain DBCS characters. You also want left-aligned output. Your invocation looks like the following example.

```sql
//RUNTEP2 EXEC PGM=IKJEFT01,DYNAMNBR=20
//SYSTSPRT DD SYSOUT=*
//SYSTSIN DD *
DSN SYSTEM(DSN)
RUN PROGRAM(DSNTEP2) PLAN(DSNTEPC1) PARMS('/ALIGN(LHS) MIXED TOLWARN(YES)') -
LIB('DSN1210.RUNLIB.LOAD')
//SYSPRINT DD SYSOUT=*
//SYSSDUMP DD SYSOUT=*
//SYSIN DD *
SELECT * FROM DSN8C10.PROJ;
```

**Example: invoking DSNTEP4**
Suppose that you want to use DSNTEP4 to execute SQL SELECT statements that might contain DBCS characters, and you want center-aligned output. You also want DSNTEP4 to fetch 250 rows at a time. Your invocation looks like the one in the following figure:

```sql
//RUNTEP2 EXEC PGM=IKJEFT01,DYNAMNBR=20
//SYSTSPRT DD SYSOUT=*
//SYSTSIN DD *
DSN SYSTEM(DSN)
RUN PROGRAM(DSNTEP4) PLAN(DSNTEPC1) PARMS('/ALIGN(MID) MIXED') -
LIB('DSN1210.RUNLIB.LOAD')
//SYSPRINT DD SYSOUT=*
//SYSSDUMP DD SYSOUT=*
//SYSIN DD *
--#SET MULT_FETCH 250
SELECT * FROM DSN8C10.EMP;
```

**Example: Changing the application compatibility level of dynamic SQL statements**
Suppose that Db2 is at function level V12R1M508 or higher, but you want to use DSNTEP2 to run dynamic SQL statements at a lower application compatibility level. You can use the following example commands bind packages for DSNTEP2 with two different APPLCOMPAT options:

```sql
BIND PACKAGE (M503TEP2) MEMBER(DSN@EP2L) APPLCOMPAT(V12R1M503) +
CURRENTDATA(NO) ACT(REP) ISO(CS) ENCODING(EBCDIC)
BIND PACKAGE (M508TEP2) MEMBER(DSN@EP2L) APPLCOMPAT(V12R1M508) +
CURRENTDATA(NO) ACT(REP) ISO(CS) ENCODING(EBCDIC)
BIND PLAN(DSNTEP2) PKLIST(M508TEP2.,M503TEP2.) +
ACTION(REPLACE) RETAIN +
CURRENTDATA(NO) ISO(CS) ENCODING(EBCDIC) SQLRULES(DB2)
```
In this example, V12R1M508 is the default application compatibility level for DSNTEP2 because the package named M508TEP2 is bound with APPLCOMPAT(V12R1M508) and is listed first in the BIND PLAN step. However, you can also use DSNTEP2 to issue dynamic SQL statements at application compatibility level V12R1M503. To do so in this example, specify the following parameters when you run DSNTEP2.

```
RUN PROGRAM(DSNTEP2) PARMS('/PKGSET(M503TEP2)')
```

DSNTEP2 implicitly issues the following statement before it runs the dynamic SQL statements, and it runs dynamic SQL statements at application compatibility level V12R1M503:

```
SET CURRENT PACKAGESET = 'M503TEP2'
```

**Example: Running dynamic SQL statements at different application compatibility levels in the same SYSIN**

Assume that you issued the same BIND commands from the previous example, and suppose that you want to create a new multi-table segmented table space, which can only be created at application compatibility V12R1M503 or lower. As in the previous example, the default application compatibility level is V12R1M508, but you can use the following example --#SET PKGSET control statement to run some statements at a lower application compatibility in the same SYSIN.

```
--#SET PKGSET M503TEP2
CREATE TABLESPACE SEGTS1...
CREATE TABLE TB1 IN SEGTS1...
CREATE TABLE TB2 IN SEGTS1...
--#SET PKGSET M508TEP2
ALTER TABLESPACE SEGTS1 MOVE TABLE TB1 TO TABLESPACE PBGTS1...
ALTER TABLESPACE SEGTS1 MOVE TABLE TB2 TO TABLESPACE PBGTS2...
```

In this example, the first three statements succeed because the -# SET PKGSET control statement tells DSNTEP2 to run them at application compatibility level V12R1M503. Another -# SET PKGSET control statement changes the application compatibility level to V12R1M508 because ALTER TABLESPACE statements must run at this level or higher to specify the MOVE TABLE clause.

**Example: changing the SQL terminator**

Suppose that you specify the parameter SQLTERM(#) to indicate that the character # is the statement terminator. Then a CREATE TRIGGER statement with embedded semicolons looks like this:

```
CREATE TRIGGER NEW_HIRE
AFTER INSERT ON EMP
FOR EACH ROW MODE DB2SQL
BEGIN ATOMIC
  UPDATE COMPANY_STATS SET NBEMP = NBEMP + 1;
END#
```

A CREATE PROCEDURE statement with embedded semicolons looks like the following statement:

```
CREATE PROCEDURE PROC1 (IN PARM1 INT, OUT SCODE INT)
LANGUAGE SQL
BEGIN
  DECLARE SQLCODE INT;
  DECLARE EXIT HANDLER FOR SQLEXCEPTION
    SET SCODE = SQLCODE;
  UPDATE TBL1 SET COL1 = PARM1;
END #
```

Be careful to choose a character for the statement terminator that is not used within the statement.

**Example: changing the SQL terminator within a series of SQL statements**

Suppose that you have an existing set of SQL statements to which you want to add a CREATE TRIGGER statement that has embedded semicolons. You can use the --#SET TERMINATOR control statement. You can use the default SQLTERM value, which is a semicolon, for all of the existing SQL statements.
Before you execute the CREATE TRIGGER statement, include the --SET TERMINATOR # control statement to change the SQL terminator to the character #:

```sql
SELECT * FROM DEPT;
SELECT * FROM ACT;
SELECT * FROM EMPPROJECT;
SELECT * FROM PROJ;
SELECT * FROM PROJACT;
--SET TERMINATOR #
CREATE TRIGGER NEW_HIRE
AFTER INSERT ON EMP
FOR EACH ROW MODE DB2SQL
BEGIN ATOMIC
  UPDATE COMPANY_STATS SET NBEMP = NBEMP + 1;
END#
```

See the following discussion of the SYSIN data set for more information about the --SET control statement.

**Sample applications supplied with Db2 for z/OS**

Db2 provides sample applications to help you with Db2 programming techniques and coding practices within each of the four environments: batch, TSO, IMS, and CICS. The sample applications contain various applications that might apply to managing a company.

This topic describes the Db2 sample applications and the environments under which each application runs. It also provides information on how to use the applications, and how to print the application listings.

You can examine the source code for the sample application programs in the online sample library included with the Db2 product. The name of this sample library is DSN1210.SDSNSAMP.

**Using the sample applications**

You can use the applications interactively by accessing data in the sample tables on screen displays (panels). You can also access the sample tables in batch when using the phone applications. All sample objects have PUBLIC authorization, which makes the samples easier to run.

**Related concepts**

Db2 programming samples (Db2 Programming samples)

**Related reference**

Db2 for z/OS Exchange

**Types of sample applications**

Db2 provides a number of sample applications that manage sample company information. These applications also demonstrate how to use stored procedures, user-defined functions, and LOBs.

**Organization application:**

The organization application manages the following company information:

- Department administrative structure
- Individual departments
- Individual employees.

Management of information about department administrative structures involves how departments relate to other departments. You can view or change the organizational structure of an individual department, and the information about individual employees in any department. The organization application runs interactively in the ISPF/TSO, IMS, and CICS environments and is available in PL/I and COBOL.

**Project application:**

The project application manages information about a company’s project activities, including the following:
• Project structures
• Project activity listings
• Individual project processing
• Individual project activity estimate processing
• Individual project staffing processing.

Each department works on projects that contain sets of related activities. Information available about these activities includes staffing assignments, completion-time estimates for the project as a whole, and individual activities within a project. The project application runs interactively in IMS and CICS and is available in PL/I only.

Phone application:
The phone application lets you view or update individual employee phone numbers. There are different versions of the application for ISPF/TSO, CICS, IMS, and batch:

• ISPF/TSO applications use COBOL and PL/I.
• CICS and IMS applications use PL/I.
• Batch applications use C, C++, COBOL, FORTRAN, and PL/I.

Stored procedure applications:
There are three sets of stored procedure applications:

IFI applications
These applications let you pass Db2 commands from a client program to a stored procedure, which runs the commands at a Db2 server using the instrumentation facility interface (IFI). There are two sets of client programs and stored procedures. One set has a PL/I client and stored procedure; the other set has a C client and stored procedure.

ODBA application
This application demonstrates how you can use the IMS ODBA interface to access IMS databases from stored procedures. The stored procedure accesses the IMS sample DL/I database. The client program and the stored procedure are written in COBOL.

Utilities stored procedure application
This application demonstrates how to call the utilities stored procedure.

SQL procedure applications
Sample applications are available for both external SQL procedures and native SQL procedures:

• The applications for external SQL procedures demonstrate how to write, prepare, and invoke such procedures. One set of applications demonstrates how to prepare SQL procedures using JCL. The other set of applications shows how to prepare SQL procedures using the SQL procedure processor. The client programs are written in C.
• The sample job for a native SQL procedure shows how to prepare a native SQL procedure, how to manage versions of native SQL procedures, and optionally, how to deploy a native SQL procedure to a remote server. The sample also prepares and executes a sample caller in the C language.

WLM refresh application
This application is a client program that calls the Db2–supplied stored procedure WLM_REFRESH to refresh a WLM environment. This program is written in C.

System parameter reporting application
This application is a client program that calls the Db2–supplied stored procedure ADMIN_INFO_SYSPARM to display the current settings of system parameters. This program is written in C.

All stored procedure applications run in the TSO batch environment.
User-defined function applications:
The user-defined function applications consist of a client program that invokes the sample user-defined functions and a set of user-defined functions that perform the following functions:
• Convert the current date to a user-specified format
• Convert a date from one format to another
• Convert the current time to a user-specified format
• Convert a date from one format to another
• Return the day of the week for a user-specified date
• Return the month for a user-specified date
• Format a floating point number as a currency value
• Return the table name for a table, view, or alias
• Return the qualifier for a table, view or alias
• Return the location for a table, view or alias
• Return a table of weather information

All programs are written in C or C++ and run in the TSO batch environment.

LOB application:
The LOB application demonstrates how to perform the following tasks:
• Define Db2 objects to hold LOB data
• Populate Db2 tables with LOB data using the LOAD utility, or using INSERT and UPDATE statements when the data is too large for use with the LOAD utility
• Manipulate the LOB data using LOB locators

The programs that create and populate the LOB objects use DSNTIAD and run in the TSO batch environment. The program that manipulates the LOB data is written in C and runs under ISPF/TSO.

Application languages and environments for the sample applications
The sample applications demonstrate how to run Db2 applications in the TSO, IMS, or CICS environments.

The following table shows the environments under which each application runs, and the languages the applications use for each environment.

<table>
<thead>
<tr>
<th>Programs</th>
<th>ISPF/TSO</th>
<th>IMS</th>
<th>CICS</th>
<th>Batch</th>
<th>SPUFI</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dynamic SQL programs</td>
<td></td>
<td></td>
<td></td>
<td>Assembler</td>
<td></td>
</tr>
<tr>
<td>Exit routines</td>
<td>Assembler</td>
<td>Assembler</td>
<td>Assembler</td>
<td>Assembler</td>
<td>Assembler</td>
</tr>
<tr>
<td>Organization</td>
<td>COBOL</td>
<td>COBOL</td>
<td>COBOL</td>
<td>Assembler</td>
<td>Assembler</td>
</tr>
<tr>
<td>Phone</td>
<td>COBOL</td>
<td>PL/I</td>
<td>PL/I</td>
<td></td>
<td>COBOL</td>
</tr>
<tr>
<td></td>
<td>PL/I</td>
<td></td>
<td></td>
<td>FORTRAN</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Assembler</td>
<td></td>
<td></td>
<td>PL/I</td>
<td>C</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>C++</td>
<td></td>
</tr>
</tbody>
</table>
Table 181. Application languages and environments (continued)

<table>
<thead>
<tr>
<th>Programs</th>
<th>ISPF/TSO</th>
<th>IMS</th>
<th>CICS</th>
<th>Batch</th>
<th>SPUFI</th>
</tr>
</thead>
<tbody>
<tr>
<td>Project</td>
<td>PL/I</td>
<td>PL/I</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SQLCA formatting</td>
<td>Assembler</td>
<td>Assembler</td>
<td>Assembler</td>
<td>Assembler</td>
<td></td>
</tr>
<tr>
<td>routines</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Stored procedures</td>
<td>COBOL</td>
<td>PL/I</td>
<td>C</td>
<td>SQL</td>
<td></td>
</tr>
<tr>
<td>User-defined</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>C</td>
</tr>
<tr>
<td>functions</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>C++</td>
</tr>
<tr>
<td>LOBs</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>C</td>
</tr>
</tbody>
</table>

Notes:
1. Assembler subroutine DSN8CA.

Sample applications in TSO

A set of Db2 sample applications run in the TSO environment.

Table 182. Sample Db2 applications for TSO

<table>
<thead>
<tr>
<th>Application</th>
<th>Program name</th>
<th>Preparation JCL member name</th>
<th>Attachment facility</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Phone</td>
<td>DSN8BC3</td>
<td>DSNTEJ2C</td>
<td>DSNELI</td>
<td>This COBOL batch program lists employee telephone numbers and updates them if requested.</td>
</tr>
<tr>
<td>Phone</td>
<td>DSN8BD3</td>
<td>DSNTEJ2D</td>
<td>DSNELI</td>
<td>This C batch program lists employee telephone numbers and updates them if requested.</td>
</tr>
<tr>
<td>Phone</td>
<td>DSN8BE3</td>
<td>DSNTEJ2E</td>
<td>DSNELI</td>
<td>This C++ batch program lists employee telephone numbers and updates them if requested.</td>
</tr>
<tr>
<td>Phone</td>
<td>DSN8BP3</td>
<td>DSNTEJ2P</td>
<td>DSNELI</td>
<td>This PL/I batch program lists employee telephone numbers and updates them if requested.</td>
</tr>
<tr>
<td>Phone</td>
<td>DSN8BF3</td>
<td>DSNTEJ2F</td>
<td>DSNELI</td>
<td>This FORTRAN program lists employee telephone numbers and updates them if requested.</td>
</tr>
<tr>
<td>Organization</td>
<td>DSN8HC3</td>
<td>DSNTEJ3C or DSNTEJ6</td>
<td>DSNALI</td>
<td>This COBOL ISPF program displays and updates information about a local department. It can also display and update information about an employee at a local or remote location.</td>
</tr>
<tr>
<td>Phone</td>
<td>DSN8SC3</td>
<td>DSNTEJ3C</td>
<td>DSNALI</td>
<td>This COBOL ISPF program lists employee telephone numbers and updates them if requested.</td>
</tr>
<tr>
<td>Application</td>
<td>Program name</td>
<td>Preparation JCL member name</td>
<td>Attachment facility</td>
<td>Description</td>
</tr>
<tr>
<td>---------------------</td>
<td>--------------</td>
<td>-------------------------------</td>
<td>---------------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Phone</td>
<td>DSN8SP3</td>
<td>DSNTEJ3P</td>
<td>DSNALI</td>
<td>This PL/I ISPF program lists employee telephone numbers and updates them if requested.</td>
</tr>
<tr>
<td>UNLOAD</td>
<td>DSNTIAUL</td>
<td>DSNTEJ2A</td>
<td>DSNELI</td>
<td>This assembler language program unloads the data from a table or view and to produce LOAD utility control statements for the data.</td>
</tr>
<tr>
<td>Dynamic SQL</td>
<td>DSNTIAD</td>
<td>DSNTIJTM</td>
<td>DSNELI</td>
<td>This assembler language program dynamically executes non-SELECT statements read in from SYSIN; that is, it uses dynamic SQL to execute non-SELECT SQL statements.</td>
</tr>
<tr>
<td>Dynamic SQL</td>
<td>DSNTEP2</td>
<td>DSNTEJ1P or DSNTEJ1L</td>
<td>DSNELI</td>
<td>This PL/I program dynamically executes SQL statements read in from SYSIN. Unlike DSNTIAD, this application can also execute SELECT statements.</td>
</tr>
<tr>
<td>Stored procedures¹</td>
<td>DSN8EP1</td>
<td>DSNTEJ6P</td>
<td>DSNELI</td>
<td>The jobs DSNTEJ6P and DSNTEJ6S prepare a PL/I version of the application. This sample executes Db2 commands using the instrumentation facility interface (IFI).</td>
</tr>
<tr>
<td>Stored procedure¹</td>
<td>DSN8EP2</td>
<td>DSNTEJ6S</td>
<td>DSNRLI</td>
<td></td>
</tr>
<tr>
<td>Stored procedures¹</td>
<td>DSN8EPU</td>
<td>DSNTEJ6U</td>
<td>DSNELI</td>
<td>The sample that is prepared by job DSNTEJ6U invokes the utilities stored procedure.</td>
</tr>
<tr>
<td>Stored procedures¹</td>
<td>DSN8ED1</td>
<td>DSNTEJ6D</td>
<td>DSNELI</td>
<td>The jobs DSNTEJ6D and DSNTEJ6T prepare a C version of the application. The C stored procedure uses result sets to return commands to the client. This sample executes Db2 commands using the instrumentation facility interface (IFI).</td>
</tr>
<tr>
<td>Stored procedures¹</td>
<td>DSN8ED2</td>
<td>DSNTEJ6T</td>
<td>DSNRLI</td>
<td></td>
</tr>
<tr>
<td>Stored procedures¹</td>
<td>DSN8EC1</td>
<td>DSNTEJ61</td>
<td>DSNRLI</td>
<td>The sample that is prepared by jobs DSNTEJ61 and DSNTEJ62 demonstrates a stored procedure that accesses IMS databases through the ODBA interface.</td>
</tr>
<tr>
<td>Stored procedures¹</td>
<td>DSN8EC2</td>
<td>DSNTEJ62</td>
<td>DSNELI</td>
<td></td>
</tr>
<tr>
<td>Stored procedures¹</td>
<td>DSN8ES1</td>
<td>DSNTEJ63</td>
<td>DSNRLI</td>
<td>The sample that is prepared by jobs DSNTEJ63 and DSNTEJ64 demonstrates how to prepare an SQL procedure using JCL.</td>
</tr>
<tr>
<td>Stored procedures¹</td>
<td>DSN8ED3</td>
<td>DSNTEJ64</td>
<td>DSNELI</td>
<td></td>
</tr>
<tr>
<td>Stored procedures¹</td>
<td>DSN8ES2</td>
<td>DSNTEJ65</td>
<td>DSNRLI</td>
<td>The sample that is prepared by job DSNTEJ65 demonstrates how to prepare an SQL procedure using the SQL procedure processor.</td>
</tr>
<tr>
<td>Stored procedures¹</td>
<td>DSN8ED6</td>
<td>DSNTEJ6W</td>
<td>DSNELI</td>
<td>The sample that is prepared by job DSNTEJ6W demonstrates how to prepare and run a client program that calls a Db2–supplied stored procedure to refresh a WLM environment.</td>
</tr>
</tbody>
</table>
### Table 182. Sample Db2 applications for TSO (continued)

<table>
<thead>
<tr>
<th>Application</th>
<th>Program name</th>
<th>Preparation JCL member name</th>
<th>Attachment facility</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stored procedures</td>
<td>DSN8ED7</td>
<td>DSNTEJ6Z</td>
<td>DSNELI</td>
<td>The sample that is prepared by job DSNTEJ6Z demonstrates how to prepare and run a client program that calls a Db2–supplied stored procedure to display the current settings of system parameters.</td>
</tr>
<tr>
<td>Stored procedures</td>
<td>DSN8ED9</td>
<td>DSNTEJ66</td>
<td>DSNELI</td>
<td>The sample that is prepared by job DSNTEJ66 demonstrates how to prepare and run a client program that calls a native SQL procedure, manages versions of that procedure, and optionally, deploys that procedure to a remote server. DSN8ES3 is the sample native SQL procedure and DSN8ED9 is the sample C language caller of DSN8ES3.</td>
</tr>
<tr>
<td>Stored procedures</td>
<td>DSN8ES3</td>
<td>DSNTEJ66</td>
<td>not applicable</td>
<td>The sample that is prepared by job DSNTEJ66 demonstrates how to prepare and run a client program that calls a native SQL procedure, manages versions of that procedure, and optionally, deploys that procedure to a remote server. DSN8ES3 is the sample native SQL procedure and DSN8ED9 is the sample C language caller of DSN8ES3.</td>
</tr>
<tr>
<td>User-defined functions</td>
<td>DSN8DUAD</td>
<td>DSNTEJ2U</td>
<td>DSNRLI</td>
<td>These C applications consist of a set of user-defined scalar functions that can be invoked through SPUFI or DSNTEP2.</td>
</tr>
<tr>
<td>User-defined functions</td>
<td>DSN8DUAT</td>
<td>DSNTEJ2U</td>
<td>DSNRLI</td>
<td>These C applications consist of a set of user-defined scalar functions that can be invoked through SPUFI or DSNTEP2.</td>
</tr>
<tr>
<td>User-defined functions</td>
<td>DSN8DUCD</td>
<td>DSNTEJ2U</td>
<td>DSNRLI</td>
<td>These C applications consist of a set of user-defined scalar functions that can be invoked through SPUFI or DSNTEP2.</td>
</tr>
<tr>
<td>User-defined functions</td>
<td>DSN8DUCT</td>
<td>DSNTEJ2U</td>
<td>DSNRLI</td>
<td>These C applications consist of a set of user-defined scalar functions that can be invoked through SPUFI or DSNTEP2.</td>
</tr>
<tr>
<td>User-defined functions</td>
<td>DSN8DUCY</td>
<td>DSNTEJ2U</td>
<td>DSNRLI</td>
<td>These C applications consist of a set of user-defined scalar functions that can be invoked through SPUFI or DSNTEP2.</td>
</tr>
<tr>
<td>User-defined functions</td>
<td>DSN8DUTI</td>
<td>DSNTEJ2U</td>
<td>DSNRLI</td>
<td>These C applications consist of a set of user-defined scalar functions that can be invoked through SPUFI or DSNTEP2.</td>
</tr>
<tr>
<td>User-defined functions</td>
<td>DSN8DUWC</td>
<td>DSNTEJ2U</td>
<td>DSNRLI</td>
<td>The user-defined table function DSN8DUWF can be invoked by the C client program DSN8DUWC.</td>
</tr>
<tr>
<td>User-defined functions</td>
<td>DSN8DUWF</td>
<td>DSNTEJ2U</td>
<td>DSNRLI</td>
<td>These C++ applications consist of a set of user-defined scalar functions that can be invoked through SPUFI or DSNTEP2.</td>
</tr>
<tr>
<td>User-defined functions</td>
<td>DSN8EUDN</td>
<td>DSNTEJ2U</td>
<td>DSNRLI</td>
<td>These C++ applications consist of a set of user-defined scalar functions that can be invoked through SPUFI or DSNTEP2.</td>
</tr>
<tr>
<td>User-defined functions</td>
<td>DSN8EUMN</td>
<td>DSNTEJ2U</td>
<td>DSNRLI</td>
<td>These C++ applications consist of a set of user-defined scalar functions that can be invoked through SPUFI or DSNTEP2.</td>
</tr>
<tr>
<td>User-defined functions</td>
<td>DSN8HDFS</td>
<td>DSNTEJBI</td>
<td>DSNRLI</td>
<td>These C++ applications consist of a set of user-defined scalar functions that can be invoked through SPUFI or DSNTEP2.</td>
</tr>
</tbody>
</table>
### Table 182. Sample Db2 applications for TSO (continued)

<table>
<thead>
<tr>
<th>Application</th>
<th>Program name</th>
<th>Preparation JCL member name</th>
<th>Attachment facility</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>User-defined functions</td>
<td>DSN8JAQL</td>
<td>DSNEJBI</td>
<td>DSNRLI</td>
<td>The user-defined scalar function JAQL_SUBMIT, which is prepared by DSNEJBI, invokes an IBM InfoSphere BigInsights Jaql query. This user-defined function can be invoked through SPUFI or DSNTEP2.</td>
</tr>
<tr>
<td>LOBs</td>
<td>DSN8DLPL</td>
<td>DSNEJ71</td>
<td>DSNELI</td>
<td>These applications demonstrate how to populate a LOB column that is greater than 32 KB, manipulate the data using the POSSTR and SUBSTR built-in functions, and display the data in ISPF using GDDM.</td>
</tr>
<tr>
<td>LOBs</td>
<td>DSN8DLC7</td>
<td>DSNEJ71</td>
<td>DSNELI</td>
<td></td>
</tr>
<tr>
<td>LOBs</td>
<td>DSN8DLRV</td>
<td>DSNEJ73</td>
<td>DSNELI</td>
<td></td>
</tr>
<tr>
<td>LOBs</td>
<td>DSN8DLPV</td>
<td>DSNEJ75</td>
<td>DSNELI</td>
<td></td>
</tr>
</tbody>
</table>

#### Note:

1. All of the stored procedure applications consist of a calling program, a stored procedure program, or both.

#### Related reference

Data sets that the precompiler uses
When you invoke the precompiler you need to provide data sets that contain input for the precompiler, such as the host programming statements and SQL statements. You also need to provide data sets where the precompiler can store its output, such as the modified source code and diagnostics messages.

**DSN8BC3**

THIS MODULE LISTS EMPLOYEE PHONE NUMBERS AND UPDATES THEM IF DESIRED.

```cobol
IDENTIFICATION DIVISION.
PROGRAM-ID. DSN8BC3.

****** DSN8BC3 - DB2 SAMPLE PHONE APPLICATION - COBOL - BATCH ***
* * MODULE NAME = DSN8BC3
* * DESCRIPTIVE NAME = DB2 SAMPLE APPLICATION
* * PHONE APPLICATION
* * BATCH
* * COBOL
* * LICENSED MATERIALS - PROPERTY OF IBM
* * 5605-DB2
* * (C) COPYRIGHT 1982, 2010 IBM CORP. ALL RIGHTS RESERVED.
* * STATUS = VERSION 10
* * FUNCTION = THIS MODULE LISTS EMPLOYEE PHONE NUMBERS AND UPDATES THEM IF DESIRED.
* * NOTES = NONE
* * MODULE TYPE = COBOL PROGRAM
* * PROCESSOR = DB2 PRECOMPILER, VS COBOL
* * MODULE SIZE = SEE LINK EDIT
* * ATTRIBUTES = NOT REENTRANT OR REUSABLE
* * ENTRY POINT = DSN8BC3
* * PURPOSE = SEE FUNCTION
* * LINKAGE = INVOKED FROM DSN RUN
* * INPUT =
```

**1030** Db2 12 for z/OS: Application Programming and SQL Guide
* SYMBOLIC LABEL/NAME = CARDIN  *
* DESCRIPTION = INPUT REQUEST FILE  *
* *
* SYMBOLIC LABEL/NAME = VPHONE  *
* DESCRIPTION = VIEW OF TELEPHONE  *
* *
* OUTPUT = *
* *
* SYMBOLIC LABEL/NAME = REPORT  *
* DESCRIPTION = REPORT OF EMPLOYEE PHONE NUMBERS  *
* *
* SYMBOLIC LABEL/NAME = VEMPLP  *
* DESCRIPTION = VIEW OF EMPLOYEE INFORMATION  *
* *
* EXIT-NORMAL = RETURN CODE 0 NORMAL COMPLETION *
* *
* EXIT-ERROR = *
* *
* RETURN CODE = NONE *
* *
* ABEND CODES = NONE *
* *
* ERROR-MESSAGES = *
* DSN8004I - EMPLOYEE SUCCESSFULLY UPDATED *
* DSN8007E - EMPLOYEE DOES NOT EXIST, UPDATE NOT DONE *
* DSN8008I - NO EMPLOYEE FOUND IN TABLE *
* DSN8053I - ROLLBACK SUCCESSFUL, ALL UPDATES REMOVED*
* DSN8060E - SQL ERROR, RETURN CODE IS: *
* DSN8061E - ROLLBACK FAILED, RETURN CODE IS: *
* DSN8069E - INVALID REQUEST, SHOULD BE 'L' OR 'U' *
* DSN8075E - MESSAGE FORMAT ROUTINE ERROR, *
* RETURN CODE IS: *
* *
* EXTERNAL REFERENCES = *
* ROUTINES/SERVICES = *
* DSN8MCG - ERROR MESSAGE ROUTINE *
* *
* DATA AREAS = NONE *
* *
* CONTROL-BLOCKS = *
* SQLCA - SQL COMMUNICATION AREA *
* *
* TABLES = NONE *
* *
* CHANGE-ACTIVITY = NONE *
* *
* *PSEUDOCODE* *
* *
* PROCEDURE *
* GET FIRST INPUT *
* DO WHILE MORE INPUT *
* CREATE REPORT HEADING *
* *
* CASE (ACTION) *
* *
* SUBCASE ('L') *
*   IF LASTNAME IS '*' THEN *
*     LIST ALL EMPLOYEES *
*   ELSE *
*     IF LASTNAME CONTAINS '%' THEN *
*       LIST EMPLOYEES GENERIC *
*     ELSE *
*       LIST EMPLOYEES SPECIFIC *
*   ENDSUB *
* *
* SUBCASE ('U') *
*   UPDATE PHONENUMBER FOR EMPLOYEE *
*   WRITE CONFIRMATION MESSAGE *
*   OTHERWISE *
*   INVALID REQUEST *
*   ENDSUB *
* *
* ENDCASE *
* GET NEXT INPUT *
* END
* IF SQL ERROR OCCURS THEN
*     DO
*     FORMAT ERROR MESSAGE
*     ROLLBACK
*     END
* END.
*
/ ENVIRONMENT DIVISION.
*----------------------
CONFIGURATION SECTION.
SPECIAL-NAMES. C01 IS TO-TOP-OF-PAGE.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
    SELECT CARDIN
    ASSIGN TO DA-S-CARDIN.
    SELECT REPOUT
    ASSIGN TO UT-S-REPORT.

DATA DIVISION.
*-----------------
FILE SECTION.
FD CARDIN
    RECORD CONTAINS 80 CHARACTERS
    BLOCK CONTAINS 0 RECORDS
    LABEL RECORDS ARE OMITTED.
    01 CARDREC PIC X(80).
FD REPOUT
    RECORD CONTAINS 120 CHARACTERS
    LABEL RECORDS ARE OMITTED
    DATA RECORD IS REPREC.
    01 REPREC PIC X(120).
/
WORKING-STORAGE SECTION.
****************************************************
* STRUCTURE FOR INPUT
****************************************************
01 IOAREA.
    02 ACTION PIC X(01).
    02 LNAME PIC X(15).
    02 FNAME PIC X(12).
    02 ENO PIC X(06).
    02 NEWNO PIC X(04).
    02 FILLER PIC X(42).
****************************************************
* REPORT HEADER STRUCTURE
****************************************************
01 REPHDR1.
    02 FILLER PIC X(29) VALUE '-----------------------------'.
    02 FILLER PIC X(21) VALUE ' TELEPHONE DIRECTORY '.
    02 FILLER PIC X(29) VALUE '-----------------------------'.
01 REPHDR2.
    02 FILLER PIC X(09) VALUE 'LAST NAME'.
    02 FILLER PIC X(07) VALUE 'FIRST NAME'.
    02 FILLER PIC X(03) VALUE SPACES.
    02 FILLER PIC X(08) VALUE 'INITIAL'.
    02 FILLER PIC X(07) VALUE 'PHONE'.
    02 FILLER PIC X(09) VALUE 'EMPLOYEE'.
    02 FILLER PIC X(03) VALUE 'WORK'.
    02 FILLER PIC X(04) VALUE 'WORK'.
01 REPHDR3.
    02 FILLER PIC X(37) VALUE SPACES.
    02 FILLER PIC X(07) VALUE 'NUMBER'.
    02 FILLER PIC X(09) VALUE 'NUMBER'.
    02 FILLER PIC X(05) VALUE 'DEPT'.
    02 FILLER PIC X(05) VALUE 'DEPT'.
    02 FILLER PIC X(04) VALUE 'NAME'.
****************************************************
* REPORT STRUCTURE
****************************************************
01 REPDATA.
02 RLNAME    PIC X(15).
02 FILLER    PIC X(01) VALUE SPACES.
02 RFNAME    PIC X(12).
02 FILLER    PIC X(04) VALUE SPACES.
02 RMIDINIT  PIC X(01).
02 FILLER    PIC X(04) VALUE SPACES.
02 RPHONE    PIC X(04).
02 FILLER    PIC X(03) VALUE SPACES.
02 REMPNO    PIC X(06).
02 FILLER    PIC X(03) VALUE SPACES.
02 RDEPTNO   PIC X(03).
02 FILLER    PIC X(02) VALUE SPACES.
02 RDEPTNAME PIC X(36).

********************************************************************
* WORKAREAS                                                  *
********************************************************************
 01 LNAME-WORK.
   49 LNAME-WORKL     PIC S9(4)  COMP.
   49 LNAME-WORKC     PIC X(15).
 01 FNAME-WORK.
   49 FNAME-WORKL     PIC S9(4)  COMP.
   49 FNAME-WORKC     PIC X(12).
 77 INPUT-SWITCH    PIC X          VALUE 'Y'.
 88 NOMORE-INPUT     VALUE 'N'.
 77 NOT-FOUND        PIC S9(9) COMP VALUE +100.

********************************************************************
* VARIABLES FOR ERROR-HANDLING                                *
********************************************************************
 01 ERROR-MESSAGE.
  02 ERROR-LEN   PIC S9(4)  COMP VALUE +960.
  02 ERROR-TEXT  PIC X(120) OCCURS 10 TIMES
                 INDEXED BY ERROR-INDEX.
  77 ERROR-TEXT-LEN PIC S9(9) COMP VALUE +120.

/********************************************************************************
* SQL INCLUDE FOR SQLCA                                              *
********************************************************************************
EXEC SQL INCLUDE SQLCA  END-EXEC.

********************************************************************************
* SQL DECLARATION FOR VIEW VPHONE                                  *
********************************************************************************
EXEC SQL DECLARE VPHONE TABLE
  (LASTNAME     VARCHAR(15)  NOT NULL,
   FIRSTNAME    VARCHAR(12)  NOT NULL,
   MIDDLEINITIAL CHAR(01)  NOT NULL,
   PHONENUMBER  CHAR(04)          ,
   EMPLOYEENUMBER CHAR(06)  NOT NULL,
   DEPTNUMBER   CHAR(03)  NOT NULL,
   DEPTNAME     VARCHAR(36)  NOT NULL)
END-EXEC.

********************************************************************************
* STRUCTURE FOR PPHONE RECORD                                     *
********************************************************************************
 01 PPHONE.
   02 LASTNAME.
      49 LASTNAMEL      PIC S9(4)  COMP.
      49 LASTNAMEC      PIC X(15)  VALUE SPACES.
   02 FIRSTNAME.
      49 FIRSTNAMEL     PIC S9(4)  COMP.
      49 FIRSTNAMEC     PIC X(12)  VALUE SPACES.
   02 MIDDLEINITIAL  PIC X(01).
   02 PHONENUMBER    PIC X(04).
   02 EMPLOYEENUMBER PIC X(06).
   02 DEPTNUMBER     PIC X(03).
   02 DEPTNAME.
      49 DEPTNAMEL     PIC S9(4)  COMP.
      49 DEPTNAMEC     PIC X(36)  VALUE SPACES.
      77 PERCENT-COUNTER PIC S9(4)  COMP.

********************************************************************************
* SQL DECLARATION FOR VIEW VEMPLP                                  *
********************************************************************************
EXEC SQL DECLARE VEMPLP TABLE
  (EMPLOYEENUMBER   CHAR(06)  NOT NULL,
   PHONENUMBER      CHAR(04)          )
END-EXEC.
**SQL CURSORS**

*** CURSOR LISTS ALL EMPLOYEE NAMES

EXEC SQL DECLARE TELE1 CURSOR FOR
SELECT *
FROM VPHONE
END-EXEC.

*** CURSOR LISTS ALL EMPLOYEE NAMES WITH A PATTERN (%) OR (_)

*** FOR LAST NAME

EXEC SQL DECLARE TELE2 CURSOR FOR
SELECT *
FROM VPHONE
WHERE LASTNAME LIKE :LNAME-WORK
AND FIRSTNAME LIKE :FNAME-WORK
END-EXEC.

*** CURSOR LISTS ALL EMPLOYEES WITH A SPECIFIC

*** LAST NAME

EXEC SQL DECLARE TELE3 CURSOR FOR
SELECT *
FROM VPHONE
WHERE LASTNAME = :LNAME
AND FIRSTNAME LIKE :FNAME-WORK
END-EXEC.

/
**PRINT HEADING**

**SELECT ACTION**

**INVALID REQUEST**

**PRINT ERROR MESSAGE**

**UNSTRING LAST NAME**

**UNSTRING FIRST NAME**

**COUNT %'S**

**LIST GENERIC NAME(S)**

**LIST SPECIFIC NAME(S)**

**LIST GENERIC NAME(S)**

**LIST SPECIFIC NAME(S)**

**LIST GENERIC NAME(S)**

**LIST SPECIFIC NAME(S)**
**LIST ALL EMPLOYEES**

PERFORM PRINT-AND-GET1
UNTIL SQLCODE IS NOT EQUAL TO ZERO.

**CLOSE CURSOR**

PRINT-AND-GET1.
PERFORM PRINT-A-LINE.
EXEC SQL FETCH TELE1 INTO :PPHONE END-EXEC.
/
******************************************************************************
** LIST GENERIC EMPLOYEES **
******************************************************************************
LIST GENERIC.

**OPEN CURSOR**

EXEC SQL OPEN  TELE2 END-EXEC.

**GET EMPLOYEES**

IF SQLCODE = NOT-FOUND
* **NO EMPLOYEE FOUND**
**PRINT ERROR MESSAGE**
MOVE '008I' TO MSGCODE
CALL 'DSN8MCG' USING MAJOR MSGCODE OUTMSG
MOVE OUTMSG TO OUTMSG2
WRITE REPREC FROM MSG-REC2
AFTER ADVANCING 2 LINES
ELSE
* **LIST GENERIC EMPLOYEE(S)**
PERFORM PRINT-AND-GET2
UNTIL SQLCODE IS NOT EQUAL TO ZERO.

**CLOSE CURSOR**

EXEC SQL OPEN  TELE3 END-EXEC.

**GET EMPLOYEES**

IF SQLCODE = NOT-FOUND
* **NO EMPLOYEE FOUND**
**PRINT ERROR MESSAGE**
MOVE '008I' TO MSGCODE
CALL 'DSN8MCG' USING MAJOR MSGCODE OUTMSG
MOVE OUTMSG TO OUTMSG2
WRITE REPREC FROM MSG-REC2
AFTER ADVANCING 2 LINES
ELSE
* **LIST SPECIFIC EMPLOYEE(S)**
PERFORM PRINT-AND-GET3
UNTIL SQLCODE IS NOT EQUAL TO ZERO.

**CLOSE CURSOR**

EXEC SQL OPEN  TELE3 END-EXEC.

**GET INFORMATION**

MOVE LASTNAMEC TO RLNAME.
MOVE FIRSTNAMEC TO RFNAME.
MOVE MIDDLEINITIAL TO RMIDINIT.
MOVE PHONENUMBER OF PPHONE TO RPHONE.
MOVE EMPLOYEENUMBER OF PPHONE TO REMPNO.
MOVE DEPTNUMBER TO RDEPTNO.

**PRINT A LINE OF INFORMATION FROM DIRECTORY**

PERFORM PRINT-A-LINE.
EXEC SQL FETCH TELE3 INTO :PPHONE END-EXEC.
/
******************************************************************************
** PRINT A LINE OF INFORMATION FROM DIRECTORY **
******************************************************************************
PRINT-A-LINE.
* **GET INFORMATION**

MOVE LASTNAMEC TO RLNAME.
MOVE FIRSTNAMEC TO RFNAME.
MOVE MIDDLEINITIAL TO RMIDINIT.
MOVE PHONENUMBER OF PPHONE TO RPHONE.
MOVE EMPLOYEENUMBER OF PPHONE TO REMPNO.
MOVE DEPTNUMBER TO RDEPTNO.
MOVE DEPTNAMEC TO RDEPTNAME.
* WRITE REPREC FROM REPDATA
  AFTER ADVANCING 2 LINES.

MOVE SPACES TO LASTNAMEC
FIRSTNAMEC
DEPTNAMEC.
/

*******************************************************************************
* UPDATES PHONE NUMBERS FOR EMPLOYEES                                    *
*******************************************************************************

TELEPHONE-UPDATE.
EXEC SQL UPDATE VEMPLP
SET PHONENUMBER = :NEWNO
WHERE EMPLOYEENUMBER = :ENO END-EXEC.

* **EMPLOYEE FOUND  
**UPDATE SUCCESSFUL  
**MESSAGE  

MOVE '004I' TO MSGCODE
ELSE
  * **NO EMPLOYEE FOUND  
**UPDATE FAILED  
**PRINT ERROR MESSAGE  

MOVE '007E' TO MSGCODE.
CALL 'DSN8MCG' USING MAJOR MSGCODE OUTMSG.
MOVE OUTMSG TO OUTMSG2.
WRITE REPREC FROM MSG-REC2
AFTER ADVANCING 2 LINES.
/

*******************************************************************************
* SQL ERROR OCCURRED - GET ERROR MESSAGE                                   *
*******************************************************************************

DBERROR.
* **SQL ERROR  
**PRINT ERROR MESSAGE  

MOVE '060E' TO MSGCODE.
CALL 'DSN8MCG' USING MAJOR MSGCODE OUTMSG.
MOVE OUTMSG TO OUTMSG1 OF MSG-REC1.
MOVE SQLCODE TO RETCODE OF MSG-REC1.
WRITE REPREC FROM MSG-REC1
AFTER ADVANCING 2 LINES.
CALL 'DSN8MCG' USING SQLCA ERROR-MESSAGE ERROR-TEXT-LEN.
IF RETURN-CODE = ZERO
  PERFORM ERROR-PRINT VARYING ERROR-INDEX
  FROM 1 BY 1 UNTIL ERROR-INDEX GREATER THAN 10
ELSE
  * **MESSAGE FORMAT  
**ROUTE RERINE ERROR  
**PRINT ERROR MESSAGE  

MOVE '075E' TO MSGCODE.
CALL 'DSN8MCG' USING MAJOR MSGCODE OUTMSG.
MOVE OUTMSG TO OUTMSG1 OF MSG-REC1.
MOVE RETURN-CODE TO RETCODE OF MSG-REC1.
WRITE REPREC FROM MSG-REC1
AFTER ADVANCING 2 LINES.

*******************************************************************************
* SQL RETURN CODE HANDLING WHEN PROCESSING CANNOT PROCEED                   *
*******************************************************************************

EXEC SQL WHENEVER SQLERROR CONTINUE END-EXEC.
EXEC SQL WHENEVER SQLWARNING CONTINUE END-EXEC.
EXEC SQL WHENEVER NOT FOUND CONTINUE END-EXEC.

* EXEC SQL ROLLBACK END-EXEC.
  IF SQLCODE = ZERO
    * **ROLLBACK SUCCESSFUL  
**PRINT CONFIRMATION  
**MESSAGE  

MOVE '053I' TO MSGCODE
ELSE
  * **ROLLBACK FAILED  
**PRINT ERROR MESSAGE  

MOVE '061E' TO MSGCODE.
CALL 'DSN8MCG' USING MAJOR MSGCODE OUTMSG.
MOVE OUTMSG TO OUTMSG1 OF MSG-REC1.
MOVE SQLCODE TO RETCODE OF MSG-REC1.
WRITE REPREC FROM MSG-REC1
  AFTER ADVANCING 2 LINES.
GO TO PROG-END.

**********************************************************************/
/*                                                                    */
/*  Module name = DSN8BD3                                             */
/*                                                                    */
/*  Descriptive name = DB2  SAMPLE APPLICATION                        */
/*                     PHONE APPLICATION                              */
/*                     BATCH                                          */
/*                     C LANGUAGE                                     */
/*                                                                    */
/*     LICENSED MATERIALS - PROPERTY OF IBM                           */
/*     5695-DB2                                                       */
/*     (C) COPYRIGHT 1982, 1995 IBM CORP.  ALL RIGHTS RESERVED.       */
/*                                                                    */
/*     STATUS = VERSION 4                                             */
/*                                                                    */
/*     Function = This module lists employee phone numbers and        */
/*             optionally updates them.                              */
/*                                                                    */
/*     Notes = none                                                   */
/*                                                                    */
/*                                                                    */
/*  Module type    = C program                                        */
/*     Processor   = DB2  precompiler, C compiler                     */
/*     Module size = see link edit                                   */
/*     Attributes  = not reentrant or reusable                       */
/*                                                                    */
/*  Entry point = DSN8BD3                                             */
/*     Purpose  = see function                                       */
/*     Linkage  = invoked from DSN command processor subcommand RUN   */
/*     Input    =                                                     */
/*                                                                    */
/*                symbolic label/name = CARDIN                        */
/*                description = INPUT REQUEST FILE                    */
/*                                                                    */
/*                symbolic label/name = VPHONE                        */
/*                description = VIEW OF TELEPHONE TABLE: PHONE        */
/*                                                                    */
/*  Output   =                                                     */
/*                                                                    */
/*                symbolic label/name = REPORT                        */
/*                description = PRINTED REPORT AND RESULTS            */
/*                                                                    */
/*                symbolic label/name = VEMPLP                        */
/*                description = VIEW OF EMPLOYEE INFORMATION          */
/*                                                                    */
/*                                                                    */
/*  Exit-normal = return code 0 normal completion                    */
/*                                                                    */
/*  Exit-error =                                                     */
/*                                                                    */
/*     Return code    = none                                         */
/*     Abend codes    = none                                         */
/*     Error-messages =                                               */
/*           DSN8000I - REQUEST IS:  ...                            */
/*           DSN8004I - EMPLOYEE SUCCESSFULLY UPDATED                 */

Related reference
“Sample applications in TSO” on page 1027
A set of Db2 sample applications run in the TSO environment.

DSN8BD3
This module lists employee phone numbers and optionally updates them.
/ * DSN8007E - EMPLOYEE DOES NOT EXIST, UPDATE NOT DONE */
/ * DSN8008I - NO EMPLOYEE FOUND IN TABLE */
/ * DSN8053I - ROLLBACK SUCCESSFUL, ALL UPDATES REMOVED */
/ * DSN8060E - SQL ERROR, RETURN CODE IS: */
/ * DSN8061E - ROLLBACK FAILED, RETURN CODE IS: */
/ * DSN8066E - INVALID REQUEST, SHOULD BE 'L' OR 'U' */
/ * DSN8075E - MESSAGE FORMAT ROUTINE ERROR, */
/ * RETURN CODE IS: */
/ *
/ * External references = */
/ * Routines/services = */
/ * DSNTIAR - translate sqlca into messages */
/ * Data-areas = none */
/ * Control-blocks = */
/ * SQLCA - sql communication area */
/ * Tables = none */
/ * Change-activity = */
/ * 10/03/94 Updated cardin statement to prevent looping. KEW1351 @51* 
/ * PN61293 @51* 
/ * +Pseudocode* 
/ * 
/ * main: */
/ * do while more input */
/ * get input */
/ * display request */
/ * process request */
/ * end */
/ * Do_req: */
/ * case (action) */
/ * subcase ('L') */
/ * create report heading */
/ * if lastname is '*' then */
/ * list all employees */
/ * else */
/ * if lastname contains '%' then */
/ * list employees generic */
/ * else */
/ * list employees specific */
/ * endsub */
/ * subcase ('U') */
/ * update phonenumber for employee */
/ * write confirmation message */
/ * otherwise */
/ * invalid request */
/ * endsub */
/ * endcase */
/ * Prt_row: */
/ * print a row of the report */
/ * Sql_err: */
/ * if sql error occurs then */
/ * rollback */
/ */
/ ******************************
/#include <stdio.h>
#include <stdlib.h>
#include <string.h>
/******************************
/* General declarations */
/******************************
#define NOTFOUND 100
/*****************************/
/* Input / Output files */
/*****************************/
FILE *cardin;  /* Input control cards */
FILE *report;                          /* Output phone report         */

/***********************************************************************/
/* Input record structure                                             */
/***********************************************************************/
EXEC SQL BEGIN DECLARE SECTION;
struct {
    char action[2];                      /* L for list or U for update */
    char lname[16];                      /* last name or pattern- L mode */
    char fname[13];                      /* first name or pattern- L mode */
    char eno[7];                         /* employee number - U mode */
    char newno[5];                       /* new phone number - U mode */
} ioarea;
char trail[43];                        /* unused portion of input rec */
char slname[16];                       /* unmodified last name pattern*/
EXEC SQL END DECLARE SECTION;

/***********************************************************************/
/* Report headings                                                    */
/***********************************************************************/
struct {
    char hdr011[30];
    char hdr012[32];
} hdr0 = {
    " REQUEST LAST NAME",
    "FIRST NAME EMPNO NEW XT.NO"};
#define rpthdr0 hdr0.hdr011

struct {
    char hdr111[29];
    char hdr112[21];
    char hdr113[30];
} hdr1 = {
    "----------------------------",
    " TELEPHONE DIRECTORY",
    "----------------------------"};
#define rpthdr1 hdr1.hdr111

struct {
    char hdr211[10];
    char hdr212[11];
    char hdr213[8];
    char hdr214[6];
    char hdr215[9];
    char hdr216[5];
    char hdr217[5];
    char hdr218[7];
    char hdr219[7];
    char hdr220[5];
    char hdr221[5];
    char hdr222[5];
    char hdr223[5];
    char hdr224[5];
    char hdr225[5];
} hdr2 = {
    "LAST NAME",
    "FIRST NAME",
    "INITIAL",
    "PHONE",
    "EMPLOYEE",
    "WORK",
    "WORK",
    "NUMBER",
    "NUMBER",
    "DEPT",
    "DEPT",
    "NAME"
};
#define rpthdr2 hdr2.hdr211,hdr2.hdr212,hdr2.hdr213,hdr2.hdr214,\
    hdr2.hdr215,hdr2.hdr216,hdr2.hdr217,hdr2.hdr221,\
    hdr2.hdr222,hdr2.hdr223,hdr2.hdr224,hdr2.hdr225

/***********************************************************************/
/* Report formats                                                     */
/***********************************************************************/
static char fmt1[]= "\n %s\n";
static char fmt2[]= " %9s%17s%10s%5s%5s\n%43s%7s%7s%5s%5s\n";
static char fmt3[]= " %-16s%-16s%-5s%-7s%-9s%-5s%-36s\n";
static char fmt4[]= " %1c%15c%12c%6c%4c%43c";
static char fmt5[]= "\n\n %s\n %s\n   --%-7s--%-15s--%-12s--%-5s--%-9s--\n";

/***********************************************************************/
/* Fields sent to message routine                                     */
/***********************************************************************/
*/
EXEC SQL INCLUDE SQLCA;

EXEC SQL DECLARE VPHONE TABLE
(LASTNAME       VARCHAR(15)  NOT NULL,
FIRSTNAME      VARCHAR(12)  NOT NULL,
MIDDLEINITIAL     CHAR( 1)  NOT NULL,
PHONENUMBER       CHAR( 4)          ,
EMPLOYEENUMBER    CHAR( 6)  NOT NULL,
DEPTNUMBER        CHAR( 3)  NOT NULL,
DEPTNAME       VARCHAR(36)  NOT NULL);

EXEC SQL BEGIN DECLARE SECTION;
struct {
    char lastname[16];
    char firstname[13];
    char middleinitial[2];
    char phonenumber[5];
    char employeenumber[7];
    char deptnumber[4];
} pphone;
EXEC SQL END   DECLARE SECTION;

EXEC SQL DECLARE VEMPLP TABLE
(EMPLOYEENUMBER    CHAR( 6)  NOT NULL,
PHONENUMBER       CHAR( 4)          );

EXEC SQL BEGIN DECLARE SECTION;
struct {
    char employeenumber[7];
    char phonenumber[5];
} pemplp;
EXEC SQL END   DECLARE SECTION;

EXEC SQL DECLARE TELE1 CURSOR FOR
SELECT *
FROM VPHONE;

EXEC SQL DECLARE TELE2 CURSOR FOR
SELECT *
FROM VPHONE
WHERE LASTNAME LIKE :lname;

EXEC SQL DECLARE TELE3 CURSOR FOR
SELECT *
FROM VPHONE
WHERE LASTNAME = :slname
    AND FIRSTNAME LIKE :fname;
EXEC SQL WHENEVER SQLERROR GOTO DBERROR;
EXEC SQL WHENEVER SQLWARNING GOTO DBERROR;
EXEC SQL WHENEVER NOT FOUND CONTINUE;

extern main()
{
/* Open the input and output files */
cardin = fopen("DD:CARDIN","r,recfm=FB,lrecl=80,blksize=80"); /*@51*/
report = fopen("DD:REPORT","w");

/* While more input, process */
while (!feof(cardin))
{
/* Read the next request */
if (fscanf(cardin, fmt4,
ioarea.action,
ioarea.lname,
ioarea.fname,
ioarea.eno,
ioarea.newno,
trail) == 6)
{
/* Display the request */
DSN8MDG(module, "000I", outmsg);
fprintf(report, fmt5,
outmsg,
rpthdr0,
ioarea.action,
ioarea.lname,
ioarea.fname,
ioarea.eno,
ioarea.newno);
    Do_req();
}
/* endwhile */
fclose(report);
} /* end main */

/* Process the current request */
Do_req()
{
char *blankloc; /* string translation pointer */
strcpy(slname, ioarea.lname); /* save untranslated last name */
while (blankloc = strpbrk(ioarea.lname, " "))
    *blankloc = '%'; /* translate blanks into % */
while (blankloc = strpbrk(ioarea.fname, " "))
    *blankloc = '%'; /* translate blanks into % */

/* Determine request type */
switch (ioarea.action[0])
{
    case 'L':/* Process LIST request */
        /* Print the report headings */
        fprintf(report, fmt1, rpthdr1);
        fprintf(report, fmt2, rpthdr2);

        /* List all employees */
        if (!strcmp(slname,"* ")){
            EXEC SQL OPEN TELE1;
            EXEC SQL FETCH TELE1 INTO :pphone;
            if (sqlca.sqlcode == NOTFOUND){ /* If no employees */
                DSN8MDG(module, "008I", outmsg);
                fprintf(report, "\%s\n", outmsg);
            } /* endif */
            while (sqlca.sqlcode == 0){
                Prt_row();
                EXEC SQL FETCH TELE1 INTO :pphone;
            } /* endwhile */
            EXEC SQL CLOSE TELE1;
        } /* endif */
    } /* else */
} /* end main */
if (strpbrk(slname, "%")) {
    EXEC SQL OPEN TELE2;
    EXEC SQL FETCH TELE2 INTO :pphone;
    if (sqlca.sqlcode == NOTFOUND) { /* If no employees */
        DSN8MDG(module, "008I", outmsg); /* found, display */
        fprintf(report, " %s
", outmsg); /* error message */
    } else {
        while (sqlca.sqlcode == 0) {
            Prt_row();
            EXEC SQL FETCH TELE2 INTO :pphone;
        } /* endwhile */
    } /* endif */
    EXEC SQL CLOSE TELE2;
}
/* List specific employee */
else {
    EXEC SQL OPEN TELE3;
    EXEC SQL FETCH TELE3 INTO :pphone;
    if (sqlca.sqlcode == NOTFOUND) { /* If no employee */
        DSN8MDG(module, "008I", outmsg); /* found, display */
        fprintf(report, " %s
", outmsg); /* error message */
    } else {
        while (sqlca.sqlcode == 0) {
            Prt_row();
            EXEC SQL FETCH TELE3 INTO :pphone;
        } /* endwhile */
    } /* endif */
} /* endif */
EXEC SQL CLOSE TELE3;
} /* endif */
break; /* end of 'L' request */
/* Update an employee phone number */
case 'U':
    EXEC SQL UPDATE VEMPLP
    SET PHONENUMBER = :ioarea.newno
    WHERE EMPLOYEEUNUMBER = :ioarea.eno;
    if (sqlca.sqlcode == 0) { /* If employee */
        DSN8MDG(module, "004I", outmsg); /* updated, display */
        fprintf(report, " %s
", outmsg); /* confirmation msg */
    } else {
        DSN8MDG(module, "007E", outmsg); /* otherwise, display */
        fprintf(report, " %s
", outmsg); /* error message */
    } /* endif */
    break; /* end of 'U' request */
/* Invalid request type */
default:
    DSN8MDG(module, "068E", outmsg); /* Display error msg */
    fprintf(report, " %s
", outmsg);
} /* endswitch */
return;

DBERROR:
    Sql_err();
} /* end Do_req */

*******************************************************************************/
/* Print a single employee on the report */
*******************************************************************************/
Prt_row()
{
    fprintf(report, fmt3, pphone.lastname,
            pphone.firstname,
            pphone.middleinitial,
            pphone.phonenumber,
            pphone.employeenumber,
            pphone.deptnumber,
            pphone.deptname);
}
*******************************************************************************/
/* SQL error handler */
*******************************************************************************/
#pragma linkage(dsntiar, OS)
Sql_err()
{ 
    define data_len 120
    define data_dim 10
    struct error_struct {
        short int error_len;
        char error_text[data_dim][data_len];
    } error_message = {data_dim * data_len};
}
extern short int dsntiar(struct sqlca *sqlca,
               struct error_struct *msg,
               int *len);

short int rc;
int i;
static int lrecl = data_len;

DSN8MDG(module, "060E", outmsg);
fprintf(report, " %s %i\n", outmsg, sqlca.sqlcode);
rc = dsntiar(&sqlca, &error_message, &lrecl); /* Format the sqlca */
if (rc == 0){ /* Print formatted */
    for (i=0;i<=7;i++) /* sqlca */
        fprintf(report, "%.120s\n", error_message.error_text [i]);
} /* endfor */
} else {
    DSN8MDG(module, "075E", outmsg);
    fprintf(report, " %s %i\n", outmsg, rc);
} /* endif */

/* Attempt to rollback any work already done */
EXEC SQL WHENEVER SQLERROR CONTINUE;
EXEC SQL WHENEVER SQLWARNING CONTINUE;
EXEC SQL WHENEVER NOT FOUND CONTINUE;

EXEC SQL ROLLBACK;
if (sqlca.sqlcode == 0){ /* If rollback */
    DSN8MDG(module, "0631", outmsg); /* completed, display*/
    fprintf(report, " %s\n", outmsg);
} else {
    DSN8MDG(module, "061E", outmsg); /* error message */
    fprintf(report, " %s %i\n", outmsg, sqlca.sqlcode);
} /* endif */
fclose(report);
exit(0);

Related reference
“Sample applications in TSO” on page 1027
A set of Db2 sample applications run in the TSO environment.

DSN8BE3
This module uses the class emp_db2 to list or update employee phone numbers from a Db2 database.

/*********************************************************************************/
/* */
/* Module name = DSN8BD3 */
/* */
/* Descriptive name = DB2 SAMPLE APPLICATION */
/* PHONE APPLICATION */
/* BATCH */
/* C++ LANGUAGE */
/* */
/* LICENSED MATERIALS - PROPERTY OF IBM */
/* 5625-DB2 */
/* (C) COPYRIGHT 1982, 2003 IBM CORP. ALL RIGHTS RESERVED. */
/* */
/* STATUS = VERSION 8 */
/* */
/* Function = This module uses the class emp_db2 to list or update */
/* employee phone numbers from a DB2 database */
/* */
/* Module type = C++ program */
/* Processor = DB2 precompiler, C++ compiler */
/* Module size = see link edit */
/* Attributes = not reentrant or reusable */
/* */
/* Entry point = DSN8BD3 */
/* */
/* Purpose = see function */
/* */
/* Linkage = invoked from DSN command processor subcommand RUN */
/* */
/* Input = symbolic label/name = CARDIN */
/* description = INPUT REQUEST FILE */
/* */
/* Output = symbolic label/name = REPORT */
/* description = PRINTED REPORT AND RESULTS */
/* */
/* Exit-normal = return code 0 normal completion */
/* Exit-error = */
/* Return code = none */
/* Abend codes = none */
/* Error-messages = */
/* DSN8000I - REQUEST IS: ... */
/* DSN8068E - INVALID REQUEST, SHOULD BE 'L' OR 'U' */
/* RETURN CODE IS: */
/* External references = */
/* Routines/services = none */
/* Data-areas = none */
/* Control-blocks = none */
/* Tables = none */
/* Change-activity = */
/* 02/05/96 Katja KFD0024 C++ sample (D9031) */
/* Created based on C sample */

#include <string.h>

#include "DSN8BEH"

/* Input record structure */
struct {
    char action[2]; /* L for list or U for update */
    char lname[16]; /* last name or pattern- L mode */
    char fname[13]; /* first name or pattern-L mode */
    char eno[7]; /* employee number- U mode */
    char newno[5]; /* new phone number- U mode */
} ioarea;

char slname[16]; /* unmodified last name pattern */
class emp_db2 proc1; /* DB2 employee object */

/* Function to process the current request */
void Do_req(FILE *outfile)
{
    char *blankloc; /* string translation pointer */

    /* Report headings */
    struct {
        char hdr111[30]; /* telephone directory */
        char hdr112[22];
        char hdr113[30];
    } hdr1 = {
        "-----------------------------",
        " TELEPHONE DIRECTORY ",
        "-----------------------------"};

    #define rpthdr1 hdr1.hdr111,hdr1.hdr112,hdr1.hdr113

    struct {
        char hdr211[10];
        char hdr212[11];
        char hdr213[ 8];
        char hdr214[ 6];
        char hdr215[ 9];
        char hdr216[ 5];
        char hdr217[ 5];
        char hdr221[ 7];
        char hdr222[ 7];
        char hdr223[ 5];
    } hdr21;

    struct {
        char hdr211[10];
        char hdr212[11];
        char hdr213[ 8];
        char hdr214[ 6];
        char hdr215[ 9];
        char hdr216[ 5];
        char hdr217[ 5];
        char hdr221[ 7];
        char hdr222[ 7];
        char hdr223[ 5];
    } hdr22;

Chapter 12. Sample data and applications supplied with Db2 for z/OS 1045
char hdr224[ 5];
char hdr225[ 5];
} hdr2 = {
    "LAST NAME",
    "FIRST NAME",
    "INITIAL",
    "PHONE",
    "EMPLOYEE",
    "WORK",
    "NUMBER",
    "NUMBER",
    "DEPT",
    "DEPT",
    "NAME"};
#define rpthdr2 hdr2.hdr211,hdr2.hdr212,hdr2.hdr213,hdr2.hdr214,
    hdr2.hdr215,hdr2.hdr216,hdr2.hdr217,hdr2.hdr221,
    hdr2.hdr222,hdr2.hdr223,hdr2.hdr224,hdr2.hdr225

/*****************************************************************
/* Report formats                                                */
/*****************************************************************/
static char fmt1[] = "\n %s\n %s\n %s\n";
static char fmt2[] =
  " %9s%17s%10s%6s%10s%5s%5s\n%43s%8s%7s%5s%5s\n";

/**************************** ***************************************************/
/* Start processing input record */
/**************************** ***************************************************/
strcpy(slname, ioarea.lname);  /* save untranslated last name */
while (blankloc = strpbrk(ioarea.lname, " "))
  *blankloc = '%';             /* translate blanks into %          */
while (blankloc = strpbrk(ioarea.fname, " "))
  *blankloc = '%';             /* translate blanks into %          */

/* Determine request type */
switch (ioarea.action[0])
{
    case 'L': /* Process LIST request */
        /* Print the report headings */
        fprintf(outfile, fmt1, rpthdr1);
        fprintf(outfile, fmt2, rpthdr2);
        if (!strcmp(slname,"*              "))
          /* List all employees */
          proc1.Listall(outfile);
        else
          {
            if (strpbrk(slname, ""))
              /* List generic employees */
              proc1.Listsome(outfile,ioarea.lname);
            else
              /* List specific employee */
              proc1.Listone(outfile,slname,ioarea.fname);
          } /* else - list selected employees */
        break; /* end 'L' request */
    case 'U': /* Update an employee phone number */
        proc1.Empupdate(outfile,ioarea.newno,ioarea.eno);
        break;
    default:
        DSN8MDG(module, "068E", outmsg);        /* Display error msg */
        fprintf(outfile, " %s
", outmsg);
        break; /* end switch */
} /* end Do_req */

/**************************** ***************************************************/
/* Function to read a request from an open file */
/**************************** ***************************************************/
int Read_req(FILE *infile)
{
    static char fmt4[] = " %1c%15c%12c%6c%4c%43c";   /* input format */
    char trail[43];                /* unused part of input record */
    char *newlloc;                 /* addr of newline char in field */
    strcpy(ioarea.action, " ");
    /* Read the request */
    fscanf(infile, fmt4, ioarea.action, trail);
    fprintf(outfile, fmt4, ioarea.action, trail);
    return;
} /* end Read_req */
strcpy(ioarea.lname,"
               ");
strcpy(ioarea.fname,"
               ");
strcpy(ioarea.eno,"
           ");
strcpy(ioarea.newno,"
           ");
/* Read the next request */
if (fscanf(infile, fmt4,
           ioarea.action,
           ioarea.lname,
           ioarea.fname,
           ioarea.eno,
           ioarea.newno,
           trail) == 6)
{
    if ((newlloc = strpbrk(ioarea.lname, "\n")) != NULL)
        *newlloc = ' ';            /* change to blank for now        */
    if ((newlloc = strpbrk(ioarea.fname, "\n")) != NULL)
        *newlloc = ' ';            /* change to blank for now        */
    ioarea.eno[6] = '\0';
    ioarea.newno[4] = '\0';
    return 0;
} else
    return 1;
} /* end Read_req */

/*******************************************************************************/
/* Function to echo a request                                            */
/*******************************************************************************/
void Echo_req(FILE *outfile)
{
    /* Local declarations */
    struct                                      /* report header     */
    {
        char hdr011[31];
        char hdr012[33];
    } hdr0 = {
        "REQUEST  LAST NAME        ",
        "FIRST NAME    EMPNO    NEW XT.NO"};
    #define rpthdr0 hdr0.hdr011
    static char fmt5[] =                        /* output format     */
        "\n\n %s
 %s%s
   --%-7s--%-15s--%-12s--%-7s--%-9s--
";
    /* End local declarations */
    /* Display the request */
    DSN8MDG(module, "000I", outmsg);
    fprintf(outfile, fmt5,
            outmsg,
            hdr0.hdr011,
            hdr0.hdr012,
            ioarea.action,
            ioarea.lname,
            ioarea.fname,
            ioarea.eno,
            ioarea.newno);
    return;
} /* end Echo_req */

/*******************************************************************************/
/* main program routine                                                  */
/*******************************************************************************/
extern main()
{
    int retcode;
    FILE *cardin;                    /* Input control cards            */
    FILE *report;                    /* Output phone report            */
    /* Open the input and output files */
    cardin = fopen
        ("DD:CARDIN","r,recfm=fb,lrecl=80,blksize=80");
    report = fopen("DD:REPORT","w");
    /* While more input, process */
    while (!feof(cardin))
    {
        /* Read the next request */
        retcode = Read_req(cardin);
        if (retcode == 0)
        {
            /* Display the request */
            Echo_req(report);
            Do_req(report);
        } else
            echo_req(cardin);
    } /* while (!feof(cardin)) */
} /* main */
Related reference
“Sample applications in TSO” on page 1027
A set of Db2 sample applications run in the TSO environment.

**DSN8BP3**

THIS MODULE LISTS EMPLOYEE PHONE NUMBERS AND UPDATES THEM IF DESIRED.

```plaintext
DSN8BP3: PROC REORDER OPTIONS(MAIN);
/******************************************************************************
  *   MODULE NAME = DSN8BP3
  *
  *   DESCRIPTIVE NAME = DB2 SAMPLE APPLICATION
  *                       PHONE APPLICATION
  *                       BATCH
  *                       PL/I
  *
  *   LICENSED MATERIALS - PROPERTY OF IBM
  *   5695-DB2
  *   (C) COPYRIGHT 1982, 1995 IBM CORP. ALL RIGHTS RESERVED.
  *
  *   STATUS = VERSION 4
  *
  *   FUNCTION = THIS MODULE LISTS EMPLOYEE PHONE NUMBERS AND
  *              UPDATES THEM IF DESIRED.
  *
  *   NOTES = NONE
  *
  *   MODULE TYPE = PL/I PROC OPTIONS(MAIN)
  *      PROCESSOR = DB2 PRECOMPILER, PL/I OPTIMIZER
  *      MODULE SIZE = SEE LINK EDIT
  *      ATTRIBUTES = REENTRANT
  *
  *   ENTRY POINT = DSN8BP3
  *      PURPOSE = SEE FUNCTION
  *      LINKAGE = INVOKED FROM DSN RUN
  *      INPUT =
  *
  *        SYMBOLIC LABEL/NAME = CARDIN
  *        DESCRIPTION = INPUT REQUEST FILE
  *        SYMBOLIC LABEL/NAME = VPHONE
  *        DESCRIPTION = VIEW OF TELEPHONE INFORMATION
  *
  *      OUTPUT =
  *
  *        SYMBOLIC LABEL/NAME = REPORT
  *        DESCRIPTION = REPORT OF EMPLOYEE PHONE NUMBERS
  *        SYMBOLIC LABEL/NAME = VEMPLP
  *        DESCRIPTION = VIEW OF EMPLOYEE INFORMATION
  *
  *   EXIT-NORMAL = RETURN CODE 0 NORMAL COMPLETION
  *
  *   EXIT-ERROR =
  *
  *      RETURN CODE = NONE
  *
  *      ABEND CODES = NONE
  *
  *      ERROR-MESSAGES =
  *        DSN8004I - EMPLOYEE SUCCESSFULLY UPDATED
  *        DSN8007E - EMPLOYEE DOES NOT EXIST, UPDATE NOT DONE
  *        DSN8008I - NO EMPLOYEE FOUND IN TABLE
  *        DSN8053I - ROLLBACK SUCCESSFUL, ALL UPDATES REMOVED
  *        DSN8060E - SQL ERROR, RETURN CODE IS:
  *        DSN8061E - ROLLBACK FAILED, RETURN CODE IS:
  *        DSN8065E - INVALID REQUEST, SHOULD BE 'L' OR 'U'
  *        DSN8075E - MESSAGE FORMAT ROUTINE ERROR,
  *                       RETURN CODE IS :
```
/* EXTERNAL REFERENCES = */
/* ROUTINES/SERVICES = */
/* DSN8MPG            - ERROR MESSAGE ROUTINE */
/* DATA-AREAS = NONE */
/* CONTROL-BLOCKS = */
/* SQLCA               - SQL COMMUNICATION AREA */
/* TABLES = NONE */
/* CHANGE-ACTIVITY = NONE */
/* */
/* *PSEUDOCODE* */
/* */
/* PROCEDURE */
/* GET FIRST INPUT */
/* DO WHILE MORE INPUT */
/* CREATE REPORT HEADING */
/* CASE (ACTION) */
/* */
/* SUBCASE ('L') */
/* IF LASTNAME IS '*' THEN */
/* LIST ALL EMPLOYEES */
/* ELSE */
/* IF LASTNAME CONTAINS '%' THEN */
/* LIST EMPLOYEES GENERIC */
/* ELSE */
/* LIST EMPLOYEES SPECIFIC */
/* ENDSUB */
/* */
/* SUBCASE ('U') */
/* UPDATE PHONENUMBER FOR EMPLOYEE */
/* WRITE CONFIRMATION MESSAGE */
/* OTHERWISE */
/* INVALID REQUEST */
/* ENDSUB */
/* */
/* ENDCASE */
/* GET NEXT INPUT */
/* END */
/* */
/* IF SQL ERROR OCCURS THEN */
/* ROLLBACK */
/* END. */
/* */
/* 1/*********************/
/* INPUT/OUTPUT FILES */
/* 1/*********************/
DCL CARDIN FILE STREAM INPUT; /* INPUT CONTROL CARDS */
DCL REPORT FILE STREAM OUTPUT PRINT; /* OUTPUT PHONE REPORT */

1/*********************/
/* ENDFILE HANDLING */
1/*********************/
ON ENDFILE (CARDIN) EOF = '1'B;

1/*********************/
/* STRUCTURE FOR INPUT */
1/*********************/
DCL 1 IOAREA,
2 ACTION CHAR( 1),      /* ACTION */
2 LNAME  CHAR(15),      /* LAST NAME */
2 FNAME  CHAR(12),      /* FIRST NAME */
2 ENO    CHAR( 6),      /* EMPLOYEE NUMBER */
2 NEWNO  CHAR( 4);      /* PHONE NUMBER */

1/*********************/
/* WORK VARIABLES */
1/*********************/
DCL LNAMEWK CHAR(15) VAR; /* WORK VERSION OF LAST NAME */
DCL FNAMEWK CHAR(12) VAR; /* WORK VERSION OF FIRST NAME */

1/*********************/
/* REPORT HEADER STRUCTURE */

Chapter 12. Sample data and applications supplied with Db2 for z/OS 1049
DCL 1 REPHDR1 STATIC,
  2 HDR111 CHAR(29) INIT ((29)'-'),
  2 HDR112 CHAR(21) INIT (' TELEPHONE DIRECTORY '),
  2 HDR113 CHAR(28) INIT ((28)'-');
DCL 1 REPHDR2 STATIC,
  2 HDR211 CHAR( 9) INIT ('LAST NAME'),
  2 HDR212 CHAR(10) INIT ('FIRST NAME'),
  2 HDR213 CHAR( 7) INIT ('INITIAL'),
  2 HDR214 CHAR( 5) INIT ('PHONE'),
  2 HDR215 CHAR( 8) INIT ('EMPLOYEE'),
  2 HDR216 CHAR( 4) INIT ('WORK'),
  2 HDR217 CHAR( 4) INIT ('WORK'),
  2 HDR218 CHAR( 6) INIT ('NUMBER'),
  2 HDR219 CHAR( 6) INIT ('NUMBER'),
  2 HDR220 CHAR( 4) INIT ('DEPT'),
  2 HDR221 CHAR( 4) INIT ('DEPT'),
  2 HDR222 CHAR( 4) INIT ('NAME');

L1: FORMAT (A(29),A(21),A(28));
L2: FORMAT (SKIP(2),A(9),X(7),A(10),X(3),A(7),X(1),A(5),X(2),A(8),
          X(1),A(4),X(1),A(4),SKIP,X(37),A(6),X(1),A(6),X(3),
          A(4),X(1),A(4),X(1),A(4));
L3: FORMAT (SKIP,A(15),X(1),A(12),X(4),A(1),X(4),A(4),X(3),A(6),X(3),
          A(3),X(2),A(36));
L4: FORMAT (COL(1),A(1),A(15),A(12),A(6),A(4));

EXEC SQL DECLARE VPHONE TABLE
   (LASTNAME       VARCHAR(15)  NOT NULL,
    FIRSTNAME      VARCHAR(12)  NOT NULL,
    MIDDLEINITIAL  CHAR(1) NOT NULL,
    PHONENUMBER    CHAR(4) ,
    EMPLOYEENUMBER CHAR(6) NOT NULL,
    DEPTNUMBER     CHAR(3) NOT NULL,
    DEPTNAME       VARCHAR(36) NOT NULL);

EXEC SQL DECLARE VPHONE TABLE
   (LASTNAME       VARCHAR(15) NOT NULL,
    FIRSTNAME      VARCHAR(12) NOT NULL,
    MIDDLEINITIAL  CHAR(1) NOT NULL,
    PHONENUMBER    CHAR(4) ,
    EMPLOYEENUMBER CHAR(6) NOT NULL,
    DEPTNUMBER     CHAR(3) NOT NULL,
    DEPTNAME       VARCHAR(36) NOT NULL);

EXEC SQL INCLUDE SQLCA;

EXEC SQL INCLUDE SQLCA;

EXEC SQL INCLUDE SQLCA;

EXEC SQL INCLUDE SQLCA;

EXEC SQL INCLUDE SQLCA;

EXEC SQL INCLUDE SQLCA;

EXEC SQL INCLUDE SQLCA;

EXEC SQL INCLUDE SQLCA;

EXEC SQL INCLUDE SQLCA;

EXEC SQL INCLUDE SQLCA;
EXEC SQL DECLARE VEMPLP TABLE  
(EMPLOYEENUMBER    CHAR( 6)  NOT NULL,  
PHONENUMBER       CHAR( 4)          );

EXEC SQL DECLARE TELE1 CURSOR FOR  
SELECT *  
FROM VPHONE;

EXEC SQL DECLARE TELE2 CURSOR FOR  
SELECT *  
FROM VPHONE  
WHERE LASTNAME LIKE :LNAMEWK  
AND FIRSTNAME LIKE :FNAMEWK;

EXEC SQL DECLARE TELE3 CURSOR FOR  
SELECT *  
FROM VPHONE  
WHERE LASTNAME = :LNAMEWK  
AND FIRSTNAME LIKE :FNAMEWK;

EXEC SQL WHENEVER SQLERROR   GOTO DBERROR;
EXEC SQL WHENEVER SQLWARNING GOTO DBERROR;
EXEC SQL WHENEVER NOT FOUND  CONTINUE;

GET FILE (CARDIN) EDIT (IOAREA) (R(L4)); /* READ FIRST REQUEST */
/* PROCESS INPUT REQUESTS */
DO WHILE (^EOF); /* CONTINUE WHILE MORE TO DO */
/* PUT REPORT HEADINGS */
/* CREATE REPORT HEADING */
/* SELECT ACTION */
/* GET WORKING VERSIONS OF LAST AND FIRST NAMES WITH NO TRAILING BLANKS */
/* LAST AND FIRST NAMES WITH */
/* NO TRAILING BLANKS */
/* SEARCH FOR ALL NAMES */
SELECT (ACTION); /* DETERMINE INPUT REQUEST */

/*****************************/
/* LIST ALL EMPLOYEES */
/*****************************/
WHEN ('L') DO; /* LIST EMPLOYEES */
IF LNAME = '*' THEN /* LIST ALL EMPLOYEES */
    DO;
        EXEC SQL OPEN TELE1; /* OPEN CURSOR FOR SEARCH */
        EXEC SQL FETCH TELE1 INTO :PPHONE; /* GET FIRST RECORD */
        IF SQLCODE = NOTFOUND THEN /* NO RECORDS FOUND */
            DO; /* GET ERROR MESSAGE */
                CALL DSN8MPG (MODULE, '008I', OUTMSG);
                PUT FILE (REPORT) EDIT (OUTMSG) (SKIP(2),A);
            END;
            /* GET AND PRINT ALL RECORDS */
        DO WHILE (SQLCODE = ZERO);
            PUT FILE (REPORT) EDIT (PPHONE) (R(L3));
            EXEC SQL FETCH TELE1 INTO :PPHONE; /* GET NEXT RECORD */
            END; /* END DO WHILE */
        EXEC SQL CLOSE TELE1; /* CLOSE CURSOR FOR SEARCH */
        END; /* END DO IF */
    END; /* END IF */
ELSE /* SELECT EMPLOYEES BY NAME */
    IF INDEX(LNAMEWK,'%') > ZERO THEN /* SEARCH ON PART OF NAME? */
        DO; /* YES: SEARCH ON PART OF */
            IF LNAMEWK = '*' THEN /* LAST NAME */
                EXEC SQL OPEN TELE2; /* OPEN CURSOR FOR SEARCH */
                EXEC SQL FETCH TELE2 INTO :PPHONE; /* GET 1ST RECORD */
            IF SQLCODE = NOTFOUND THEN /* NO RECORDS FOUND */
                DO; /* GET ERROR MESSAGE */
                    CALL DSN8MPG (MODULE, '008I', OUTMSG);
                    PUT FILE (REPORT) EDIT (OUTMSG) (SKIP(2),A);
                END;
                /* GET AND PRINT ALL RECORDS */
            DO WHILE (SQLCODE = ZERO);
                PUT FILE (REPORT) EDIT (PPHONE) (R(L3));
                EXEC SQL FETCH TELE2 INTO :PPHONE; /* GET NEXT RECORD */
                END; /* END DO WHILE */
            EXEC SQL CLOSE TELE2; /* CLOSE CURSOR FOR SEARCH */
        END; /* END DO IF */
    ELSE /* NO - SEARCH ON LAST NAME */
        /* & Optionally FIRST NAME */
        EXEC SQL OPEN TELE3; /* OPEN CURSOR FOR SEARCH */
        EXEC SQL FETCH TELE3 INTO :PPHONE; /* GET 1ST RECORD */
    IF SQLCODE = NOTFOUND THEN /* NO RECORDS FOUND */
        DO; /* GET ERROR MESSAGE */
            CALL DSN8MPG (MODULE, '008I', OUTMSG);
            PUT FILE (REPORT) EDIT (OUTMSG) (SKIP(2),A);
        END;
        /* GET AND PRINT ALL RECORDS */
    DO WHILE (SQLCODE = ZERO);
        PUT FILE (REPORT) EDIT (PPHONE) (R(L3));
        EXEC SQL FETCH TELE3 INTO :PPHONE; /* GET NEXT RECORD */
        END; /* END DO WHILE */
    EXEC SQL CLOSE TELE3; /* CLOSE CURSOR FOR SEARCH */
    END; /* END DO ELSE */
END; /* END WHEN */

/*****************************/
/* LIST GENERIC EMPLOYEES */
/*****************************/
ELSE /* SELECT EMPLOYEES BY NAME */
    IF INDEX(LNAMEWK,'%') > ZERO THEN /* SEARCH ON PART OF NAME? */
        DO; /* YES: SEARCH ON PART OF */
            /* & LAST NAME */
            EXEC SQL OPEN TELE2; /* OPEN CURSOR FOR SEARCH */
            EXEC SQL FETCH TELE2 INTO :PPHONE; /* GET 1ST RECORD */
        IF SQLCODE = NOTFOUND THEN /* NO RECORDS FOUND */
            DO; /* GET ERROR MESSAGE */
                CALL DSN8MPG (MODULE, '008I', OUTMSG);
                PUT FILE (REPORT) EDIT (OUTMSG) (SKIP(2),A);
            END;
            /* GET AND PRINT ALL RECORDS */
        DO WHILE (SQLCODE = ZERO);
            PUT FILE (REPORT) EDIT (PPHONE) (R(L3));
            EXEC SQL FETCH TELE2 INTO :PPHONE; /* GET NEXT RECORD */
            END; /* END DO WHILE */
        EXEC SQL CLOSE TELE2; /* CLOSE CURSOR FOR SEARCH */
    END; /* END DO IF */
ELSE /* NO - SEARCH ON LAST NAME */
    /* & Optionally FIRST NAME */
    EXEC SQL OPEN TELE3; /* OPEN CURSOR FOR SEARCH */
    EXEC SQL FETCH TELE3 INTO :PPHONE; /* GET 1ST RECORD */
    IF SQLCODE = NOTFOUND THEN /* NO RECORDS FOUND */
        DO; /* GET ERROR MESSAGE */
            CALL DSN8MPG (MODULE, '008I', OUTMSG);
            PUT FILE (REPORT) EDIT (OUTMSG) (SKIP(2),A);
        END;
        /* GET AND PRINT ALL RECORDS */
    DO WHILE (SQLCODE = ZERO);
        PUT FILE (REPORT) EDIT (PPHONE) (R(L3));
        EXEC SQL FETCH TELE3 INTO :PPHONE; /* GET NEXT RECORD */
        END; /* END DO WHILE */
    EXEC SQL CLOSE TELE3; /* CLOSE CURSOR FOR SEARCH */
END; /* END DO ELSE */
END; /* END WHEN */

/*****************************/
/* LIST SPECIFIC EMPLOYEES */
/*****************************/
ELSE /* NO - SEARCH ON LAST NAME */
    IF LNAMEWK = '*' THEN /* LAST NAME */
        /* & OPTIONALLY FIRST NAME */
        /* SEE IF FIRST NAME ENTERED */
        EXEC SQL OPEN TELE3; /* OPEN CURSOR FOR SEARCH */
        EXEC SQL FETCH TELE3 INTO :PPHONE; /* GET 1ST RECORD */
    IF SQLCODE = NOTFOUND THEN /* NO RECORDS FOUND */
        DO; /* GET ERROR MESSAGE */
            CALL DSN8MPG (MODULE, '008I', OUTMSG);
            PUT FILE (REPORT) EDIT (OUTMSG) (SKIP(2),A);
        END;
        /* GET AND PRINT ALL RECORDS */
    DO WHILE (SQLCODE = ZERO);
        PUT FILE (REPORT) EDIT (PPHONE) (R(L3));
        EXEC SQL FETCH TELE3 INTO :PPHONE; /* GET NEXT RECORD */
        END; /* END DO WHILE */
    EXEC SQL CLOSE TELE3; /* CLOSE CURSOR FOR SEARCH */
END; /* END DO ELSE */
END; /* END WHEN */

/*****************************/
/* UPDATES PHONE NUMBERS FOR EMPLOYEES */
/*****************************/
WHEN ('U') DO; /* TELEPHONE UPDATE */
EXEC SQL UPDATE VEMPLP

SET PHONENUMBER = :NEWNO /* CHANGE PHONE NO. */
WHERE EMPLOYEE_NUMBER = :ENO;

IF SQLCODE = ZERO THEN /* WAS UPDATE OK? */
DO;
   CALL DSN8MPG (MODULE, '004I', OUTMSG); /* YES */
   PUT FILE (REPORT) EDIT (OUTMSG) (SKIP(2),A); /* YES */
   END; /* EMPLOYEE FOUND */
   /UPDATE SUCCESSFUL*/
ELSE
   CALL DSN8MPG (MODULE, '007E', OUTMSG); /* UPDATE FAILED*/
   PUT FILE (REPORT) EDIT (OUTMSG) (SKIP(2),A);
   END; /* END ELSE*/
END;

/* UPDATE SUCCESSFUL*/

ELSE /* INVALID REQUEST */
DO;
   CALL DSN8MPG (MODULE, '068E', OUTMSG);
   PUT FILE (REPORT) EDIT (OUTMSG) (SKIP(2),A);
   END; /* END ELSE*/
END; /* END WHEN*/

GET FILE (CARDIN) EDIT (IOAREA) (R(L4)); /* READ NEXT REQUEST */
END; /* END EOF*/
GOTO PGMEND; /* BYPASS SQL ERROR HANDLING */

 Chapter 12. Sample data and applications supplied with Db2 for z/OS 1053
ELSE
DO;                                        /* ROLLBACK FAILED,*/
/* RETURN CODE IS: */
CALL DSN8MPG (MODULE, '061E', OUTMSG);
PUT FILE (REPORT) EDIT (OUTMSG,SQLCODE) (SKIP(2),A,F(10));
END;
PGMEND:                                        /*  PROGRAM END    */
END;
END;

Related reference
“Sample applications in TSO” on page 1027
A set of Db2 sample applications run in the TSO environment.

DSN8BF3
THIS MODULE LISTS EMPLOYEE PHONE NUMBERS AND UPDATES THEM IF DESIRED.

PROGRAM DSN8B3
****************************************************************************
*                                                                    *
*   MODULE NAME = DSN8BF3, PROGRAM DSN8B3                            *
*                                                                    *
*   DESCRIPTIVE NAME = DB2  SAMPLE APPLICATION                       *
*                      PHONE APPLICATION                             *
*                      BATCH                                         *
*                      FORTRAN                                       *
*                                                                    *
*   LICENSED MATERIALS - PROPERTY OF IBM                             *
* 5695-DB2                                                          *
* (C) COPYRIGHT 1982, 1995 IBM CORP.  ALL RIGHTS RESERVED.           *
*                                                                    *
*   STATUS = VERSION 4                                               *
*                                                                    *
*   FUNCTION = THIS MODULE LISTS EMPLOYEE PHONE NUMBERS AND          *
*              UPDATES THEM IF DESIRED.                              *
*                                                                    *
*   NOTES = NONE                                                     *
*                                                                    *
*                                                                    *
*   MODULE TYPE = FORTRAN PROGRAM                                    *
*      PROCESSOR = DB2  PRECOMPILED, VS FORTRAN                      *
*      MODULE SIZE = SEE LINK EDIT                                   *
*      ATTRIBUTES = NOT REENTRANT OR REUSABLE                        *
*                                                                    *
*   ENTRY POINT = DSN8BF3                                            *
*      PURPOSE  = SEE FUNCTION                                       *
*      LINKAGE  = INVOKED FROM DSN RUN                               *
*      INPUT    =                                                    *
*                                                                    *
*                 SYMBOLIC LABEL/NAME = FT05F001                     *
*                 DESCRIPTION = INPUT REQUEST FILE                   *
*                                                                    *
*                 SYMBOLIC LABEL/NAME = VPHONE                       *
*                 DESCRIPTION = VIEW OF TELEPHONE INFORMATION        *
*                                                                    *
*   EXIT-NORMAL = RETURN CODE 0 NORMAL COMPLETION                    *
*                                                                    *
*   EXIT-ERROR =                                                     *
*                                                                    *
*      RETURN CODE = NONE                                            *
*                                                                    *
*      ABEND CODES =  NONE                                           *
*                                                                    *
*      ERROR-MESSAGES =                                              *
*            DSN8000I - REQUEST IS:  ...
*            DSN8004I - EMPLOYEE SUCCESSFULLY UPDATED                

1054  Db2 12 for z/OS: Application Programming and SQL Guide
* DSN8007E - EMPLOYEE DOES NOT EXIST, UPDATE NOT DONE  
* DSN8008I - NO EMPLOYEE FOUND IN TABLE  
* DSN8051I - PROGRAM ENDED  
* DSN8053I - ROLLBACK SUCCESSFUL, ALL UPDATES REMOVED  
* DSN8060E - SQL ERROR, RETURN CODE IS:  
* DSN8061E - ROLLBACK FAILED, RETURN CODE IS:  
* DSN8066E - INVALID REQUEST, SHOULD BE 'L' OR 'U'  
* DSN8075E - MESSAGE FORMAT ERROR,  
  RETURN CODE IS:  
*  
* EXTERNAL REFERENCES =  
* ROUTINES/SERVICES =  
  * DSNTIR - TRANSLATE SQLCA INTO MESSAGES  
*  
* DATA-AREAS = NONE  
*  
* CONTROL-BLOCKS =  
  * SQLCA - SQL COMMUNICATION AREA  
*  
* TABLES = NONE  
*  
* CHANGE-ACTIVITY = NONE  
*  
* EXEC SQL DECLARE VPHONE TABLE  
  EXEC SQL DECLARE VEMPLP TABLE  
*  
* PROCEDURE  
*  
* DO WHILE MORE INPUT  
*  
* GET INPUT  
*  
* CREATE REPORT HEADING  
*  
* CASE (ACTION)  
*  
* SUBCASE ('L')  
*    IF LASTNAME IS '*' THEN  
*      LIST ALL EMPLOYEES  
*    ELSE  
*      IF LASTNAME CONTAINS '%' THEN  
*        LIST EMPLOYEES GENERIC  
*      ELSE  
*        LIST EMPLOYEES SPECIFIC  
*    ENDSUB  
*  
* SUBCASE ('U')  
*    UPDATE PHONENUMBER FOR EMPLOYEE  
*    WRITE CONFIRMATION MESSAGE  
*  
* OTHERWISE  
*  
* INVALID REQUEST  
*  
* ENDSUB  
*  
* ENDCASE  
*  
* GET NEXT INPUT  
*  
* END  
*  
* IF SQL ERROR OCCURS THEN  
*  
* ROLLBACK  
*  
* END.  
*  
* Chapter 12. Sample data and applications supplied with Db2 for z/OS  1055
* PHONE FIELDS *
*****************************************************
CHARACTER * 15 LASTNM
CHARACTER * 12 FIRSTN
CHARACTER *  1 MIDINI
CHARACTER *  4 PHONEN
CHARACTER *  6 EMPNO
CHARACTER *  3 DEPTNO
CHARACTER * 36 DEPTNM
*****************************************************
* INPUT FIELDS *
*****************************************************
CHARACTER *  1 ACTION
CHARACTER * 15 LNAME
CHARACTER * 12 FNAME
CHARACTER *  6 ENO
CHARACTER *  4 NEWNO
CHARACTER * 15 LNAMEW
CHARACTER * 12 FNAMEW
*****************************************************
* SQL CURSORS *
*****************************************************
EXEC SQL DECLARE TELE1 CURSOR FOR
C    SELECT * 
C    FROM VPHONE
EXEC SQL DECLARE TELE2 CURSOR FOR
C    SELECT * 
C    FROM VPHONE
C    WHERE LASTNAME LIKE :LNAMEW
C    AND FIRSTNAME LIKE :FNAMEW
EXEC SQL DECLARE TELE3 CURSOR FOR
C    SELECT * 
C    FROM VPHONE
C    WHERE LASTNAME = :LNAME
C    AND FIRSTNAME LIKE :FNAMEW
*****************************************************
* SQL RETURN CODES: OK/NOTFOUND *
*****************************************************
INTEGER OK/0/,NOTFND/100/
*****************************************************
* REPORT FORMATS AND INPUT *
*****************************************************
100 FORMAT ('0',A29,A21,A28)
200 FORMAT ('0',A9,7X,A10,3X,A7,1X,A5,2X,A8,
C       1X,A4,1X,A4,/,38X,A6,1X,A6,3X,
C       A4,1X,A4,1X,A4,/,)
300 FORMAT (' ',A15,1X,A12,4X,A1,4X,A4,3X,A6,3X,
C       A3,2X,A36)
400 FORMAT (A1,A15,A12,A6,A4)
500 FORMAT ('0', A)
600 FORMAT ('0', A, I8)
700 FORMAT ('1', A, //,
C      S5,'REQUEST',2X,'LAST NAME',8X,'FIRST NAME',4X,
C      'EMPNO',3X,'NEW XT.NO',//,
C      3X,'--',A1,6X,'--',A15,'--',A12,'--',A6,'--',A4,'--')
800 FORMAT ('1',)
*****************************************************
* MESSAGES *
*****************************************************
CHARACTER * 30 DSN800
CHARACTER * 48 DSN804
CHARACTER * 60 DSN868
CHARACTER * 59 DSN807
CHARACTER * 45 DSN869
CHARACTER * 59 DSN853
CHARACTER * 51 DSN861
CHARACTER * 45 DSN808
CHARACTER * 64 DSN875
CHARACTER * 32 DSN851
* VARIABLES USED WITH DSNTIR *

** MISCELLANEOUS VARIABLES **

* SQL COMMUNICATION AREA *

EXEC SQL INCLUDE SQLCA

* DATA STATEMENTS *

DATA           PERC15
C/'%%%%%%%%%%%%%%%%%'/
DATA           DSN800
C/'DSN8000I:  DSN8BF3-REQUEST IS:'/
DATA           DSN804
C/'DSN8004I:  DSN8BF3-EMPLOYEE SUCCESSFULLY UPDATED'/
DATA           DSN808
C/'DSN8008E:  DSN8BF3-INVALID REQUEST, SHOULD BE 'L' OR 'U'/'/
DATA           DSN807
C/'DSN8007E:  DSN8BF3-EMPLOYEE DOES NOT EXIST, UPDATE NOT DONE'/
DATA           DSN860
C/'DSN8060E:  DSN8BF3-SQL ERROR, RETURN CODE IS:'/
DATA           DSN853
C/'DSN8053I:  DSN8BF3-ROLLBACK SUCCESSFUL, ALL UPDATES REMOVED'/
DATA           DSN861
C/'DSN8061E:  DSN8BF3-ROLLBACK FAILED, SQLCODE IS:'/
DATA           DSN808
C/'DSN8008I:  DSN8BF3-NO EMPLOYEE FOUND IN TABLE'/
DATA           DSN875
C/'DSN8075E:  DSN8BF3-MESSAGE FORMAT ROUTINE ERROR, RETURN CODE IS:
C'/
DATA           DSN851
C/'DSN8051I:  DSN8BF3-PROGRAM ENDED'/

* SQL RETURN CODE HANDLING *

EXEC SQL WHENEVER SQLERROR   GOTO 4000
EXEC SQL WHENEVER SQLWARNING GOTO 4000

* START OF PROGRAM *

* CONTINUE WHILE MORE INPUT *

1000 CONTINUE

* GET NEXT INPUT *

READ (UNIT=05,FMT=400,END=3000) ACTION, LNAME, FNAME, ENO, NEWNO
WRITE (UNIT=06,FMT=700) DSN800, ACTION, LNAME, FNAME, ENO, NEWNO
WRITE (UNIT=06,FMT=800)

* CREATE REPORT HEADING *

**CREATE REPORT HEADING

WRITE (UNIT=06,FMT=100) '-----------------------------------------------',

Chapter 12. Sample data and applications supplied with Db2 for z/OS 1057
C ' TELEPHONE DIRECTORY ',
C '----------------------------'
WRITE (UNIT=06,FMT=200) 'LAST NAME', 'FIRST NAME', 'INITIAL',
C 'PHONE', 'EMPLOYEE', 'WORK', 'WORK', 'NUMBER',
C 'NUMBER', 'DEPT', 'DEPT', 'NAME'
*
**SELECT ACTION
*
IF (ACTION .EQ. 'L') THEN
GOTO 1010
*
ELSE IF (ACTION .EQ. 'U') THEN
GOTO 1700
*
ELSE
GOTO 1800
END IF
1010 CONTINUE
***********************************************
* ACTION - LIST                              *
***********************************************
IF (LNAME .NE. '*') GOTO 1300
***********************************************
* LIST ALL EMPLOYEES                          *
***********************************************
*
EXEC SQL OPEN TELE1
NBRETR = 0
1100 CONTINUE
*
EXEC SQL FETCH TELE1 INTO
C :LASTNM,:FIRSTN,:MIDINI,
C :PHONEN,:EMPNO,:DEPTNO,:DEPTNM
IF (SQLCOD .EQ. NOTFND) GO TO 1200
*
EXEC SQL CLOSE TELE1
GO TO 1000
*
ELSE DETERMINE IF LASTNAME *
* OR FIRSTNAME IS GIVEN      *
1300 CONTINUE
*
IPOS=INDEX(LNAME,'%')
***********************************************
* REPLACE FIRST BLANK AND FOLLOWING          *
* CHARACTERS IN LASTNAME WORK (LNAMEW)        *
* WITH CHARACTER % FOR LIKE PREDICATE.        *
***********************************************
IBLANK=INDEX(LNAME,' ')
IF (IBLANK .GT. 1 ) THEN
LNAMEW = LNAME(1:IBLANK-1)//PERC15(1:15-IBLANK+1)
ELSE IF (IBLANK .EQ. 1) THEN
LNAMEW=PERC15
IPOS = 1
ELSE
IPOS = 0
ENDIF

1200 CONTINUE
IF (NBRETR .EQ. 0) WRITE (UNIT=06,FMT=500) DSN808
*
EXEC SQL CLOSE TELE1
GO TO 1000
ELSE

********************************************
* REPLACE FIRST BLANK AND FOLLOWING     *
* CHARACTERS IN FIRSTNAME WORK (FNAMEW) *
* WITH CHARACTER % FOR LIKE PREDICATE.  *
********************************************

IBLANK=INDEX(FNAME,' ')
IF (IBLANK .GT. 1 ) THEN
  FNAMEW = FNAME(1:IBLANK-1)//PERC15(1:12-IBLANK+1)
ELSE IF (IBLANK .EQ. 1) THEN
  FNAMEW=PERC15(1:12)
ELSE
END IF

IF (IPOS .LE. 0) GOTO 1600

********************************************
*     LIST GENERIC EMPLOYEES              *
********************************************

*                                               **OPEN CURSOR
EXEC SQL OPEN TELE2
NBRETR = 0

1400        CONTINUE
*                                               **GET EMPLOYEES
EXEC SQL FETCH TELE2 INTO
  C                    :LASTNM,:FIRSTN,:MIDINI,
  C                    :PHONEN,:EMPNO,:DEPTNO,:DEPTNM
IF (SQLCOD .EQ. NOTFND) GO TO 1500
*                                               **LIST GENERIC EMPLOYEES
  NBRETR = NBRETR + 1
  WRITE (UNIT=06,FMT=300)
  C                    LASTNM,FIRSTN,MIDINI,
  C                    PHONEN,EMPNO,DEPTNO, DEPTNM
GOTO 1400
*                                               **EMPLOYEE NOT FOUND
*                                               **PRINT ERROR MESSAGE
1500 CONTINUE
IF (NBRETR .EQ. 0) WRITE (UNIT=06,FMT=500) DSN808
*                                               **CLOSE CURSOR
EXEC SQL CLOSE TELE2
GOTO 1000

********************************************
*    LIST SPECIFIC EMPLOYEES               *
********************************************

1600 CONTINUE
*                                               **OPEN CURSOR
EXEC SQL OPEN TELE3
NBRETR = 0

1620 CONTINUE
*                                               **GET EMPLOYEES
EXEC SQL FETCH TELE3 INTO
  C                    :LASTNM,:FIRSTN,:MIDINI,
  C                    :PHONEN,:EMPNO,:DEPTNO,:DEPTNM
IF (SQLCOD .EQ. NOTFND) GO TO 1640
*                                               **LIST SPECIFIC EMPLOYEES
  NBRETR = NBRETR + 1
  WRITE (UNIT=06,FMT=300)
  C                    LASTNM,FIRSTN,MIDINI,
  C                    PHONEN,EMPNO,DEPTNO, DEPTNM
GOTO 1620
*                                               **EMPLOYEE NOT FOUND
*                                               **PRINT ERROR MESSAGE
1640 CONTINUE
IF (NBRETR .EQ. 0) WRITE (UNIT=06,FMT=500) DSN808
EXEC SQL CLOSE TELE3  
GO TO 1000

UPDATE PHONE NUMBERS FOR EMPLOYEES

1700 CONTINUE
   EXEC SQL UPDATE VEMPLP
   SET PHONENUMBER = :NEWNO
   WHERE EMPLOYEEENUMBER = :ENO
   IF (SQLCOD .EQ. OK) THEN
      WRITE (UNIT=06,FMT=500) DSN804
   ELSE
      WRITE (UNIT=06,FMT=500) DSN807
   END IF
   GO TO 1000

1800 CONTINUE
   WRITE (UNIT=06,FMT=500) DSN868
   GO TO 1000

***************
* END OF LOOP  
* FOR MORE INPUT*
***************

3000 CONTINUE
   WRITE (UNIT=06,FMT=800)
   WRITE (UNIT=06,FMT=500) DSN851
   RETURN

***************
* IF SQL ERROR OCCURRED - GET ERROR MESSAGE*
***************
EXEC SQL WHENEVER SQLERROR CONTINUE
EXEC SQL WHENEVER SQLWARNING CONTINUE
EXEC SQL WHENEVER NOT FOUND CONTINUE

4000 CONTINUE
   WRITE (UNIT=06,FMT=600) DSN860, SQLCOD
   CALL DSNTIR ( ERRLEN, ERRTXT, ICODE )
   IF (ICODE .EQ. OK) THEN
      DO 4100 I=1, 10
         WRITE (UNIT=06,FMT=500) ERRTXT(I)
      4100 CONTINUE
   ELSE
      WRITE (UNIT=06,FMT=600) DSN875, ICODE
   END IF
   EXEC SQL ROLLBACK
   IF (SQLCOD .EQ. OK) THEN
**ROLLBACK SUCCESSFUL**
**PRINT CONFIRMATION**
**MESSAGE**
```
WRITE (UNIT=06,FMT=500) DSN853
ELSE
**ROLLBACK FAILED**
**PRINT ERROR MESSAGE**
WRITE (UNIT=06,FMT=600) DSN861, SQLCOD
END IF
RETURN
END
```

**Related reference**
“Sample applications in TSO” on page 1027
A set of Db2 sample applications run in the TSO environment.

**DSN8HC3**

This module displays the Db2 department and employee tables and updates them if desired.

```
IDENTIFICATION DIVISION.
*--------------------
PROGRAM-ID. DSN8HC3.
*---------------------------------------------------------------*
*                                                               *
*   MODULE NAME = DSN8HC3                                       *
*                                                               *
*   DESCRIPTIVE NAME = DB2  SAMPLE APPLICATION                 *
*                      ORGANIZATION APPLICATION                 *
*                      ISPF                                     *
*                      COBOL                                    *
*  LICENSED MATERIALS - PROPERTY OF IBM                         *
*  5615-DB2                                                     *
*  (C) COPYRIGHT 1982, 2013 IBM CORP.  ALL RIGHTS RESERVED.     *
*                                                               *
*   STATUS = VERSION 11                                          *
*                                                               *
*   FUNCTION = THIS MODULE DISPLAYS THE Db2 DEPARTMENT AND      *
*              EMPLOYEE TABLES AND UPDATES THEM IF DESIRED.     *
*                                                               *
*   NOTES =                                                     *
*      DEPENDENCIES = REQUIRED ISPF PANELS:                     *
*                     DSN8SSH                                   *
*                     DSN8SSH1                                  *
*                     DSN8SSH2                                  *
*                     DSN8SSH3                                  *
*                     DSN8SSH4                                  *
*                     DSN8SSH5                                  *
*                                                               *
*      RESTRICTIONS = NONE                                      *
*                                                               *
*   MODULE TYPE = VS COBOL II PROGRAM                           *
*      PROCESSOR   = DB2 PRECOMPILER, VS COBOL II               *
*      MODULE SIZE = SEE LINKEDIT                               *
*      ATTRIBUTES  = NOT REENTRANT OR REUSABLE                  *
*                                                               *
*   ENTRY POINT =  DSN8HC3                                      *
*      PURPOSE = SEE FUNCTION                                   *
*      LINKAGE = INVOKED FROM ISPF                             *
*                                                               *
*   INPUT = PARAMETERS EXPLICITLY PASSED TO THIS FUNCTION:     *
*      INPUT-MESSAGE:                                           *
*                                                               *
*      SYMBOLIC LABEL/NAME = DSN8SSH                            *
*      DESCRIPTION = MAIN MENU                                  *
*                                                               *
*      SYMBOLIC LABEL/NAME = DSN8SSH2                           *
*      DESCRIPTION = DEPARTMENT PANEL                           *
*                                                               *
*      SYMBOLIC LABEL/NAME = DSN8SSH3                           *
*      DESCRIPTION = SELECT FROM LIST PANEL                     *
*                                                               *
*      SYMBOLIC LABEL/NAME = DSN8SSH4                           *
*      DESCRIPTION = SELECT FROM LIST PANEL                     *
```
* CONTROL-BLOCKS = * 00013900
* SQLCA = SQL COMMUNICATION AREA  * 00014000
* TABLES = NONE  * 00014100
* CHANGE-ACTIVITY = NONE  * 00014200
* *PSEUDOCODE*
* SET UP RETURN CODE HANDLING  0000-PROGRAM-START* 00015000
* SET PREVIOUS LOCATION TO LOCAL  * 00015100
* DO UNTIL NO MORE TERMINAL INPUT  * 00015200
* IF ACTION IS ERASE THEN  * 00015300
* IF CURRENT AND PREVIOUS LOCATION DIFFER THEN  * 00015400
* IF REMOTE LOCATION THEN  * 00015500
* CONNECT TO REMOTE LOCATION  * 00015600
* ELSE RESET TO LOCAL LOCATION  * 00015700
* DETERMINE PROCESSING REQUEST  * 00015800
* IF ACTION FIELD ADD:  * 00015900
* IF OBJECT FIELD IS DE:
  ADD RECORD TO VDEPT TABLES  2000-ADDDEPT  * 00016000
  AT ALL LOCATIONS  * 00016100
* ELSE IF OBJECT FIELD IS EM:
  ADD RECORD TO VEMP TABLE  3000-ADDEMP  * 00016200
* ELSE:  * 00016300
* ELSE:
* IF OBJECT FIELD DE OR DS:  5000-DEPARTMENT  * 00016500
* IF "LIST GENERIC":  5100-GENDEPT  * 00016600
* CHOOSE CURSOR BASED ON  5110-GETDEPNTAX  * 00016700
* SEARCH CRITERIA AND DATA  * 00016800
* CREATE ISPF TABLE  * 00016900
* DO UNTIL NO MORE RECORDS:
  FETCH RECORD  * 00017000
  STORE RECORD IN TABLE  * 00017100
* DISPLAY DEPARTMENT LIST  5121-GETDEPT  * 00017200
* ON SCREEN  * 00017300
* STORE SELECTED DEPT ID IN
  HOST VARIABLE  * 00017400
* ELSE:
* ELSE:
* IF OBJFLD IS DS:
  5200-DISPLAYDEPT  * 00017500
  FETCH SELECTED DEPT  * 00017600
  DISPLAY DEPT ON SCREEN  5210-DISDEPTACT  * 00017700
* IF ACTION IS ERASE:
  5220-ERASEDEPT  * 00017800
  AT ALL LOCATIONS  * 00017900
* PERFORM CASCADE DELETE  5223-DELETEDEPT  * 00017900
* ELSE IF OBJECT FIELD IS EM:
  ADD RECORD TO VHDEPT TABLES  2000-ADDDEPT  * 00018000
* IF ACTION FIELD ADD:  * 00018100
* DETERMINE PROCESSING REQUEST  * 00018200
* ELSE RESET TO LOCAL LOCATION  * 00018300
* CONNECT TO REMOTE LOCATION  * 00018400
* IF CURRENT AND PREVIOUS LOCATION DIFFER THEN  * 00018500
* GET PANEL INPUT  1000-MAIN-LOOP  * 00018600
* DO UNTIL NO MORE TERMINAL INPUT  * 00018700
* SET PREVIOUS LOCATION TO LOCAL  * 00018800
* SET UP RETURN CODE HANDLING  0000-PROGRAM-START* 00018900
* Chapter 12. Sample data and applications supplied with Db2 for z/OS 1063

*PSEUDOCODE*
* END-DO UNTIL NO MORE TERMINAL INPUT
* RELEASE ALL CONNECTIONS

ENVIRONMENT DIVISION.
*---------------------------------------------------------------*
* ISPF DIALOG VARIABLE NAMES                                   * 00025700
*---------------------------------------------------------------*
* EXEC SQL INCLUDE SQLCA END-EXEC.                            00025800

FILE SECTION.                                                  00023200
DATA DIVISION.                                                  00022900
SELECT MSGOUT ASSIGN TO UT-S-SYSPRINT.                        00022800
FILE-CONTROL.                                                  00022700
INPUT-OUTPUT SECTION.                                          00022600
*------------------------                                         00022500
ENVIRONMENT DIVISION.                                          00022400
*---------------------------------------------------------------* 00022300
*   RELEASE ALL CONNECTIONS                                     * 00022200
*   END-DO UNTIL NO MORE TERMINAL INPUT                         * 00022100
Chapter 12. Sample data and applications supplied with Db2 for z/OS 1065
* SELECT DEPARTMENT VARIABLES
  03 SD-VAR-STG PIC 9(06) COMP VALUE 01. 00038500
  03 DN3-VAR-STG PIC 9(06) COMP VALUE 03. 00038600
  03 DNAME3-VAR-STG PIC 9(06) COMP VALUE 36. 00038900
  03 DMGR3-VAR-STG PIC 9(06) COMP VALUE 06. 00039100
  03 MGRN-VAR-STG PIC 9(06) COMP VALUE 18. 00039400
* SELECT EMPLOYEE VARIABLES
*  03 SEM-VAR-STG PIC 9(06) COMP VALUE 01. 00039500
  03 EN4-VAR-STG PIC 9(06) COMP VALUE 06. 00039600
  03 EMPN-VAR-STG PIC 9(06) COMP VALUE 17. 00039700
  03 DN4-VAR-STG PIC 9(06) COMP VALUE 03. 00039800
  03 DNAME4-VAR-STG PIC 9(06) COMP VALUE 36. 00039900
*  03 MSGS-VAR-STG PIC 9(06) COMP VALUE 79. 00040000
* ISPF DIALOG SERVICES DECLARATIONS
  01 I-VDEFINE PIC X(08) VALUE 'VDEFINE'. 00040500
  01 I-VGET PIC X(08) VALUE 'VGET'. 00040600
  01 I-VPUT PIC X(08) VALUE 'VPUT'. 00040700
  01 I-DISPLAY PIC X(08) VALUE 'DISPLAY'. 00040800
  01 I-TBISPL PIC X(08) VALUE 'TBISPL'. 00040900
  01 I-TBTOP PIC X(08) VALUE 'TBTOP'. 00041000
  01 I-TBCREATE PIC X(08) VALUE 'TBCREATE'. 00041100
  01 I-TBCLOSE PIC X(08) VALUE 'TBCLOSE'. 00041200
  01 I-TBADD PIC X(08) VALUE 'TBADD'. 00041300
  01 I-TRIVAL PIC X(08) VALUE 'TRIVAL'. 00041400
  01 I-TBOQUERY PIC X(08) VALUE 'TBOQUERY'. 00041500
* ISPF CALL MODIFIERS
  01 I-NOWRITE PIC X(08) VALUE 'NOWRITE'. 00041600
  01 I-REPLACE PIC X(08) VALUE 'REPLACE'. 00042000
  01 I-CHAR PIC X(08) VALUE 'CHAR'. 00042100
* ISPF PANEL NAMES
  01 SEL-PANEL PIC X(08) VALUE 'DSN8SSH1'. 00042200
  01 STR-PANEL PIC X(08) VALUE 'DSN8SSH1'. 00042300
  01 DEPT-PANEL PIC X(08) VALUE 'DSN8SSH2'. 00042400
  01 GEND-PANEL PIC X(08) VALUE 'DSN8SSH3'. 00042500
  01 GENE-PANEL PIC X(08) VALUE 'DSN8SSH4'. 00042600
  01 EMP-PANEL PIC X(08) VALUE 'DSN8SSH5'. 00043000
* LOCAL-VARIABLES
  01 LOCAL-VARIABLES. 00043300
  03 DATAW PIC X(36). 00043500
  03 GENDATA PIC X(36). 00043600
  03 SEL-DEPT PIC X(01). 00043700
  03 SEL-EMP PIC X(01). 00043800
  03 MGR-NAMe PIC X(18). 00043900
  03 EMP-NAMe PIC X(17). 00044000
  03 TOKEN PIC X(70). 00044100
  03 TEMPOLOC PIC X(16). 00044200
  03 PREVLOC PIC X(16). 00044300
  03 TEMPDEPT PIC X(03). 00044400
  03 CURDEPT PIC X(03). 00044500
  03 DELDEPT PIC X(03). 00044600
  03 STACKTOP PIC S9(04). 00044700
  03 DEPTPTR PIC S9(04). 00044800
  03 LISTSPT PIC S9(04). 00044900
  03 LOCPT PIC S9(04). 00045000
  03 LOCOT PIC S9(04). 00045100
  03 CONVSQL PIC S9(15) COMP-3. 00045200
  03 OUTMSG PIC X(69). 00045300
  03 TMSG REDEFINED OUTMSG.
  05 TMSGTXT PIC X(46). 00045400
  05 FILLER PIC X(23). 00045500
  03 MSGS PIC X(79) VALUE SPACES. 00045700
  03 MSGS-DETAIL REDEFINED MSGS.
  05 OUT-MESSAGE PIC X(46). 00045900
  05 SQL-CODE PIC X(04). 00046000
  05 FILLER PIC X(29). 00046100
  05 DEPTS OCCURS 1000 TIMES.
  03 DEPTS-ITEM PIC X(03). 00046600
Chapter 12. Sample data and applications supplied with Db2 for z/OS 1069
EXEC SQL DECLARE ALLEDEPT1 CURSOR FOR
SELECT DEPTNO, DEPTNAME, MGRNO, ADMINDEPT, LOCATION,
       SUBSTR(FIRSTNME, 1, 1) || MIDINIT || ' ' || LASTNAME
FROM VHEDEPT, VEMP
WHERE MGRNO = EMPNO
      AND DEPTNAME LIKE :GENDATA
ORDER BY 1
AND MGRNO IS NULL
FROM VHEDEPT
WHERE MGRNO IS NULL
      AND DEPTNAME LIKE :GENDATA
ORDER BY 1
EXEC SQL DECLARE ALLEDEPT2 CURSOR FOR
SELECT DEPTNO, DEPTNAME, MGRNO,
       SUBSTR(FIRSTNME, 1, 1) || MIDINIT || ' ' || LASTNAME
FROM VHEDEPT, VEMP
WHERE MGRNO = EMPNO
      AND DEPTNAME LIKE :GENDATA
ORDER BY 1
AND MGRNO IS NULL
FROM VHEDEPT
WHERE MGRNO IS NULL
      AND DEPTNAME LIKE :GENDATA
ORDER BY 1
EXEC SQL DECLARE ALLEDEPT3 CURSOR FOR
SELECT DEPTNO, DEPTNAME, MGRNO,
       SUBSTR(FIRSTNME, 1, 1) || MIDINIT || ' ' || LASTNAME
FROM VHEDEPT, VEMP
WHERE MGRNO = EMPNO
      AND DEPTNAME LIKE :GENDATA
ORDER BY 1
AND MGRNO IS NULL
FROM VHEDEPT
WHERE MGRNO IS NULL
      AND DEPTNAME LIKE :GENDATA
ORDER BY 1
EXEC SQL DECLARE ALLEDEPT4 CURSOR FOR
SELECT DEPTNO, DEPTNAME, MGRNO,
       SUBSTR(FIRSTNME, 1, 1) || MIDINIT || ' ' || LASTNAME
FROM VHEDEPT, VEMP
WHERE MGRNO = EMPNO
      AND LASTNAME LIKE :GENDATA
ORDER BY 1
EXEC SQL DECLARE ALLEDEPT5 CURSOR FOR
SELECT DEPTNO, DEPTNAME, MGRNO,
       SUBSTR(FIRSTNME, 1, 1) || MIDINIT || ' ' || LASTNAME
FROM VHEDEPT, VEMP
WHERE MGRNO = EMPNO
      AND DEPTNAME = :GENDATA
ORDER BY 1
EXEC SQL DECLARE ALLEDEPT6 CURSOR FOR
SELECT DEPTNO, DEPTNAME, MGRNO,
       SUBSTR(FIRSTNME, 1, 1) || MIDINIT || ' ' || LASTNAME
FROM VHEDEPT, VEMP
WHERE MGRNO = EMPNO
      AND DEPTNAME = :GENDATA
ORDER BY 1
WHERE MGRNO IS NULL
  AND MGRNO = :GENDATA
END-EXEC.
* EXEC SQL DECLARE ALLDEPT7 CURSOR FOR
  SELECT DEPTNO, DEPTNAME, MGRNO,
  SUBSTR(FIRSTNME, 1, 1) || MIDINIT || ' ' || LASTNAME
  FROM VHDEPT, VEMP
  WHERE MGRNO = EMPNO
  AND LASTNAME = :GENDATA
  ORDER BY 1
END-EXEC.
* EXEC SQL DECLARE EMPI CURSOR FOR
  SELECT DEPTNO, DEPTNAME, MGRNO, ADMRDEPT, LOCATION,
  EMPNO, FIRSTNME, MIDINIT, LASTNAME
  FROM VHDEPT, VEMP
  WHERE EMPNO = :DATAW
  AND WORKDEPT = DEPTNO
  UNION
  SELECT ' ', ' ', ' ', ' ', ' ',
  EMPNO, FIRSTNME, MIDINIT, LASTNAME, ' ' FROM VEMP
  WHERE EMPNO = :DATAW
  AND WORKDEPT IS NULL
END-EXEC.
* EXEC SQL DECLARE ALLEMP1 CURSOR FOR
  SELECT EMPNO, SUBSTR(FIRSTNME, 1, 1) || ' ' || LASTNAME,
  WORKDEPT, DEPTNAME
  FROM VHDEPT, VEMP
  WHERE DEPTNO = WORKDEPT
  AND EMPNO LIKE :GENDATA
  UNION
  SELECT EMPNO, SUBSTR(FIRSTNME, 1, 1) || ' ' || LASTNAME,
  WORKDEPT, ' ' FROM VEMP
  WHERE EMPNO LIKE :GENDATA
  AND WORKDEPT IS NULL
  ORDER BY 1
END-EXEC.
* EXEC SQL DECLARE ALLEMP2 CURSOR FOR
  SELECT EMPNO, SUBSTR(FIRSTNME, 1, 1) || ' ' || LASTNAME,
  WORKDEPT, DEPTNAME
  FROM VHDEPT, VEMP
  WHERE DEPTNO = WORKDEPT
  AND LASTNAME LIKE :GENDATA
  ORDER BY 1
END-EXEC.
* EXEC SQL DECLARE ALLEMP3 CURSOR FOR
  SELECT EMPNO, SUBSTR(FIRSTNME, 1, 1) || ' ' || LASTNAME,
  WORKDEPT, DEPTNAME
  FROM VHDEPT, VEMP
  WHERE DEPTNO = WORKDEPT
  AND LASTNAME = :GENDATA
  ORDER BY 1
END-EXEC.
* EXEC SQL DECLARE DEPTSTR CURSOR FOR
  SELECT DEPTNO, DEPTNAME, MGRNO, ADMRDEPT, LOCATION,
  FIRSTNME, MIDINIT, LASTNAME
  FROM VHDEPT, VEMP
  WHERE ADMRDEPT = :DATAW
  AND MGRNO = EMPNO
  UNION
  SELECT DEPTNO, DEPTNAME, MGRNO, ADMRDEPT, LOCATION,
  ' ', ' ', ' ' FROM VHDEPT, VEMP
  WHERE ADMRDEPT IS NULL
  ORDER BY 1
END-EXEC.
FROM VHDEPT
WHERE ADMRDEPT = :DATAW
AND MGRNO IS NULL
ORDER BY 1

CALL 'ISPLINK' USING I-VDEFINE, EFN2-VAR, EMP-FIRST-NAME, 00095700
I-CHAR, EN2-VAR-STG. 00095600
CALL 'ISPLINK' USING I-VDEFINE, EN2-VAR, EMP-NUMB, 00095500
I-CHAR, DLOC-VAR-STG. 00095400
CALL 'ISPLINK' USING I-VDEFINE, DLOC-VAR, DEPT-LOC, 00095300
I-CHAR, DADM-VAR-STG. 00095200
CALL 'ISPLINK' USING I-VDEFINE, DADM-VAR, DEPT-ADMR, 00095100
I-CHAR, DMGR2-VAR-STG. 00095000
CALL 'ISPLINK' USING I-VDEFINE, DMGR2-VAR, DEPT-MGR, 00094900
I-CHAR, DNAME2-VAR-STG. 00094800
CALL 'ISPLINK' USING I-VDEFINE, DNAME2-VAR, DEPT-NAME, 00094700
I-CHAR, DN2-VAR-STG. 00094600
CALL 'ISPLINK' USING I-VDEFINE, DN1-VAR, DEPT-NUMB, 00094500
I-CHAR, ELN1M-VAR-STG. 00094400
CALL 'ISPLINK' USING I-VDEFINE, ELN1M-VAR, EMP1-LAST-NAME, 00094300
I-CHAR, EMI1M-VAR-STG. 00094200
CALL 'ISPLINK' USING I-VDEFINE, EMI1M-VAR, EMP1-MID-INIT, 00094100
I-CHAR, EFN1M-VAR-STG. 00094000
CALL 'ISPLINK' USING I-VDEFINE, EFN1M-VAR, EMP1-FIRST-NAME, 00093900
I-CHAR, DMGR1M-VAR-STG. 00093800
CALL 'ISPLINK' USING I-VDEFINE, DMGR1M-VAR, DEPT1-MGR, 00093700
I-CHAR, DNAME1M-VAR-STG. 00093600
CALL 'ISPLINK' USING I-VDEFINE, DNAME1M-VAR, DEPT1-NAME, 00093500
I-CHAR, DN1M-VAR-STG. 00093400

EXEC SQL WHENEVER SQLERROR GOTO L8000-P3-DBERROR END-EXEC.
EXEC SQL WHENEVER SQLWARNING GOTO L8000-P3-DBERROR END-EXEC.
EXEC SQL WHENEVER NOT FOUND CONTINUE END-EXEC.

* DEFIE COBOL - SPF VARIABLES

0000-PROGRAM-START.
CALL 'ISPLINK' USING I-VDEFINE, CH-VAR, ROWS-CHANGED,
I-CHAR, CH-VAR-STG. 00089700
CALL 'ISPLINK' USING I-VDEFINE, QROWS, NUMROWS,
I-CHAR, QROWS-STG. 00089990

* ACTION PANEL

CALL 'ISPLINK' USING I-VDEFINE, ACT-VAR, ACTION,
I-CHAR, AC-VAR-STG. 00090400
CALL 'ISPLINK' USING I-VDEFINE, OBJ-VAR, OBJFLD,
I-CHAR, OB-VAR-STG. 00090600
CALL 'ISPLINK' USING I-VDEFINE, SEA-VAR, SEARCH-CRIT,
I-CHAR, SE-VAR-STG. 00090800
CALL 'ISPLINK' USING I-VDEFINE, LOC-VAR, LOCATION,
I-CHAR, LO-VAR-STG. 00091000
CALL 'ISPLINK' USING I-VDEFINE, DAT-VAR, NAMEID,
I-CHAR, DT-VAR-STG. 00091200

* DEPARTMENT STRUCTURE PANEL

CALL 'ISPLINK' USING I-VDEFINE, DN1M-VAR, DEPT1-NUMB,
I-CHAR, DN1M-VAR-STG. 00091600
CALL 'ISPLINK' USING I-VDEFINE, DN1M-VAR, DEPT1-NAME,
I-CHAR, DN1M-VAR-STG. 00091700
CALL 'ISPLINK' USING I-VDEFINE, DMGR1M-VAR, DEPT1-MGR,
I-CHAR, DMGR1M-VAR-STG. 00092100
CALL 'ISPLINK' USING I-VDEFINE, EM11M-VAR, EMP1-MID-INIT,
I-CHAR, EM11M-VAR-STG. 00092300
CALL 'ISPLINK' USING I-VDEFINE, ELN1M-VAR, EMP1-LAST-NAME,
I-CHAR, ELN1M-VAR-STG. 00092600
CALL 'ISPLINK' USING I-VDEFINE, DN1-VAR, DEPT-NUMB,
I-CHAR, DN1-VAR-STG. 00092800
CALL 'ISPLINK' USING I-VDEFINE, DN1-VAR, DEPT-NAME,
I-CHAR, DN1-VAR-STG. 00093000
CALL 'ISPLINK' USING I-VDEFINE, DMGR1-VAR, DEPT-MGR,
I-CHAR, DMGR1-VAR-STG. 00093200
CALL 'ISPLINK' USING I-VDEFINE, EM11-VAR, EMP-MID-INIT,
I-CHAR, EM11-VAR-STG. 00093600
CALL 'ISPLINK' USING I-VDEFINE, ELN1-VAR, EMP-LAST-NAME,
I-CHAR, ELN1-VAR-STG. 00093800

* DISPLAY PANEL

CALL 'ISPLINK' USING I-VDEFINE, ACTL-VAR, ACTION-LIST,
I-CHAR, ACL-VAR-STG. 00094000
CALL 'ISPLINK' USING I-VDEFINE, DN2-VAR, DEPT-NUMB,
I-CHAR, DN2-VAR-STG. 00094500
CALL 'ISPLINK' USING I-VDEFINE, DN2-VAR, DEPT-NAME,
I-CHAR, DN2-VAR-STG. 00094600
CALL 'ISPLINK' USING I-VDEFINE, DMGR2-VAR, DEPT-MGR,
I-CHAR, DMGR2-VAR-STG. 00095000
CALL 'ISPLINK' USING I-VDEFINE, DAMD-VAR, DEPT-ADM,
I-CHAR, DAMD-VAR-STG. 00095100
CALL 'ISPLINK' USING I-VDEFINE, DLLOC-VAR, DEPT-LOC,
I-CHAR, DLLOC-VAR-STG. 00095300
CALL 'ISPLINK' USING I-VDEFINE, EN2-VAR, EMP-NUMB,
I-CHAR, EN2-VAR-STG. 00095500
CALL 'ISPLINK' USING I-VDEFINE, EN2-VAR, EMP-FIRST-NAME,
I-CHAR, EN2-VAR-STG. 00095700
CALL 'ISPLINK' USING I-VDEFINE, EN2-VAR, EMP-SECOND-NAME,
CALL 'ISPLINK' USING I-VDEFINE, EM12-VAR, EMP-MID-INIT. 000995900
I-CHAR, EM12-VAR-STG. 000996000
CALL 'ISPLINK' USING I-VDEFINE, ELN2-VAR, EMP-LAST-NAME. 000961000
I-CHAR, ELN2-VAR-STG. 000962000
CALL 'ISPLINK' USING I-VDEFINE, EW1-VAR, EMP-WORK-DEPT. 000963000
I-CHAR, EW1-VAR-STG.
* SELECT DEPARTMENT PANEL
* CALL 'ISPLINK' USING I-VDEFINE, SD-VAR, SDL-DEPT, 000968000
I-CHAR, SD-VAR-STG.
CALL 'ISPLINK' USING I-VDEFINE, DN3-VAR, DEPT-NUMB. 000970000
I-CHAR, DN3-VAR-STG.
CALL 'ISPLINK' USING I-VDEFINE, DNAME3-VAR, DEPT-NAME. 000972000
I-CHAR, DNAME3-VAR-STG.
CALL 'ISPLINK' USING I-VDEFINE, DMGR3-VAR, DEPT-MGR. 000973000
I-CHAR, DMGR3-VAR-STG.
CALL 'ISPLINK' USING I-VDEFINE, MGRN-VAR, MGR-NAME. 000974000
I-CHAR, MGRN-VAR-STG.
* SELECT EMPLOYEE PANEL
* CALL 'ISPLINK' USING I-VDEFINE, SEM-VAR, SEL-EMP. 000980000
I-CHAR, SEM-VAR-STG.
CALL 'ISPLINK' USING I-VDEFINE, EN4-VAR, EMP-NUMB. 000983000
I-CHAR, EN4-VAR-STG.
CALL 'ISPLINK' USING I-VDEFINE, EMPN-VAR, EMP-NAME. 000984000
I-CHAR, EMPN-VAR-STG.
CALL 'ISPLINK' USING I-VDEFINE, DN4-VAR, EMP-WORK-DEPT. 000986000
I-CHAR, DN4-VAR-STG.
CALL 'ISPLINK' USING I-VDEFINE, DNAME4-VAR, DEPT-NAME. 000987000
I-CHAR, DNAME4-VAR-STG.
* CALL 'ISPLINK' USING I-VDEFINE, MSGS-VAR, MSGS. 000992000
I-CHAR, MSGS-VAR-STG.
* MAIN PROGRAM
* MOVE 'N' TO SEL-EXIT. 000997000
MOVE SPACES TO PREVLOC.
PERFORM 1000-MAIN-LOOP THRU 1000-MAIN-LOOP-EXIT 001000000
UNTIL SEL-EXIT = 'Y'. 001001000
MOVE 0 TO RETURN-CODE.
MOVE SPACES TO MSGS.
CALL 'ISPLINK' USING I-VPUT, MSGS-VAR. 001004000
GOBACK.
* 1000-MAIN-LOOP.
CALL 'ISPLINK' USING I-DISPLAY, SEL-PANEL. 001007000
IF RETURN-CODE = 8 THEN
EXEC SQL COMMIT END-EXEC
EXEC SQL RELEASE ALL SQL END-EXEC
MOVE 'Y' TO SEL-EXIT.
ELSE
MOVE SPACES TO MSGS.
MOVE 'N' TO GEND-EXIT, GENE-EXIT
CALL 'ISPLINK' USING I-VGET, ACTION-VARS
MOVE NAMEID TO DATAW
MOVE 0 TO LENGTH-COUNTER
INSPECT DATAW
TALLYING LENGTH-COUNTER FOR CHARACTERS
BEFORE INITIAL SPACE
IF SEARCH-CRIT = 'DI' AND LENGTH-COUNTER > 3 OR
SEARCH-CRIT = 'MI' AND LENGTH-COUNTER > 6 OR
SEARCH-CRIT = 'EI' AND LENGTH-COUNTER > 6 THEN
MOVE '074E' TO MSGCODE
CALL 'DSN8MC8' USING MODULE, MSGCODE, OUTMSG
MOVE OUTMSG TO MSGS
ELSE
PERFORM 1100-CONNECT THRU 1100-CONNECT-EXIT
PERFORM 1200-DOACTION THRU 1200-DOACTION-EXIT.
1000-MAIN-LOOP-EXIT. 001031000
EXIT.
* CONNECT TO NEW LOCATION
* 1100-CONNECT.
IF LOCATION NOT EQUAL TO PREVLOC THEN
MOVE LOCATION TO PREVLOC
IF LOCATION NOT EQUAL TO SPACES THEN

EXEC SQL CONNECT TO :LOCATION END-EXEC.
ELSE
EXEC SQL CONNECT RESET END-EXEC.
EXIT.
*
* DETERMINE PROCESSING REQUEST
*
* ADD A DEPARTMENT
*
2000-ADDDEPT.
CALL 'ISPLINK' USING I-VPUT, IDEN-VAR.
PERFORM 2100-DISDEPTDATA THRU 2100-DISDEPTDATA-EXIT.
CALL 'ISPLINK' USING I-VPUT, ADD-DEPT-VARS.
CALL 'ISPLINK' USING I-DISPLAY, DEPT-PANEL.
IF RETURN-CODE NOT EQUAL TO 8 THEN
EXEC SQL WHENEVER SQLERROR CONTINUE END-EXEC
EXEC SQL INSERT INTO VHDEPT
END-EXEC
PERFORM 2200-ADDEPTCODES THRU 2200-ADDEPTCODES-EXIT
EXEC SQL WHENEVER SQLERROR GOTO L8000-P3-DBERROR END-EXEC
PERFORM 2300-GETEMPREC THRU 2300-GETEMPREC-EXIT
CALL 'ISPLINK' USING I-DISPLAY, DEPT-PANEL.
EXIT.
*
* DISPLAY INPUT DATA ON PANEL
*
2100-DISDEPTDATA.
MOVE SPACES TO DEPT-RECORD.
IF SEARCH-CRIT = 'DI' THEN
MOVE DATAW TO DEPT-RECORD.
ELSE
IF SEARCH-CRIT = 'DN' THEN
MOVE DATAW TO DEPT-MGR.
ELSE
IF SEARCH-CRIT = 'MT' THEN
MOVE DATAW TO DEPT-MGR.
EXIT.
*
* CHECK RETURN CODE FROM INSERT. IF OK, ADD TO OTHER LOCATIONS.*
2200-ADDEPTCODES.
IF SQLERRP = SPACES THEN
EXEC SQL WHENEVER SQLERROR GOTO L8000-P3-DBERROR END-EXEC
EXEC SQL CONNECT TO :LOCATION END-EXEC.
ELSE
EXEC SQL CONNECT RESET END-EXEC.
EXIT.
*
INTO TOKEN

IF TOKEN = 'RDD' THEN
    MOVE '213E' TO MSGCODE
    CALL 'DSN8MCG' USING MODULE, MSGCODE, OUTMSG
    MOVE OUTMSG TO MSGS
ELSE
    IF TOKEN = 'RDE' THEN
        MOVE '210E' TO MSGCODE
        CALL 'DSN8MCG' USING MODULE, MSGCODE, OUTMSG
        MOVE OUTMSG TO MSGS
    ELSE
        GO TO L8000-P3-DBERROR
    END-EXEC.
ELSE
    IF SQLCODE NOT EQUAL TO 0 THEN
        GO TO L8000-P3-DBERROR
    ELSE
        EXEC SQL OPEN LOCS END-EXEC
        MOVE 0 TO LOCPTR
        PERFORM 2210-BUILDLOCTABLE THRU 2210-BUILDLOCTABLE-EXIT
        UNTIL SQLCODE NOT EQUAL TO 0
        EXEC SQL CLOSE LOCS END-EXEC
        MOVE LOCPTR TO LOCTOP
        MOVE 0 TO LOCPTR
        PERFORM 2220-ADDLOCS THRU 2220-ADDLOCS-EXIT UNTIL LOCPTR = LOCTOP
        MOVE '012I' TO MSGCODE
        CALL 'DSN8MCG' USING MODULE, MSGCODE, OUTMSG
        MOVE OUTMSG TO MSGS
        MOVE DEPT-LOC TO LOCATION
        PERFORM 1100-CONNECT THRU 1100-CONNECT-EXIT.
        2200-ADDDEPTCODES-EXIT.
        EXIT.
*---------------------------------------------------------------*
* BUILD TABLE OF UNIQUE LOCATIONS IN VHDEPT
*---------------------------------------------------------------*
2210-BUILDLOCTABLE.
    EXEC SQL FETCH LOCS INTO :TEMPLOC END-EXEC.
    IF SQLCODE = 0 THEN
        ADD 1 TO LOCPTR
        MOVE TEMPLOC TO LOCLIST (LOCPTR).
    2210-BUILDLOCTABLE-EXIT.
    EXIT.
*---------------------------------------------------------------*
* ADD NEW DEPARTMENT TO VHDEPT VIEWS AT ALL LOCATIONS
*---------------------------------------------------------------*
2220-ADDLOCS.
    IF LOCPTR < LOCTOP THEN
        ADD 1 TO LOCPTR
        MOVE LOCLIST (LOCPTR) TO TEMPLOC
    EXEC SQL CONNECT TO :TEMPLOC END-EXEC
    END-EXEC.
2220-ADDLOCS-EXIT.
    EXIT.
*---------------------------------------------------------------*
* RETRIEVE MANAGER INFO FOR NEW DEPARTMENT
*---------------------------------------------------------------*
2300-GETEMPREC.
    CALL 'ISPLINK' USING I-VGET, ADD-DEPT-VARS.
    EXEC SQL SELECT *
    FROM VEMP
    WHERE EMPNO = :DEPT-MGR
    END-EXEC.
    IF SQLCODE = 100 THEN
        MOVE SPACES TO EMP-RECORD.
    2300-GETEMPREC-EXIT.
    EXIT.
*---------------------------------------------------------------*
* ADD AN EMPLOYEE
*---------------------------------------------------------------*
EXEC SQL CLOSE CURDEPTLOC END-EXEC.
EXEC SQL FETCH DEPTLOC INTO :DEPT-LOC END-EXEC.
EXEC SQL OPEN DEPTLOC END-EXEC.

3000-ADDEMP.
CALL 'ISPLINK' USING I-VGET, IDEN-VAR.
PERFORM 3100-DISEMPDATA THRU 3100-DISEMPDATA-EXIT.
CALL 'ISPLINK' USING I-VPUT, ADD-EMP-VARS.
EXEC SQL WHENEVER SQLERROR CONTINUE END-EXEC.
EXEC SQL OPEN CURDEPTLOC END-EXEC.
PERFORM 3300-GETDEPTREC THRU 3300-GETDEPTREC-EXIT.
EXEC SQL OPEN DEPTLOC END-EXEC.
EXEC SQL FETCH DEPTLOC INTO :DEPT-LOC END-EXEC.
EXEC SQL CLOSE CURDEPTLOC END-EXEC.

3000-ADDEMP-EXIT.
EXEC SQL WHENEVER SQLERROR GOTO L8000-P3-DBERROR.
EXEC SQL OPEN DEPTLOC END-EXEC.
EXEC SQL FETCH DEPTLOC INTO :DEPT-LOC END-EXEC.
EXEC SQL OPEN CURDEPTLOC END-EXEC.
EXEC SQL CLOSE CURDEPTLOC END-EXEC.

DISPLAY INPUT DATA ON PANEL.

3100-DISEMPDATA.
MOVE SPACES TO DEPT-RECORD.
MOVE SPACES TO EMP-RECORD.
IF SEARCH-CRIT = 'EI' THEN
  MOVE SPACES TO EMP-RECORD.
ELSE
  MOVE SPACES TO EMP-RECORD.
ENDIF.

3100-DISEMPDATA-EXIT.

3200-ADDEMPCODES.
IF SQLERRP = SPACES THEN
  MOVE '079E' TO SQLERRP.
ELSE
  IF SQLCODE = -803 THEN
    MOVE '095E' TO SQLERRP.
  ELSE
    IF SQLCODE = -530 THEN
      MOVE '200E' TO SQLERRP.
    ELSE
      IF SQLCODE = 0 THEN
        MOVE '092I' TO SQLERRP.
      ELSE
        GO TO L8000-P3-DBERROR.
      END-EXEC.
    END-EXEC.
  END-EXEC.
ENDIF.

3200-ADDEMPCODES-EXIT.

3300-GETDEPTREC.
CALL 'ISPLINK' USING I-VGET, ADD-EMP-VARS.
WHERE DEPTNO = :EMP-WORK-DEPT.
END-EXEC.
IF SQLCODE = 100 THEN
MOVE SPACES TO DEPT-RECORD
ELSE
PERFORM 3310-CHECKDEPTIND THRU 3310-CHECKDEPTIND-EXIT.
3300-GETDEPTREC-EXIT.
EXIT.
*
* IF MGRNO NULL, MOVE BLANKS INTO FIELD
* 3310-CHECKDEPTIND.
* IF DEPT-MGR-IND < 0 THEN
* MOVE SPACES TO DEPT-MGR.
3310-CHECKDEPTIND-EXIT.
EXIT.
*
* SET LOCATION TO CURRENT SERVER
3320-SETURLOC.
IF LOCATION EQUAL TO SPACES THEN
EXEC SQL FETCH CURDEPTLOC
INTO :LOCATION
END-EXEC.
3320-SETURLOC-EXIT.
EXIT.
*
* MOVE APPROPRIATE ACTION INTO ACTION-LIST
4000-ACTION.
IF ACTION = 'E' THEN
MOVE ' ERASE' TO ACTION-LIST
ELSE
IF ACTION = 'U' THEN
MOVE ' UPDATE' TO ACTION-LIST
ELSE
MOVE 'DISPLAY' TO ACTION-LIST.
MOVE 0 TO PERCENT-COUNTER.
INSPECT DATAW
TALLYING PERCENT-COUNTER FOR ALL '%'.
IF PERCENT-COUNTER > 0 THEN
INSPECT DATAW
REPLACING ALL " " BY '%'.
4000-ACTION-EXIT.
EXIT.
*
* PERFORM ACTION ON DEPARTMENT OR DEPARTMENT STRUCTURE
5000-DEPARTMENT.
IF NOT (SEARCH-CRIT = 'DI' AND PERCENT-COUNTER = 0) THEN
MOVE DATAN TO GENDATA
PERFORM 5100-GENDEPT THRU 5100-GENDEPT-EXIT
UNTIL GEND-EXIT = 'Y'
ELSE
IF OBJFLD = 'DE' THEN
PERFORM 5200-DISPLAYDEPT THRU 5200-DISPLAYDEPT-EXIT
ELSE
PERFORM 5300-STRUCTURE THRU 5300-STRUCTURE-EXIT.
5000-DEPARTMENT-EXIT.
EXIT.
*
* GENERIC LIST OF DEPARTMENTS
5100-GENDEPT.
CALL 'ISPLINK' USING I-TBCREATE, DEPT-TABLE, W-BLANK,
SEL-DEPT-VARS, I-NOWRITE, I-REPLACE.
MOVE SPACE TO SEL-DEPT.
PERFORM 5110-GETDEPTTAB THRU 5110-GETDEPTTAB-EXIT.
CALL 'ISPLINK' USING I-TBQUERY, DEPT-TABLE, W-BLANK,
W-BLANK, QROWS.
IF NUMROWS = 1 AND GENDATA = DATAN THEN
MOVE 'Y' TO SPECIAL-EXIT
CALL 'ISPLINK' USING I-TBGET, DEPT-TABLE.
MOVE DEPT-NUMB TO DATAN
ELSE
MOVE 'N' TO SPECIAL-EXIT
IF NUMROWS = 0 THEN
PERFORM 5120-DEPTMSG THRU 5120-DEPTMSG-EXIT
MOVE 'Y' TO GEND-EXIT
ASDF
ELSE
CALL 'ISPLINK' USING I-VPUT, ACTL-VAR
CALL 'ISPLINK' USING I-TBTOP, DEPT-TABLE,
GEND-PANEL
IF RETURN-CODE = 8 THEN
MOVE 'Y' TO GEND-EXIT
ELSE
IF ROWS-CHANGED > 0 THEN
CALL 'ISPLINK' USING I-TBGET, DEPT-TABLE
MOVE DEPT-NUMB TO DATAW
ELSE
MOVE 'Y' TO GEND-EXIT.
IF GEND-EXIT = 'N' THEN
IF OBJFLD = 'DE' THEN
PERFORM 5200-DISPLAYDEPT THRU 5200-DISPLAYDEPT-EXIT
ELSE
PERFORM 5300-STRUCTURE THRU 5300-STRUCTURE-EXIT.
END-EXEC.
IF SPECIAL-EXIT = 'Y' THEN
MOVE 'Y' TO GEND-EXIT.
CALL 'ISPLINK' USING I-TBCLOSE, DEPT-TABLE.
5100-GENDDEPT-EXIT.
EXIT.
**-------------------------------------------*
** CREATE TABLE OF DEPARTMENTS TO FIT SEARCH-CRIT          *
**-----------------------------------------------------------*
5110-GETDEPTTAB
IF SEARCH-CRIT = 'DI' THEN
EXEC SQL OPEN ALLDEPT1 END-EXEC
MOVE SPACES TO SQLERRP
PERFORM 5111-ALLDEPT1 THRU 5111-ALLDEPT1-EXIT
UNTIL SQLCODE NOT EQUAL TO 0 OR GEND-EXIT = 'Y'
EXEC SQL CLOSE ALLDEPT1 END-EXEC
ELSE
IF SEARCH-CRIT = 'DN' AND PERCENT-COUNTER > 0 THEN
EXEC SQL OPEN ALLDEPT2 END-EXEC
MOVE SPACES TO SQLERRP
PERFORM 5112-ALLDEPT2 THRU 5112-ALLDEPT2-EXIT
UNTIL SQLCODE NOT EQUAL TO 0 OR GEND-EXIT = 'Y'
EXEC SQL CLOSE ALLDEPT2 END-EXEC
ELSE
IF SEARCH-CRIT = 'DN' THEN
EXEC SQL OPEN ALLDEPT5 END-EXEC
MOVE SPACES TO SQLERRP
PERFORM 5113-ALLDEPT5 THRU 5113-ALLDEPT5-EXIT
UNTIL SQLCODE NOT EQUAL TO 0 OR GEND-EXIT = 'Y'
EXEC SQL CLOSE ALLDEPT5 END-EXEC
ELSE
IF SEARCH-CRIT = 'MI' AND
PERCENT-COUNTER > 0 THEN
EXEC SQL OPEN ALLDEPT3 END-EXEC
MOVE SPACES TO SQLERRP
PERFORM 5114-ALLDEPT3 THRU 5114-ALLDEPT3-EXIT
UNTIL SQLCODE NOT EQUAL TO 0 OR GEND-EXIT = 'Y'
EXEC SQL CLOSE ALLDEPT3 END-EXEC
ELSE
IF SEARCH-CRIT = 'MI' THEN
EXEC SQL OPEN ALLDEPT6 END-EXEC
MOVE SPACES TO SQLERRP
PERFORM 5115-ALLDEPT6 THRU 5115-ALLDEPT6-EXIT
UNTIL SQLCODE NOT EQUAL TO 0 OR GEND-EXIT = 'Y'
EXEC SQL CLOSE ALLDEPT6 END-EXEC
ELSE
IF SEARCH-CRIT = 'MN' AND
PERCENT-COUNTER > 0 THEN
EXEC SQL OPEN ALLDEPT4 END-EXEC
MOVE SPACES TO SQLERRP
PERFORM 5116-ALLDEPT4 THRU 5116-ALLDEPT4-EXIT
UNTIL SQLCODE NOT EQUAL TO 0 OR GEND-EXIT = 'Y'
EXEC SQL CLOSE ALLDEPT4 END-EXEC
ELSE
IF SEARCH-CRIT = 'MN' THEN
EXEC SQL OPEN ALLDEPT7 END-EXEC
MOVE SPACES TO SQLERRP
EXEC SQL FETCH ALLDEPT7
TO 0 OR GEND-EXIT = 'Y'
EXEC SQL CLOSE ALLDEPT7
END-EXEC.

5110-GETDEPTTAB-EXIT.
EXIT.*

5111-ALLDEPT1.
EXEC SQL FETCH ALLDEPT1
   INTO :DEPT-NUMB, :DEPT-NAME,
      :DEPT-MGR:DEPT-MGR-IND,
      :MGR-NAME
END-EXEC.
IF SQLERRP = SPACES THEN
   MOVE '079E' TO MSGCODE
   MOVE 'Y' TO GEND-EXIT
ELSE
   IF SQLCODE = 0 THEN
      PERFORM 3310-CHECKDEPTIND THRU
      3310-CHECKDEPTIND-EXIT
      CALL 'ISPLINK' USING I-TBADD, DEPT-TABLE.
   5111-ALLDEPT1-EXIT.
   EXIT.*

5112-ALLDEPT2.
EXEC SQL FETCH ALLDEPT2
   INTO :DEPT-NUMB, :DEPT-NAME,
      :DEPT-MGR:DEPT-MGR-IND,
      :MGR-NAME
END-EXEC.
IF SQLERRP = SPACES THEN
   MOVE '079E' TO MSGCODE
   MOVE 'Y' TO GEND-EXIT
ELSE
   IF SQLCODE = 0 THEN
      PERFORM 3310-CHECKDEPTIND THRU
      3310-CHECKDEPTIND-EXIT
      CALL 'ISPLINK' USING I-TBADD, DEPT-TABLE.
   5112-ALLDEPT2-EXIT.
   EXIT.*

5113-ALLDEPT5.
EXEC SQL FETCH ALLDEPT5
   INTO :DEPT-NUMB, :DEPT-NAME,
      :DEPT-MGR:DEPT-MGR-IND,
      :MGR-NAME
END-EXEC.
IF SQLERRP = SPACES THEN
   MOVE '079E' TO MSGCODE
   MOVE 'Y' TO GEND-EXIT
ELSE
   IF SQLCODE = 0 THEN
      PERFORM 3310-CHECKDEPTIND THRU
      3310-CHECKDEPTIND-EXIT
      CALL 'ISPLINK' USING I-TBADD, DEPT-TABLE.
   5113-ALLDEPT5-EXIT.
   EXIT.*

5114-ALLDEPT3.
EXEC SQL FETCH ALLDEPT3
   INTO :DEPT-NUMB, :DEPT-NAME,
      :DEPT-MGR:DEPT-MGR-IND,
      :MGR-NAME
END-EXEC.
IF SQLERRP = SPACES THEN
   MOVE '079E' TO MSGCODE
   MOVE 'Y' TO GEND-EXIT
ELSE
   IF SQLCODE = 0 THEN
      PERFORM 3310-CHECKDEPTIND THRU
      3310-CHECKDEPTIND-EXIT
      CALL 'ISPLINK' USING I-TBADD, DEPT-TABLE.
   5114-ALLDEPT3-EXIT.
   EXIT.*

5115-ALLDEPT6.
EXEC SQL FETCH ALLDEPT6
   INTO :DEPT-NUMB, :DEPT-NAME,
      :DEPT-MGR:DEPT-MGR-IND,
      :MGR-NAME
   EXIT.
*---------------------------------------------------------------*
* DISPLAY A DEPARTMENT                                          *
*---------------------------------------------------------------*
EXIT. 00154100
5115-ALLDEPT6-EXIT. 00154200
EXEC SQL FETCH DEPT1 INTO :DEPT-NUMB, :DEPT-NAME, 00154300
EXEC SQL OPEN DEPT1 END-EXEC. 00154400
MOVE SPACES TO EMP-RECORD. 00154500
MOVE SPACES TO DEPT-RECORD. 00154600
EXEC SQL FETCH DEPT1 INTO :DEPT-NUMB, :DEPT-NAME, 00154700
EXEC SQL OPEN DEPT1 END-EXEC. 00154800
PERFORM 5210-DISDEPTACT THRU 5210-DISDEPTACT-EXIT. 00154900
END-EXEC. 00155000
*---------------------------------------------------------------*
* PRINT CORRECT 'DEPARTMENT NOT FOUND' MESSAGE                  *
*---------------------------------------------------------------*
EXIT. 00155100
5116-ALLDEPT4. 00155200
EXEC SQL FETCH ALLDEPT4 00155300
INTO :DEPT-NUMB, :DEPT-NAME, 00155400
EXEC SQL OPEN DEPT4 END-EXEC. 00155500
PERFORM 3310-CHECKDEPTIND THRU 3310-CHECKDEPTIND-EXIT 00155600
CALL 'ISPLINK' USING I-TBADD, DEPT-TABLE. 00155700
EXEC SQL FETCH ALLDEPT7 00155800
EXEC SQL OPEN DEPT7 END-EXEC. 00155900
PERFORM 3310-CHECKDEPTIND THRU 3310-CHECKDEPTIND-EXIT 00156000
CALL 'ISPLINK' USING I-TBADD, DEPT-TABLE. 00156100
EXEC SQL FETCH ALLDEPT4 00156200
EXEC SQL OPEN DEPT4 END-EXEC. 00156300
PERFORM 3310-CHECKDEPTIND THRU 3310-CHECKDEPTIND-EXIT 00156400
CALL 'ISPLINK' USING I-TBADD, DEPT-TABLE. 00156500
EXEC SQL FETCH ALLDEPT6 00156600
EXEC SQL OPEN DEPT6 END-EXEC. 00156700
PERFORM 3310-CHECKDEPTIND THRU 3310-CHECKDEPTIND-EXIT 00156800
CALL 'ISPLINK' USING I-TBADD, DEPT-TABLE. 00156900
EXEC SQL FETCH ALLDEPT4 00157000
EXEC SQL OPEN DEPT4 END-EXEC. 00157100
PERFORM 3310-CHECKDEPTIND THRU 3310-CHECKDEPTIND-EXIT 00157200
CALL 'ISPLINK' USING I-TBADD, DEPT-TABLE. 00157300
EXEC SQL FETCH ALLDEPT7 00157400
EXEC SQL OPEN DEPT7 END-EXEC. 00157500
PERFORM 3310-CHECKDEPTIND THRU 3310-CHECKDEPTIND-EXIT 00157600
CALL 'ISPLINK' USING I-TBADD, DEPT-TABLE. 00157700
EXEC SQL FETCH ALLDEPT4 00157800
EXEC SQL OPEN DEPT4 END-EXEC. 00157900
PERFORM 3310-CHECKDEPTIND THRU 3310-CHECKDEPTIND-EXIT 00158000
CALL 'ISPLINK' USING I-TBADD, DEPT-TABLE. 00158100
EXEC SQL FETCH ALLDEPT6 00158200
EXEC SQL OPEN DEPT6 END-EXEC. 00158300
PERFORM 3310-CHECKDEPTIND THRU 3310-CHECKDEPTIND-EXIT 00158400
CALL 'ISPLINK' USING I-TBADD, DEPT-TABLE. 00158500
EXEC SQL FETCH ALLDEPT4 00158600
EXEC SQL OPEN DEPT4 END-EXEC. 00158700
PERFORM 3310-CHECKDEPTIND THRU 3310-CHECKDEPTIND-EXIT 00158800
CALL 'ISPLINK' USING I-TBADD, DEPT-TABLE. 00158900
EXEC SQL FETCH ALLDEPT7 00159000
EXEC SQL OPEN DEPT7 END-EXEC. 00159100
PERFORM 3310-CHECKDEPTIND THRU 3310-CHECKDEPTIND-EXIT 00159200
CALL 'ISPLINK' USING I-TBADD, DEPT-TABLE. 00159300
EXEC SQL FETCH ALLDEPT4 00159400
EXEC SQL OPEN DEPT4 END-EXEC. 00159500
PERFORM 3310-CHECKDEPTIND THRU 3310-CHECKDEPTIND-EXIT 00159600
CALL 'ISPLINK' USING I-TBADD, DEPT-TABLE. 00159700
EXEC SQL FETCH ALLDEPT6 00159800
EXEC SQL OPEN DEPT6 END-EXEC. 00159900
PERFORM 3310-CHECKDEPTIND THRU 3310-CHECKDEPTIND-EXIT 00160000
CALL 'ISPLINK' USING I-TBADD, DEPT-TABLE. 00160100
EXEC SQL FETCH ALLDEPT4 00160200
EXEC SQL OPEN DEPT4 END-EXEC. 00160300
PERFORM 3310-CHECKDEPTIND THRU 3310-CHECKDEPTIND-EXIT 00160400
CALL 'ISPLINK' USING I-TBADD, DEPT-TABLE. 00160500
EXEC SQL FETCH ALLDEPT7 00160600
EXEC SQL OPEN DEPT7 END-EXEC. 00160700
PERFORM 3310-CHECKDEPTIND THRU 3310-CHECKDEPTIND-EXIT 00160800
CALL 'ISPLINK' USING I-TBADD, DEPT-TABLE. 00160900
EXEC SQL FETCH ALLDEPT4 00161000
EXEC SQL OPEN DEPT4 END-EXEC. 00161100
PERFORM 3310-CHECKDEPTIND THRU 3310-CHECKDEPTIND-EXIT 00161200
EXEC SQL FETCH ALLDEPT6 00161300
EXEC SQL OPEN DEPT6 END-EXEC. 00161400
5210-DISDEPTACT.
  IF SQLERRP = SPACES THEN
    EXEC SQL CLOSE DEPT1 END-EXEC
    MOVE '013I' TO MSGCODE
    CALL 'DSN8MCG' USING MODULE, MSGCODE, OUTMSG
    EXEC SQL CLOSE DEPT1 END-EXEC
  ELSE
    EXEC SQL CLOSE DEPT1 END-EXEC
    PERFORM 5210-DELDEPTS THRU 5210-DELDEPTS-EXIT
    IF SQLCODE NOT EQUAL TO 0 THEN
      CALL 'ISPLINK' USING I-DISPLAY, DEPT-PANEL
      IF RETURN-CODE NOT EQUAL TO 8 THEN
        PERFORM 5220-ERASEDEPT THRU
        5222-DELETEDEPTACT-EXIT
      ELSE
        PERFORM 5220-ERASEDEPT THRU
        5222-DELETEDEPTACT-EXIT
        IF ACTION = 'E' THEN
          PERFORM 5220-ERASEDEPT THRU
          5222-DELETEDEPTACT-EXIT
        ELSE
          PERFORM 5220-ERASEDEPT THRU
          5222-DELETEDEPTACT-EXIT
          IF ACTION = 'U' THEN
            PERFORM 5220-ERASEDEPT THRU
            5222-DELETEDEPTACT-EXIT
        END-IF
      END-IF
    END-IF
  END-IF
EXEC SQL CLOSE DEPT1 END-EXEC
EXIT.
  *---------------------------------------------------------------*
  * DISPLAY, ERASE, OR UPDATE DEPARTMENT                          *
  *---------------------------------------------------------------*
5210-DISDEPTACT. 00161500
  IF SQLERRP = SPACES THEN
    EXEC SQL CLOSE DEPT1 END-EXEC
    MOVE '013I' TO MSGCODE
    CALL 'DSN8MCG' USING MODULE, MSGCODE, OUTMSG
    EXEC SQL CLOSE DEPT1 END-EXEC
  ELSE
    EXEC SQL CLOSE DEPT1 END-EXEC
    PERFORM 5210-DELDEPTS THRU 5210-DELDEPTS-EXIT
    IF SQLCODE NOT EQUAL TO 0 THEN
      CALL 'ISPLINK' USING I-DISPLAY, DEPT-PANEL
      IF RETURN-CODE NOT EQUAL TO 8 THEN
        PERFORM 5220-ERASEDEPT THRU
        5222-DELETEDEPTACT-EXIT
      ELSE
        PERFORM 5220-ERASEDEPT THRU
        5222-DELETEDEPTACT-EXIT
        IF ACTION = 'E' THEN
          PERFORM 5220-ERASEDEPT THRU
          5222-DELETEDEPTACT-EXIT
        ELSE
          PERFORM 5220-ERASEDEPT THRU
          5222-DELETEDEPTACT-EXIT
          IF ACTION = 'U' THEN
            PERFORM 5220-ERASEDEPT THRU
            5222-DELETEDEPTACT-EXIT
        END-IF
      END-IF
    END-IF
  END-IF
EXEC SQL CLOSE DEPT1 END-EXEC
EXIT.
  *---------------------------------------------------------------*
  * ERASE A DEPARTMENT                                           *
  *---------------------------------------------------------------*
5220-ERASEDEPT. 00161600
  MOVE 1 TO DEPTPTR.
  MOVE 0 TO LISTPTR.
  MOVE DATAW TO DEPTS (DEPTPTR).
  PERFORMANCE 5221-DELDEPTS THRU 5221-DELDEPTS-EXIT
  UNTIL DEPTPTR = 0.
  MOVE LISTPTR TO STACKTOP.
  PERFORMANCE 5223-DELDEPEND THRU 5223-DELDEPEND-EXIT
  UNTIL LISTPTR = 0.
  EXEC SQL OPEN LOCS END-EXEC.
  MOVE TO LOCPTR.
  PERFORMANCE 5224-DELETELOCSD THRU 2210-BUILDLOCSTABLE-EXIT
  UNTIL SQLCODE NOT EQUAL TO 0.
  EXEC SQL CLOSE DEPT1 END-EXEC
  EXEC SQL OPEN LOCS END-EXEC.
  MOVE 0 TO LOCPTR.
  PERFORMANCE 5224-DELETELOCSD THRU 2210-BUILDLOCSTABLE-EXIT
  UNTIL LOCPTR = STACKTOP.
  MOVE '013I' TO MSGCODE.
  CALL 'DSN8MCG' USING MODULE, MSGCODE, OUTMSG.
  EXEC SQL CLOSE DEPT1 END-EXEC
  EXEC SQL CLOSE DEPT1 END-EXEC
  IF SQLERRP = SPACES THEN
    5210-DISDEPTACT.
  ELSE
    EXEC SQL CLOSE DEPT1 END-EXEC
    CALL 'ISPLINK' USING I-DISPLAY, DEPT-PANEL
    IF RETURN-CODE NOT EQUAL TO 8 THEN
      EXEC SQL CLOSE DEPT1 END-EXEC
      IF SQLCODE = 100 THEN
        EXEC SQL CLOSE DEPT1 END-EXEC
        EXIT.
      END-IF
    END-IF
  END-IF
EXEC SQL CLOSE DEPT1 END-EXEC
EXIT.
  *---------------------------------------------------------------*
  * ERASE DEPARTMENT FROM OTHER LOCATIONS                         *
  *---------------------------------------------------------------*
5220-ERASEDEPT. 00161700
  PERFORMANCE 5220-DELETELOCSD THRU 2210-BUILDLOCSTABLE-EXIT
  UNTIL SQLCODE NOT EQUAL TO 0.
EXEC SQL CLOSE DEPT1 END-EXEC
EXIT.
  *---------------------------------------------------------------*
  * BUILD TABLE OF DEPARTMENTS DEPENDENT ON ERASED DEPARTMENTS    *
  * AND DEPARTMENTS DEPENDENT ON THOSE DEPARTMENTS ETC.           *
  *---------------------------------------------------------------*
5222-GETSUBDEPTS.
  EXEC SQL FETCH SUBDEPDS INTO :TEMPDEP END-EXEC.
  IF SQLCODE = 0 THEN
    ADD 1 TO DEPTPTR
    MOVE TEMPDEP TO DEPTS (DEPTPTR).
  END-IF
EXEC SQL CLOSE DEPT1 END-EXEC
EXIT.
  *---------------------------------------------------------------*
  * DISPLAY, ERASE, OR UPDATE DEPARTMENT                          *
  *---------------------------------------------------------------*
5210-DISDEPTACT. 00161800
  IF SQLERRP = SPACES THEN
    EXEC SQL CLOSE DEPT1 END-EXEC
    MOVE '013I' TO MSGCODE
    CALL 'DSN8MCG' USING MODULE, MSGCODE, OUTMSG
    EXEC SQL CLOSE DEPT1 END-EXEC
  ELSE
    EXEC SQL CLOSE DEPT1 END-EXEC
    PERFORM 5210-DELDEPTS THRU 5210-DELDEPTS-EXIT
    IF SQLCODE NOT EQUAL TO 0 THEN
      CALL 'ISPLINK' USING I-DISPLAY, DEPT-PANEL
      IF RETURN-CODE NOT EQUAL TO 8 THEN
        PERFORM 5220-ERASEDEPT THRU
        5222-DELETEDEPTACT-EXIT
      ELSE
        PERFORM 5220-ERASEDEPT THRU
        5222-DELETEDEPTACT-EXIT
        IF ACTION = 'E' THEN
          PERFORM 5220-ERASEDEPT THRU
          5222-DELETEDEPTACT-EXIT
        ELSE
          PERFORM 5220-ERASEDEPT THRU
          5222-DELETEDEPTACT-EXIT
          IF ACTION = 'U' THEN
            PERFORM 5220-ERASEDEPT THRU
            5222-DELETEDEPTACT-EXIT
        END-IF
      END-IF
    END-IF
  END-IF
EXEC SQL CLOSE DEPT1 END-EXEC
EXIT.
  *---------------------------------------------------------------*
  * BUILD TABLE OF DEPARTMENTS DEPENDENT ON ERASED DEPARTMENTS    *
  * AND DEPARTMENTS DEPENDENT ON THOSE DEPARTMENTS ETC.           *
  *---------------------------------------------------------------*
5222-GETSUBDEPTS.
  EXEC SQL FETCH SUBDEPDS INTO :TEMPDEP END-EXEC.
  IF SQLCODE = 0 THEN
    ADD 1 TO DEPTPTR
    MOVE TEMPDEP TO DEPTS (DEPTPTR).
  END-IF
EXEC SQL CLOSE DEPT1 END-EXEC
EXIT.
  *---------------------------------------------------------------*
EXIT.
* 00169700
  
* ENFORCE REFERENTIAL INTEGRITY RULE ON VHDEPT BY CASCADE  00169990
* DELETING DEPARTMENTS DEPENDENT ON DELETED DEPARTMENTS  00170100
  
5223-DELDEPEND.
  00170200
  MOVE DEPTLIST (LISTPTR) TO DELDEPT.
  00170300
  EXEC SQL DELETE FROM VHDEPT  00170600
  WHERE DEPTNO = :DELENDT  00170600
  END-EXEC.
  00170900
  SUBTRACT 1 FROM LISTPTR.
  00170800
5223-DELDEPEND-EXIT.
  00170900
  EXIT.
  00171100
  
* 00171200
  
P**ERFORM CASCADE DELETE AT ALL LOCATIONS  00171300
  
5224-DELETELOCs.
  IF LOCPR < LOCTOP THEN
    00171500
      ADD 1 TO LOCPTR
      00171700
      MOVE LOCLIST (LOCPR) TO TEMPPLOC
      00171800
      EXEC SQL CONNECT TO :TEMPLOC END-EXEC
      00171900
      MOVE STACKTOP TO LISTPTR
      00172000
      PERFORM 5223-DELDEPEND THRU 5223-DELDEPEND-EXIT
      UNTIL LISTPTR = 0.
  5224-DELETELOCs-EXIT.
  00172100
  EXIT.
  00172300
  
* 00172400
  
5230-UPDATEDEPT.
  EXEC SQL WHENEVER SQLERROR CONTINUE END-EXEC.
  00173100
  EXEC SQL UPDATE VHDEPT
  00173200
  SET DEPTNAME = :DEPT-NAME,
  00173300
  MGRNO = :DEPT-MGR,
  00173400
  ADMRDEPT = :DEPT-ADMR,
  00173500
  LOCATION = :DEPT-LOC,
  WHERE DEPTNO = :DECDATAW.
  00173600
  END-EXEC.
  00173800
  IF SQLCODE = -530 THEN
    00173900
      UNSTRING SQLERRMC
      INTO TOKEN
      00174200
      IF TOKEN = 'RDD' THEN
        00174300
          MOVE '215E' TO MSGCODE
          00174400
          CALL 'DSN8MCG' USING MODULE, MSGCODE, OUTMSG
          00174500
          MOVE OUTMSG TO MSGS
          00174600
          IF TOKEN = 'RDE' THEN
            00174700
              CALL 'DSN8MCG' USING MODULE, MSGCODE, OUTMSG
              00174800
          ELSE
            00174900
              GO TO L8000-P3-DBERROR.
            00175000
          ELSE
            00175100
              GO TO L8000-P3-DBERROR.
            00175200
          END-EXEC.
          00175300
        00175400
      ELSE
        00175500
          GO TO L8000-P3-DBERROR.
        00175600
      END-EXEC.
      00175700
    00174000
  ELSE
    00175800
      MOV OUTMSG TO MSGS
      CALL 'DSN8MCG' USING MODULE, MSGCODE, OUTMSG
      MOVE '214E' TO MSGCODE
      IF TOKEN = 'RDE' THEN
        00175900
          CALL 'DSN8MCG' USING MODULE, MSGCODE, OUTMSG
          MOVE '215E' TO MSGCODE
          IF TOKEN = 'RDD' THEN
            00176000
              INTO TOKEN
              00176100
              DELIMITED BY HIGH-VALUE
              00176200
              UNSTRING SQLERRMC
              00176300
              WHERE DEPTNO = :DELENDT.
              00176400
              EXEC SQL WHENEVER SQLERROR GOTO L8000-P3-DBERROR.
              00176500
              EXEC SQL OPEN LOCS END-EXEC.
              00176600
              PERFORM 2300-GETEMPREC THRU 2300-GETEMPREC-EXIT.
              00176700
              EXEC SQL WHENEVER SQLERROR CONTINUE END-EXEC.
              00176800
            00176900
          ELSE
            00177000
              GO TO L8000-P3-DBERROR.
            00177100
          ELSE
            00177200
              EXEC SQL WHENEVER SQLERROR CONTINUE END-EXEC.
              00177300
              PERFORM 2300-GETEMPREC THRU 2300-GETEMPREC-EXIT.
              00177400
              EXEC SQL WHENEVER SQLERROR CONTINUE END-EXEC.
              00177500
              EXIT.
              00177700
  00177800
* UPDATE DEPARTMENT TO VHDEPT VIEWS AT ALL LOCATIONS *

** 5231-UPDATELOC.

** IF LOCPT < LOCTOP THEN

** ADD 1 TO LOCPT

** MOVE LOCPT (LOCPT) TO TEMPOC

** EXEC SQL CONNECT TO :TEMPLOC END-EXEC

** EXEC SQL UPDATE VHDEPT

** SET DEPTNAME = :DEPT-NAME,

** MGRNO = :DEPT-MGR,

** ADMREPT = :DEPT-ADM,

** LOCATION = :DEPT-LOC

** WHERE DEPTNO = :DEPT-NUMB

** END-EXEC.

** 5231-UPDATELOC-EXIT.

** EXIT.

** DISPLAY DEPARTMENT STRUCTURE

** 5300-STRUCTURE.

** MOVE SPACES TO DEPT-RECORD.

** MOVE SPACES TO EMP-RECORD.

** MOVE SPACES TO DEPT1-RECORD.

** MOVE SPACES TO EMP1-RECORD.

** EXEC SQL OPEN DEPT1 END-EXEC.

** MOVE SPACES TO SQLERRP.

** EXEC SQL FETCH DEPT1 INTO :DEPT1-NUMB, :DEPT1-NAME,

** :DEPT1-MGR:DEPT1-MGR-IND,

** :DEPT1-ADMR, :DEPT1-Loc,

** :EMP1-FIRST-NAME,

** :EMP1-MID-INIT,

** :EMP1-LAST-NAME,

** :EMP1-WK-DEPT:WORK1-DEPT-IND

** END-EXEC.

** PERFORM 5310-DISSTR THRU 5310-DISSTR-EXIT.

** DISPLAY DEPARTMENTS EXIT.

** EXIT.

** DISPLAY DEPARTMENTS REPORTING TO SELECTED DEPARTMENT

** 5310-DISSTR.

** IF SQLERRP = SPACES THEN

** EXEC SQL CLOSE DEPT1 END-EXEC

** MOVE '079E' TO MSGCODE

** CALL 'DSN8MCG' USING MODULE, MSGCODE, OUTMSG

** MOVE OUTMSG TO MSGS

** ELSE

** IF SQLCODE = 100 THEN

** EXEC SQL CLOSE DEPT1 END-EXEC

** MOVE '0111' TO MSGCODE

** CALL 'DSN8MCG' USING MODULE, MSGCODE, OUTMSG

** MOVE OUTMSG TO MSGS

** ELSE

** EXEC SQL CLOSE DEPT1 END-EXEC

** PERFORM 5311-CHECKDEPT1IND THRU

** 5311-CHECKDEPT1IND-EXIT

** CALL 'ISPLINK' USING I-TBCREATE, DS-TABLE, W-BLANK,

** DS-VARS, I-NOWRITE, I-REPLACE

** EXEC SQL OPEN DEPTSTR END-EXEC

** PERFORM 5312-GETSTRTAB THRU 5312-GETSTRTAB-EXIT

** UNTIL SQLCODE NOT EQUAL TO 0

** EXEC SQL CLOSE DEPTSTR END-EXEC

** CALL 'ISPLINK' USING I-TBDISPL, DS-TABLE, STR-PANEL

** CALL 'ISPLINK' USING I-VPUT, HEAD-DEPT-VARS

** CALL 'ISPLINK' USING I-TBCLOSE, DS-TABLE.

** 5310-DISSTR-EXIT.

** EXIT.

** IF MGRNO NULL, MOVE BLANKS INTO FIELD

** 5311-CHECKDEPT1IND.

** IF DEPT1-MGR-IND < 0 THEN

** MOVE SPACES TO DEPT1-MGR.

** 5311-CHECKDEPT1IND-EXIT.

** EXIT.
EXEC SQL OPEN ALLEMP1 END-EXEC
UNTIL SQLCODE NOT EQUAL TO 0 OR GENE-EXIT = 'Y'
PERFORM 6111-ALLEMP1 THRU 6111-ALLEMP1-EXIT
MOVE SPACES TO SQLERRP
EXEC SQL CLOSE ALLEMP1 END-EXEC

IF SEARCH-CRIT = 'EI' THEN
6110-GETEMPTAB.
*---------------------------------------------------------------*
* CREATE TABLE OF EMPLOYEES TO FIT SEARCH-CRIT                  *
*---------------------------------------------------------------*
EXIT.

6100-GENEMP-EXIT.
CALL 'ISPLINK' USING I-TBCLOSE, EMP-TABLE.
MOVE 'Y' TO GENE-EXIT.
IF SPECIAL-EXIT = 'Y' THEN
PERFORM 6200-DISPLAYEMP THRU 6200-DISPLAYEMP-EXIT.
6000-EMPLOYEE-EXIT.
EXIT.

*---------------------------------------------------------------*
* GENERIC LIST OF EMPLOYEES                                      *
*---------------------------------------------------------------*
6100-GENEMP.
CALL 'ISPLINK' USING I-TBCREATE, EMP-TABLE, W-BLANK,
SEL-EMP-VARS, I-NOWRITE, I-REPLACE.
MOVE SPACE TO SEL-EMP.
PERFORM 6110-GETEMPTAB THRU 6110-GETEMPTAB-EXIT.
CALL 'ISPLINK' USING I-TBQUERY, EMP-TABLE, W-BLANK, W-BLANK,
ROWS.
IF NUMROWS = 1 AND DAWAT = GENDATA THEN
MOVE 'Y' TO SPECIAL-EXIT
CALL 'ISPLINK' USING I-TBGET, EMP-TABLE
MOVE EMP-NUMB TO DAWAT
ELSE
MOVE 'N' TO SPECIAL-EXIT
IF NUMROWS = 0 THEN
PERFORM 6120-EMPMSG THRU 6120-EMPMSG-EXIT
MOVE 'Y' TO GENE-EXIT.
ELSE
CALL 'ISPLINK' USING I-VPUT, ACTL-VAR
CALL 'ISPLINK' USING I-TBTOP, EMP-TABLE
CALL 'ISPLINK' USING I-TBDISPL, EMP-TABLE,
GENE-PANEL.
IF RETURN-CODE = 8 THEN
MOVE 'Y' TO GENE-EXIT.
ELSE
IF ROWS-CHANGED > 0 THEN
CALL 'ISPLINK' USING I-TBGET, EMP-TABLE
MOVE EMP-NUMB TO DAWAT
ELSE
MOVE 'Y' TO GENE-EXIT.
IF GENE-EXIT = 'N' THEN
PERFORM 6200-DISPLAYEMP THRU 6200-DISPLAYEMP-EXIT.
IF SPECIAL-EXIT = 'Y' THEN
MOVE 'Y' TO GENE-EXIT.
CALL 'ISPLINK' USING I-TBCLOSE, EMP-TABLE.
6100-GENEMP-EXIT.
EXIT.

*---------------------------------------------------------------*
* CREATE TABLE OF EMPLOYEES TO FIT SEARCH-CRIT                  *
*---------------------------------------------------------------*
6110-GETEMPTAB.
IF SEARCH-CRIT = 'EI' THEN
EXEC SQL OPEN ALLEMP1 END-EXEC
MOVE SPACES TO SQLERRP
PERFORM 6111-ALLEMP1 THRU 6111-ALLEMP1-EXIT
UNTIL SQLCODE NOT EQUAL TO 0 OR GENE-EXIT = 'Y'
EXEC SQL CLOSE ALLEMP1 END-EXEC
ELSE
ELSE
  MOVE '001I' TO MSGCODE.
  CALL 'DSN8MCG' USING MODULE, MSGCODE, OUTMSG
  MOVE OUTMSG TO MSGS.
  6120-EMPSMSG-EXIT.
  EXIT.
*  DISPLAY AN EMPLOYEE  *  6200-DISPLAYEMP
  MOVE SPACES TO DEPT-RECORD.
  MOVE SPACES TO EMP-RECORD.
  EXEC SQL OPEN EMP1 END-EXEC.
  MOVE SPACES TO SQLERRP.
  END-EXEC.
  PERFORM 6120-DISEMPSACT THRU 6120-DISEMPSACT-EXIT.
  6200-DISPLAYEMP-EXIT.
  EXIT.
*  DISPLAY, ERASE, OR UPDATE EMPLOYEE  *  6210-DISEMPSACT
  IF SQLERRP = SPACES THEN
    EXEC SQL CLOSE EMP1 END-EXEC
    MOVE '079E' TO MSGCODE
    CALL 'DSN8MCG' USING MODULE, MSGCODE, OUTMSG
    MOVE OUTMSG TO MSGS
  ELSE
    IF SQLCODE = 100 THEN
      EXEC SQL CLOSE EMP1 END-EXEC
      MOVE '063I' TO MSGCODE
      CALL 'DSN8MCG' USING MODULE, MSGCODE, OUTMSG
      MOVE OUTMSG TO MSGS
    ELSE
      EXEC SQL DELETE FROM VEMP WHERE EMPNO = :DATAW
      EXEC SQL DELETE FROM EMP1
      6220-ERASEEMP-EXIT.
    END-EXEC.
    PERFORM 6230-UPDATEEMP THRU 6230-UPDATEEMP-EXIT.
    6230-UPDATEEMP-EXIT.
  EXIT.
*  ERASE AN EMPLOYEE  *  6220-ERASEEMP
  EXEC SQL DELETE FROM VEMP
  WHERE EMPNO = :DATAW
  END-EXEC.
  IF SQLCODE = 0 THEN
    MOVE '083I' TO MSGCODE
    CALL 'DSN8MCG' USING MODULE, MSGCODE, OUTMSG
    MOVE OUTMSG TO MSGS.
  ELSE
    CALL 'ISPLINK' USING I-DISPLAY, EMP-PANEL
    3310-CHECKDEPTIND-EXIT
    PERFORM 3310-CHECKDEPTIND THRU 3310-CHECKDEPTIND-EXIT
    IF RETURN-CODE NOT EQUAL TO 8 THEN
      IF ACTION = 'E' THEN
        PERFORM 6220-ERASEEMP THRU 6220-ERASEEMP-EXIT
      ELSE
        PERFORM 6120-EMPMSG THRU 6120-EMPMSG-EXIT
      END-EXEC.
    ELSE
      IF ACTION = 'U' THEN
        PERFORM 6230-UPDATEEMP THRU 6230-UPDATEEMP-EXIT
      6230-UPDATEEMP-EXIT.
    END-EXEC.
  EXIT.
*  UPDATE AN EMPLOYEE  *  6230-UPDATEEMP
  CALL 'DSN8MCG' USING MODULE, MSGCODE, OUTMSG
  MOVE OUTMSG TO MSGS.
  IF DEPT-LOC NOT EQUAL TO LOCATION THEN
    MOVE '217E' TO MSGCODE
    CALL 'DSN8MCG' USING MODULE, MSGCODE, OUTMSG
    MOVE OUTMSG TO MSGS
  ELSE
    MOVE SPACES TO DEPT-RECORD.
    MOVE SPACES TO EMP-RECORD.
    EXEC SQL OPEN EMP1 END-EXEC.
    MOVE SPACES TO SQLERRP.
    END-EXEC.
    PERFORM 6120-DISEMPSACT THRU 6120-DISEMPSACT-EXIT.
    6200-DISPLAYEMP-EXIT.
  EXIT.
*  DISPLAY AN EMPLOYEE  *  6200-DISPLAYEMP
  EXEC SQL DELETE FROM VEMP
  WHERE EMPNO = :DATAW
  EXEC SQL DELETE FROM EMP1
  6220-ERASEEMP-EXIT.
  6120-EMPSMSG-EXIT.
  EXIT.
EXEC SQL WHENEVER SQLERROR CONTINUE END-EXEC
EXEC SQL UPDATE VEMP
   SET FIRSTNME = :EMP-FIRST-NAME, MIDINIT = :EMP-MID-INIT,
   LASTNAME = :EMP-LAST-NAME, WORKDEPT = :EMP-WORK-DEPT
WHERE EMPNO = :DATAW
END-EXEC
IF SQLCODE = -530 THEN
   MOVE '203E' TO MSGCODE
   CALL 'DSN8MCG' USING MODULE, MSGCODE, OUTMSG
   MOVE OUTMSG TO MSGS
ELSE
   IF SQLCODE = 0 THEN
      MOVE '004I' TO MSGCODE
      CALL 'DSN8MCG' USING MODULE, MSGCODE, OUTMSG
      MOVE OUTMSG TO MSGS
   ELSE
      GO TO L8000-P3-DBERROR.
   END-EXEC
   CALL 'ISPLINK' USING I-DISPLAY, EMP-PANEL.
6230-UPDATEEMP-EXIT.
EXIT.
*--------------------------------------------------------------*
* DB2 ERROR PROCESSING                                          *
*--------------------------------------------------------------*
L8000-P3-DBERROR.
MOVE SQLCAID TO SQLCAID-VALUE.
MOVE SQLCABC TO CONV.
MOVE SQLERRM TO SQLERRM-VALUE.
MOVE SQLERRRC TO SQLERRRC-VALUE.
MOVE SQLERRRM TO SQLERRRM-VALUE.
MOVE SQLERRRTO SQLERRRTO-VALUE.
MOVE SQLERRT TO SQLERRT-VALUE.
MOVE SQLERRR (1) TO CONV.
MOVE SQLERRRTO SQLERRRTO-VALUE.
MOVE SQLERRRT TO SQLERRRT-VALUE.
MOVE SQLERRD (2) TO CONV.
MOVE SQLERRD2 TO SQLERRD2-VALUE.
MOVE SQLERRD3 TO SQLERRD3-VALUE.
MOVE SQLERRD4 TO SQLERRD4-VALUE.
MOVE SQLERRD5 TO SQLERRD5-VALUE.
MOVE SQLERRD6 TO SQLERRD6-VALUE.
MOVE SQLERRD7 TO SQLERRD7-VALUE.
MOVE SQLERRD8 TO SQLERRD8-VALUE.
MOVE SQLERRD9 TO SQLERRD9-VALUE.
MOVE SQLERRA TO SQLERRA-VALUE.
MOVE SQLSTATE TO SQLSTATE-VALUE.
OPEN OUTPUT MSGOUT.
WRITE MSGREC FROM SQLCA-LINE0.
WRITE MSGREC FROM SQLCA-LINE1.
WRITE MSGREC FROM SQLCA-LINE2.
WRITE MSGREC FROM SQLCA-LINE3.
WRITE MSGREC FROM SQLCA-LINE4.
WRITE MSGREC FROM SQLCA-LINE5.
WRITE MSGREC FROM SQLCA-LINE6.
WRITE MSGREC FROM SQLCA-LINE7.
WRITE MSGREC FROM SQLCA-LINE8.
WRITE MSGREC FROM SQLCA-LINE9.
WRITE MSGREC FROM SQLCA-LINE10.
WRITE MSGREC FROM SQLCA-LINE11.
WRITE MSGREC FROM SQLCA-LINE12.
WRITE MSGREC FROM SQLCA-LINE13.
WRITE MSGREC FROM SQLCA-LINE14.
CLOSE MSGOUT.
GOBACK.
THIS MODULE LISTS EMPLOYEE PHONE NUMBERS AND UPDATES THEM IF DESIRED.

<table>
<thead>
<tr>
<th>Line</th>
<th>Description</th>
<th>Symbolic Label/Name</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>IDENTIFICATION DIVISION.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>PROGRAM-ID. DSN8SC3.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>* MODULE NAME = DSN8SC3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>* DESCRIPTIVE NAME = DB2 SAMPLE APPLICATION PHONE APPLICATION</td>
<td>DSN8SSL</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>* ISPF</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>* COBOL</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>* COPYRIGHT = 5615-DB2 (C) COPYRIGHT 1982, 2013 IBM CORP.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>* SEE COPYRIGHT INSTRUCTIONS</td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>* LICENSED MATERIALS - PROPERTY OF IBM</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>* STATUS = STATUS = VERSION 11</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>* FUNCTION = THIS MODULE LISTS EMPLOYEE PHONE NUMBERS AND UPDATES THEM IF DESIRED.</td>
<td>VEMPLP</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>* NOTES = DEPENDENCIES = TWO ISPF PANELS ARE REQUIRED: DSN8SSL AND DSN8SSN</td>
<td></td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>* RESTRICTIONS = NONE</td>
<td></td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>* MODULE TYPE = VS COBOL II PROGRAM</td>
<td></td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>* PROCESSOR = DB2 PRECOMPILED, VS COBOL II</td>
<td></td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>* MODULAR SIZE = SEE LINKEDIT</td>
<td></td>
<td></td>
</tr>
<tr>
<td>17</td>
<td>* ATTRIBUTES = NOT REENTRANT OR REUSABLE</td>
<td></td>
<td></td>
</tr>
<tr>
<td>18</td>
<td>* ENTRY POINT = DSN8SC3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>19</td>
<td>* PURPOSE = SEE FUNCTION</td>
<td></td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>* LINKAGE = INVOKED FROM ISPF</td>
<td></td>
<td></td>
</tr>
<tr>
<td>21</td>
<td>* INPUT = PARAMETERS EXPLICITLY PASSED TO THIS FUNCTION: INPUT-MESSAGE:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>22</td>
<td>* OUTPUT = PARAMETERS EXPLICITLY RETURNED: OUTPUT-MESSAGE:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>23</td>
<td>* SYMOLIC LABEL/NAME = DSN8SSL</td>
<td></td>
<td></td>
</tr>
<tr>
<td>24</td>
<td>* DESCRIPTION = PHONE MENU 1 (SELECT)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>25</td>
<td>* SYMOLIC LABEL/NAME = DSN8SSN</td>
<td></td>
<td></td>
</tr>
<tr>
<td>26</td>
<td>* DESCRIPTION = PHONE MENU 2 (LIST)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>27</td>
<td>* SYMOLIC LABEL/NAME = VPHONE</td>
<td></td>
<td></td>
</tr>
<tr>
<td>28</td>
<td>* DESCRIPTION = VIEW OF TELEPHONE DATA</td>
<td></td>
<td></td>
</tr>
<tr>
<td>29</td>
<td>* SYMOLIC LABEL/NAME = VEMPLP</td>
<td></td>
<td></td>
</tr>
<tr>
<td>30</td>
<td>* DESCRIPTION = VIEW OF EMPLOYEE DATA</td>
<td></td>
<td></td>
</tr>
<tr>
<td>31</td>
<td>* SYMOLIC LABEL/NAME = DSN8SSL</td>
<td></td>
<td></td>
</tr>
<tr>
<td>32</td>
<td>* DESCRIPTION = PHONE MENU 1 (SELECT)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>33</td>
<td>* SYMOLIC LABEL/NAME = DSN8SSN</td>
<td></td>
<td></td>
</tr>
<tr>
<td>34</td>
<td>* DESCRIPTION = PHONE MENU 2 (LIST)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>35</td>
<td>* EXIT-NORMAL = RETURN CODE 0 NORMAL COMPLETION</td>
<td></td>
<td></td>
</tr>
<tr>
<td>36</td>
<td>* EXIT-ERROR =</td>
<td></td>
<td></td>
</tr>
<tr>
<td>37</td>
<td>* RETURN CODE = NONE</td>
<td></td>
<td></td>
</tr>
<tr>
<td>38</td>
<td>* ABEND CODES = NONE</td>
<td></td>
<td></td>
</tr>
<tr>
<td>39</td>
<td>* ERROR-MESSAGES =</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Chapter 12. Sample data and applications supplied with Db2 for z/OS 1089
EXEC SQL DECLARE VPHONE TABLE {
  (LASTNAME VARCHAR(15),
   FIRSTNAME VARCHAR(12),
   MIDDLEINITIAL CHAR(1),
   PHONENUMBER CHAR(4),
   EMPLOYEE NUMBER CHAR(6))
}
Chapter 12. Sample data and applications supplied with Db2 for z/OS  1091

DEPTNUMBER  CHAR(3) NOT NULL, 00023500
DEPTNAME    VARCHAR(36) NOT NULL) END-EXEC. 00023600

** STRUCTURE FOR PHONE RECORD ** 00023700
**---------------------------------------------------------------* 00023800
01 PPHONE. 00023900
  02 LAST-NAME PIC X(15). 00024100
  02 FIRST-NAME PIC X(12). 00024200
  02 MIDDLE-INITIAL PIC X(01). 00024300
  02 PHONE-NUMBER PIC X(04). 00024400
  02 EMPLOYEE-NUMBER PIC X(06). 00024500
  02 DEPT-NUMBER PIC X(03). 00024600
  02 DEPTNAME PIC X(36). 00024700

** SQL DECLARATION FOR VIEW VEMPLP ** 00024800
**---------------------------------------------------------------* 00024900
EXEC SQL DECLARE VEMPLP TABLE 00025000
  EMPLOYEENUMBER CHAR(6) 00025100
  'PHONE NUMBER CHAR(4)) END-EXEC. 00025200

** SQL CURSORS ** 00025300
**---------------------------------------------------------------* 00025400
EXEC SQL DECLARE TELE1 CURSOR FOR 00025500
  SELECT * 00025600
  FROM VPHONE 00025700
  END-EXEC. 00025800

EXEC SQL DECLARE TELE2 CURSOR FOR 00025900
  SELECT * 00026000
  FROM VPHONE 00026100
WHERE LASTNAME LIKE :LNAMEW 00026200
  AND FIRSTNAME LIKE :FNAMEW 00026300
  END-EXEC. 00026400

EXEC SQL DECLARE TELE3 CURSOR FOR 00026500
  SELECT * 00026600
  FROM VPHONE 00026700
WHERE LASTNAME = :LNAMEW 00026800
  AND FIRSTNAME LIKE :FNAMEW 00026900
  END-EXEC. 00027000

EXEC SQL WHENEVER SQLERROR GOTO L8000-P3-DBERROR END-EXEC. 00027100
EXEC SQL WHENEVER SQLWARNING GOTO L8000-P3-DBERROR END-EXEC. 00027200
EXEC SQL WHENEVER NOT FOUND CONTINUE END-EXEC. 00027300

** SQL RETURN CODE HANDLING ** 00027400
**---------------------------------------------------------------* 00027500
EXEC SQL WHENEVER SQLERROR GOTO L8000-P3-DBERROR END-EXEC. 00027600
EXEC SQL WHENEVER SQLWARNING GOTO L8000-P3-DBERROR END-EXEC. 00027700
EXEC SQL WHENEVER NOT FOUND CONTINUE END-EXEC. 00027800

** DEFINE COBOL - SPF VARIABLES ** 00027900
**---------------------------------------------------------------* 00028000
0000-PROGRAM-START. 00028100
  CALL 'ISPLINK' USING I-VDEFINE, CH-VAR, ROWS-CHANGED, 00028200
    I-CHAR, CH-VAR-STG. 00028300
  CALL 'ISPLINK' USING I-VDEFINE, FN-VAR, EMP-FIRST-NAME, 00028400
    I-CHAR, FN-VAR-STG. 00028500
  CALL 'ISPLINK' USING I-VDEFINE, MI-VAR, EMP-MIDDLE-INITIAL, 00028600
    I-CHAR, MI-VAR-STG. 00028700
  CALL 'ISPLINK' USING I-VDEFINE, LN-VAR, EMPLAST, 00028800
    I-CHAR, LN-VAR-STG. 00028900
  CALL 'ISPLINK' USING I-VDEFINE, PN-VAR, EMPPHONE, 00029000
    I-CHAR, PN-VAR-STG. 00029100
  CALL 'ISPLINK' USING I-VDEFINE, EN-VAR, EMPNUMB, 00029200
    I-CHAR, EN-VAR-STG. 00029300
  CALL 'ISPLINK' USING I-VDEFINE, WD-VAR, EMP-DEPT-NUMBER, 00029400
    I-CHAR, WD-VAR-STG. 00029500
  CALL 'ISPLINK' USING I-VDEFINE, WN-VAR, EMP-DEPTNAME, 00029600
    I-CHAR, WN-VAR-STG. 00029700
  CALL 'ISPLINK' USING I-VDEFINE, FI-VAR, FNAMEI, 00029800
    I-CHAR, FI-VAR-STG. 00029900
  CALL 'ISPLINK' USING I-VDEFINE, LI-VAR, LNAMEI, 00030000
    I-CHAR, LI-VAR-STG. 00030100
  CALL 'ISPLINK' USING I-VDEFINE, MSGS-VAR, MSGS, 00030200
    I-CHAR, MSGS-VAR-STG. 00030300

** MAIN PROGRAM ** 00030400
**---------------------------------------------------------------* 00030500
MOVE 'N' TO SEL-EXIT. 00030600
PERFORM 1000-MAIN-LOOP THRU 1000-MAIN-LOOP-EXIT 00030700
UNTIL SEL-EXIT = 'Y'.
MOVE 0 TO RETURN-CODE.
GOBACK.
* 1000-MAIN-LOOP.
CALL 'ISPLINK' USING I-DISPLAY, SEL-PANEL.
MOVE SPACES TO MSGS.
MOVE SPACES TO OUTMSG.
IF RETURN-CODE = 8
MOVE 'Y' TO SEL-EXIT
ELSE
MOVE 'N' TO DISPLAY-TABLE
CALL 'ISPLINK' USING I-VGET, SEL-VARS
MOVE LNAM1 TO LNAM1
MOVE FNAM1 TO FNAM1
PERFORM 3000-GET-TYPE THRU 3000-GET-TYPE-EXIT
IF DISPLAY-TABLE = 'Y'
PERFORM 6000-DISPLAY LIST
THRU 6000-DISPLAY-EXIT.
CALL 'ISPLINK' USING I-VPUT MSGS-VAR.
1000-MAIN-LOOP-EXIT.
EXIT.
*
* DETERMINE PROCESSING REQUEST
** LIST ALL EMPLOYEES
* 3000-LIST-ALL.
EXEC SQL OPEN TELE1 END-EXEC.
MOVE SPACES TO SQLERRP.
EXEC SQL FETCH TELE1 INTO :PPHONE END-EXEC.
IF SQLERRP = '079E'
MOVE '0081' TO MSGCODE
CALL 'DSN8MCG' USING MODULE, MSGCODE, OUTMSG
MOVE OUTMSG TO MSGS
ELSE
IF SOLCODE = 100
MOVE '0081' TO MSGCODE
CALL 'DSN8MCG' USING MODULE, MSGCODE, OUTMSG
MOVE OUTMSG TO MSGS
ELSE
MOVE 'Y' TO DISPLAY-TABLE
CALL 'ISPLINK' USING I-TBCREATE, TABLE-NAME,
W-BLANK, EMP-VARS, I-NOWRITE, I-REPLACE
PERFORM 3500-LIST-AND-GET
THRU 3500-LIST-AND-GET-EXIT
UNTIL SOLCODE NOT EQUAL 0.
EXEC SQL CLOSE TELE1 END-EXEC.
3000-LIST-ALL-EXIT.
EXIT.
*
3500-LIST-AND-GET.
MOVE PPHONE TO EMP-RECORD.
Related reference

“Sample applications in TSO” on page 1027
A set of Db2 sample applications run in the TSO environment.

**DSN8SP3**

THIS MODULE LISTS EMPLOYEE PHONE NUMBERS AND UPDATES THEM IF DESIRED.
DO WHILE NOT EXIT-PRESSED

PROCEDURE

*PSEUDOCODE*

HAS BEEN ESTABLISHED. ISSUE 079E IF NOT.

CHECK SQLERRP FOR NON-BLANKS TO ENSURE CONNECTION V2R3
HAD BEEN ESTABLISHED. ISSUE 079E IF NOT.

*PROCODE*

PROCEDURE

DO WHILE NOT EXIT-PRESSED
CALL GET-TYPE
CALL GET-LIST
CALL DISPLAY-LIST

GET_TYPE:
IF LASTNAME IS '*'
  TYPE = 'ALL'
ELSE
  IF LASTNAME CONTAINS '%'
    TYPE = 'GENERIC'
  ELSE
    TYPE = 'SPECIFIC'

GET_LIST:
CASE (TYPE)
  SUBCASE ('ALL')
    GET ALL EMPLOYEES
  SUBCASE ('GENERIC')
    GET GENERIC EMPLOYEES
  SUBCASE ('SPECIFIC')
    GET SPECIFIC EMPLOYEES
ENDSUB

DISPLAY_LIST:
DISPLAY LIST
IF NOT EXIT-PRESSED
  UPDATE PHONE NUMBER(S)
  WRITE CONFIRMATION MESSAGE
END

P3_DBERROR:
IF SQL ERROR OCCURS THEN
  FORMAT ERROR MESSAGE
  ROLLBACK
END

END.
INIT('( ZTDSELS FNAMED MINITD LNAMED PNOD ENOD WDEPTD WNAMED )');

DCL EMP_VARS CHAR(48) STATIC /* DISPLAY VARS */
INIT('( FNAMED MINITD LNAMED PNOD ENOD WDEPTD WNAMED )');

/********************************************************************/
/* ISPF DIALOG SERVICES DECLARATIONS */
/********************************************************************/
/* PROGRAM NAME */
DCL ISPLINK EXTERNAL ENTRY OPTIONS(ASM INTER RETCODE);

/* ISPF DIALOG SERVICE TYPES */
DCL I_VDEFINE CHAR(8) STATIC INIT('VDEFINE ');
DCL I_VGET CHAR(8) STATIC INIT('VGET ');
DCL I_VPUT CHAR(8) STATIC INIT('VPUT ');
DCL I_DISPLAY CHAR(8) STATIC INIT('DISPLAY ');
DCL I_TBDISPL CHAR(8) STATIC INIT('TBDISPL ');
DCL I_TBTOP CHAR(8) STATIC INIT('TBTOP ');
DCL I_TBCREATE CHAR(8) STATIC INIT('TBCREATE ');
DCL I_TBCLOSE CHAR(8) STATIC INIT('TBCLOSE ');
DCL I_TBADD CHAR(8) STATIC INIT('TBADD ');
DCL I_TBPUT CHAR(8) STATIC INIT('TBPUT ');

/* ISPF CALL MODIFIERS */
DCL I_NOWRITE CHAR(8) STATIC INIT('NOWRITE ');
DCL I_REPLACE CHAR(8) STATIC INIT('REPLACE ');
DCL I_CHAR CHAR(8) STATIC INIT('CHAR ');

/* PANEL NAMES */
DCL SEL_PANEL CHAR(8) STATIC INIT('DSN8SSL'); /* SELECTION PANEL */
DCL DIS_PANEL CHAR(8) STATIC INIT('DSN8SSN'); /* LIST PANEL */

/* LOCAL VARIABLES FOR ISPF VARIABLES */
DCL LNAMEI CHAR(15) INIT(' '); /* LAST-NAME INPUT */
DCL FNAMEI CHAR(12) INIT(' '); /* FIRST-NAME INPUT */
DCL MSGS CHAR(79) INIT(' '); /* MESSAGE FOR ISPF PANEL */

DCL 1 EMP_RECORD,                  /* PANEL DISPLAY INFORMATION */
    2 LASTNAME    CHAR (15),
    2 FIRSTNAME  CHAR (12),
    2 MIDDLEINITIAL CHAR (1),
    2 PHONENUMBER CHAR (4),
    2 EMPLOYEENUMBER CHAR (6),
    2 DEPTNUMBER  CHAR (3),
    2 DEPTNAME    CHAR (36);

1/********************************************************************/
/* DECLARATION FOR PROGRAM LOGIC */
1/********************************************************************/
/* CONSTANTS */
DCL YES BIT(1) STATIC INIT('1'B);
DCL NO BIT(1) STATIC INIT('0'B);
DCL ZERO FIXED BIN(31,0) STATIC INIT(0);

/* FLAGS */
DCL SEL_EXIT BIT(1);            /* EXIT PRESSED? FLAG */
DCL DIS_EXIT BIT(1);            /* EXIT PRESSED? FLAG */
DCL DIS_TABLE BIT(1);           /* DISPLAY-TABLE? FLAG */
DCL MORE_CHANGES BIT(1);        /* MORE CHANGES TO PROCESS? */

/* DATA VARIABLES */
DCL ROWS_CHANGED PIC'9999';
DCL TYPE CHAR(8);               /* TYPE OF LIST */
DCL LNAMES CHAR(15);            /* LAST NAME SELECTION VALUE */
DCL FNAMES CHAR(12);            /* FIRST NAME SELECTION VALUE */

1/********************************************************************/
/* SQL DECLARATIONS */
1/********************************************************************/

EXEC SQL INCLUDE SQLCA;
DCL SQL_PIC PIC'-999';

EXEC SQL DECLARE VPHONE TABLE
  (LASTNAME VARCHAR(15),
   FIRSTNAME VARCHAR(12),
   MIDDLEINITIAL CHAR(1),
   PHONENUMBER CHAR(4),
   EMPLOYEENUMBER CHAR(6),
   ...
DEPTNUMBER CHAR(3) NOT NULL,
DEPTNAME VARCHAR(36) NOT NULL);
/* STRUCTURE FOR PHONE RECORD */
DCL 1 PPHONE,
  2 LASTNAME CHAR (15) VAR,
  2 FIRSTNAME CHAR (12) VAR,
  2 MIDDLEINITIAL CHAR (1),
  2 PHONENUMBER CHAR (4),
  2 EMPLOYEENUMBER CHAR (6),
  2 DEPTNUMBER CHAR (3),
  2 DEPTNAME CHAR (36) VAR;
/* SQL DECLARATION FOR VIEW VEMPLP */
EXEC SQL DECLARE VEMPLP TABLE
  (EMPLOYEENUMBER CHAR(6),
   PHONENUMBER CHAR(4));

EXEC SQL DECLARE TELE1 CURSOR FOR
SELECT *
FROM VPHONE;
EXEC SQL DECLARE TELE2 CURSOR FOR
SELECT *
FROM VPHONE
WHERE LASTNAME LIKE :LNAMES
  AND FIRSTNAME LIKE :FNAMES;
EXEC SQL DECLARE TELE3 CURSOR FOR
SELECT *
FROM VPHONE
WHERE LASTNAME  =  :LNAMES
  AND FIRSTNAME LIKE :FNAMES;

EXEC SQL WHENEVER SQLERROR GOTO P3_DBERROR;
EXEC SQL WHENEVER SQLWARNING GOTO P3_DBERROR;
EXEC SQL WHENEVER NOT FOUND CONTINUE;

/* DEFINE PL/I - ISPF VARIABLES */
CALL ISPLINK(I_VDEFINE, CH_VAR, ROWS_CHANGED,
  I_CHAR, STG(ROWS_CHANGED));
CALL ISPLINK(I_VDEFINE, FN_VAR, EMP_RECORD.FIRSTNAME,
  I_CHAR, STG(EMP_RECORD.FIRSTNAME));
CALL ISPLINK(I_VDEFINE, MI_VAR, EMP_RECORD.MIDDLEINITIAL,
  I_CHAR, STG(EMP_RECORD.MIDDLEINITIAL));
CALL ISPLINK(I_VDEFINE, LN_VAR, EMP_RECORD.LASTNAME,
  I_CHAR, STG(EMP_RECORD.LASTNAME));
CALL ISPLINK(I_VDEFINE, PN_VAR, EMP_RECORD.PHONENUMBER,
  I_CHAR, STG(EMP_RECORD.PHONENUMBER));
CALL ISPLINK(I_VDEFINE, EN_VAR, EMP_RECORD.EMPLOYEENUMBER,
  I_CHAR, STG(EMP_RECORD.EMPLOYEENUMBER));
CALL ISPLINK(I_VDEFINE, WD_VAR, EMP_RECORD.DEPTNUMBER,
  I_CHAR, STG(EMP_RECORD.DEPTNUMBER));
CALL ISPLINK(I_VDEFINE, WN_VAR, EMP_RECORD.DEPTNAME,
  I_CHAR, STG(EMP_RECORD.DEPTNAME));
CALL ISPLINK(I_VDEFINE, FI_VAR, FNNAMEI,
  I_CHAR, STG(FNNAMEI));
CALL ISPLINK(I_VDEFINE, LI_VAR, LNNAMEI,
  I_CHAR, STG(LNNAMEI));
CALL ISPLINK(I_VDEFINE, MSGS_VAR, MSGS,
  I_CHAR, STG(MSGS));

/* MAIN PROGRAM */
SEL_EXIT = '0'B;  /* INITIALIZE EXIT BIT */
DO WHILE (^SEL_EXIT);  /* DO WHILE NOT EXIT */
  CALL ISPLINK(I_DISPLAY, SEL_PANEL);
  MSGS = ' ';  /* RESET THE MSG FIELD */
  OUTMSG = ' ';  /* RESET THE MSG FIELD */
  SEL_EXIT = (PLIRETV = 8);  /* SEL_EXIT = TRUE IF RC=8 */
/* EXIT WAS NOT SPECIFIED SO PROCESS THE REQUEST */

IF ^SEL_EXIT THEN /* IF USER PRESSED ENTER */
DO;
   DIS_TABLE = NO; /* INIT FLAG TO NO */
   CALL ISPLINK(I_VGET, SEL_VARS);
   LNAMES = LNAMEI; /* COPY INPUT TO WORKING VAR */
   FNAMES = FNAMEI; /* COPY INPUT TO WORKING VAR */
   CALL GET_TYPE; /* DETERMINE LIST TYPE */
   CALL GET_LIST; /* GET LIST OF EMPLOYEES */
   IF DIS_TABLE THEN /* END IF USER PRESSED ENTER */
      CALL DISPLAY_LIST;
   END;
   CALL ISPLINK(I_VPUT, MSGS_VAR); /* SET PANEL MESSAGE */
END; /* END DO WHILE NOT EXIT */

CALL PLIRETC(ZERO); /* SET EXIT RETURN CODE TO 0 */
RETURN;

/* GET TYPE OF LIST */

GET_TYPE: PROCEDURE;
   IF LNAMES = '*' THEN /* LIST DIRECTORY */
      TYPE = 'ALL';
   ELSE
      IF INDEX(LNAMES, '%') > 0 THEN /* GENERIC LIST */
         TYPE = 'GENERIC';
         LNAMES = TRANSLATE(LNAMES, '%', ' '); /* CHG SPACES TO % */
         FNAMES = TRANSLATE(FNAMES, '%', ' '); /* CHG SPACES TO % */
      END; /* END IF GENERIC */
      ELSE /* SPECIFIC NAME */
         TYPE = 'SPECIFIC';
         FNAMES = TRANSLATE(FNAMES, '%', ' '); /* CHG SPACES TO % */
      END; /* END IF SPECIFIC */
   END GET_TYPE;

/* GET LIST OF EMPLOYEES */

GET_LIST: PROCEDURE;
   SQLERRP = ' '; /* CONNECTION CHECK: INIT SQLERRP */
   SELECT (TYPE); /* OPEN CURSOR & GET FIRST RECORD */
      WHEN ('ALL') /* FOR ALL EMPLOYEES */
         DO;
            EXEC SQL OPEN TELE1;
            EXEC SQL FETCH TELE1 INTO :PPHONE;
         END;
      WHEN ('GENERIC') /* FOR GENERIC EMPLOYEES */
      DO;
      END;
      OTHERWISE /* FOR SPECIFIC EMPLOYEE(S) */
      DO;
      END;
      END; /* SELECT */

/* NO EMPLOYEE FULFILLED THE REQUEST */

SELECT;
   WHEN (SQLERRP = ' ') /* NO CONNECTION TO DB2 */
      DO;
         CALL DSNBMPG (MODULE, '079E', OUTMSG);
MSGS = OUTMSG;                   /* SET ISPF ERROR MESSAGE */
END;                               /* END NO EMPLOYEE FOUND */
WHEN (SQLCODE = 100)
  DO;
    CALL DSN8MPG (MODULE, '008I', OUTMSG);
    MSGS = OUTMSG;                   /* SET ISPF ERROR MESSAGE */
    END;                               /* END NO EMPLOYEE FOUND */
  OTHERWISE
/***************************************************************************/
/* EMPLOYEES EXIST THAT FULFILL THE REQUEST. DISPLAY THEM.                 */
/***************************************************************************/
  DO;                                   /* BUILD RESULT TABLE     */
    DIS_TABLE = YES;
    CALL ISPLINK(I_TBCREATE, TABLE_NAME, ' ', EMP_VARS, I_NOWRITE, I_REPLACE);
    DO WHILE (SQLCODE = 0);               /* WHILE MORE ENTRIES   */
      EMP_RECORD = PPHONE, BY NAME;
      CALL ISPLINK(I_TBADD, TABLE_NAME);  /* ADD TO ISPF TABLE    */
      SELECT (TYPE);                      /* GET NEXT RECORD      */
        WHEN ('ALL')
          EXEC SQL FETCH TELE1 INTO :PPHONE;
        WHEN ('GENERIC')
          EXEC SQL FETCH TELE2 INTO :PPHONE;
        OTHERWISE
          EXEC SQL FETCH TELE3 INTO :PPHONE;
      END;                                /* END SELECT           */
    END;                                  /* END WHILE MORE       */
  END;                                    /* END EMPLOYEE FOUND   */
/***************************************************************************/
/* CLOSE THE CURSORS                                                */
/***************************************************************************/
  SELECT (TYPE);
    WHEN ('ALL')
      EXEC SQL CLOSE TELE1;
    WHEN ('GENERIC')
      EXEC SQL CLOSE TELE2;
    OTHERWISE
      EXEC SQL CLOSE TELE3;
  END;
END GET_LIST;
1/********************************************************************/
/* DISPLAY/UPDATE EMPLOYEE PHONE NUMBERS                            */
/********************************************************************/
DISPLAY_LIST: PROCEDURE;
  EXEC SQL WHENEVER SQLERROR CONTINUE;  /* CHANGE ERROR HANDLING  */
  EXEC SQL WHENEVER SQLWARNING CONTINUE;  /* FOR UPDATE             */
  CALL ISPLINK(I_TBTOP, TABLE_NAME);
  CALL ISPLINK(I_TBDISPL, TABLE_NAME, DIS_PANEL);
  DIS_EXIT = (PLIRETV = 8);               /* WAS EXIT PRESSED?      */
  IF ^DIS_EXIT THEN                       /* IF EXIT NOT PRESSED    */
    DO;
      CALL ISPLINK(I_VGET, DIS_VARS);
      MORE_CHANGES = (ROWS_CHANGED > 0);  /* ANY CHANGES?           */
      DO WHILE(MORE_CHANGES);              /* FIND PHONE NUM UPDATES */
        EXEC SQL UPDATE VEMPLP            /* PERFORM UPDATE         */
          SET PHONENUMBER = :EMP_RECORD.PHONENUMBER
          WHERE EMPLOYEENUMBER = :EMP_RECORD.EMPLOYEENUMBER;
        IF SQLCODE ^= 0 THEN              /* IF UPDATE FAILED       */
          DO;
            CALL DSN8MPG(MODULE, '060E', OUTMSG);
            SQL_PIC = SQLCODE;
            MSGS = SUBSTR(OUTMSG,1,46) || SQL_PIC;
            EXEC SQL ROLLBACK;
            MORE_CHANGES = NO;
          END;                            /* END UPDATE FAILED      */
        ELSE                              /* SUCCESSFUL UPDATE      */
          DO;
            CALL DSN8MPG(MODULE, '004I', OUTMSG);
            MSGS = OUTMSG;
            CALL ISPLINK(I_TBPUT, TABLE_NAME);
          END;                           /* DISPLAY CHANGES        */
CALL ISPLINK(I_TBDISPL, TABLE_NAME);
CALL ISPLINK(I_VGET, DIS_VARS);
END;
ELSE                          /* NO MORE CHANGES        */
  MORE_CHANGES = NO;
END;                          /* END SUCCESSFUL UPDATE */
END;                          /* DO WHILE MORE CHANGES  */
END;                                           /* END IF ^DIS_EXIT    */

END DISPLAY_LIST;

1/*********************************************************************/
/* ERROR HANDLING                                                   */
 /*********************************************************************/
P3_DBERROR:
  CALL DSN8MPG(MODULE, '060E', OUTMSG);  /* GET FULL MSG TEXT     */
  SQL_PIC = SQLCODE;
  MSGS = SUBSTR(OUTMSG,1,46) || SQL_PIC; /* APPEND SQL CODE        */
  CALL ISPLINK(I_VPUT, MSGS_VAR);        /* PUT MSG OUT            */
  RETURN;                                 /* EXIT PROGRAM           */
END DSN8SP3;

Related reference
“Sample applications in TSO” on page 1027
A set of Db2 sample applications run in the TSO environment.

DSN8EP1
PASS DB2 COMMANDS TO BE EXECUTED BY THE STORED PROCEDURE PROGRAM DSN8EP2.

DSN8EP1: PROCEDURE(PARMS) OPTIONS(MAIN);

//*********************************************************/
/* MODULE NAME = DSN8EP1 (SAMPLE PROGRAM) */
/* DESCRIPTIVE NAME = STORED PROCEDURE REQUESTER PROGRAM */
/* LICENSED MATERIALS - PROPERTY OF IBM */
/* 5675-DB2 */
/* (C) COPYRIGHT 1982, 2000 IBM CORP. ALL RIGHTS RESERVED. */
/* STATUS = VERSION 7 */
/* FUNCTION = PASS DB2 COMMANDS TO BE EXECUTED BY THE STORED */
/* PROCEDURE PROGRAM DSN8EP2. GET INPUT FROM 'SYSIN'. */
/* PASS THE COMMAND AND RECEIVE THE COMMAND RESULTS */
/* VIA THE PARAMETERS CONTAINED IN THE EXEC SQL CALL */
/* STATEMENT. WRITE THE RESULTS TO 'SYSPRINT'. */
/* DEPENDENCIES = NONE */
/* RESTRICTIONS = */
/* 1. BEGIN DB2 COMMANDS WITH A HYPHEN AND END THEM */
/* WITH A SEMICOLON. A '*' IN COLUMN ONE OR '-' */
/* ANYWHERE ON A LINE (EXCEPT WITHIN A COMMAND) CAN */
/* BE USED TO DENOTE COMMENTS. */
/* 2. THIS PROGRAM ACCEPTS COMMANDS OF AT MOST 4096 BYTES. */
/* PROGRAM SIZES = */
/* THE FOLLOWING VARIABLES CAN BE CHANGED TO FIT THE */
/* SPECIFIC ENVIRONMENT OF THE USER. */
/* VARIABLE VALUE MEANING */
/* NAME:          */
/* PAGEWIDTH 133  MAXIMUM WIDTH OF A PAGE IN */
/* CHARACTERS (INCLUDING THE CONTROL */
/* CHARACTER IN COLUMN ONE) */

Chapter 12. Sample data and applications supplied with Db2 for z/OS 1101
* MAXPAGWD 125 PRINT LINE WIDTH CONTROLLER = + 00450000
* MAXIMUM WIDTH - 1 (FOR CONTROL CHARACTER) - 6 (LENGTH OF THE
* COLUMN DISPLAY) - 1 ( A '-' BETWEEN THE COLUMN NUMBER DISPLAY
* THE SQL OUTPUT DISPLAY). + 00490000
* MAXPAGLN 60 MAXIMUM NUMBER OF LINES ON THE
* PRINT OUTPUT PAGES 2 THRU N. PAGE + 00520000
* 1 WILL HAVE MAXPAGLN + 1 LINES. + 00540000
* INPUT 72 LENGTH OF THE INPUT RECORD + 00550000
* INPUT = + 00560000
* * 1. INPUT STATEMENTS WILL BE TRANSFERRED + 00600000
* TO THE STATEMENT BUFFER WITH ONE BLANK BETWEEN
* WORDS. + 00610000
* * 2. BLANKS IN DELIMITED STRINGS WILL BE + 00630000
* TRANSFERRED INTO THE STATEMENT BUFFER + 00650000
* EXACTLY AS THEY APPEAR IN THE INPUT + 00660000
* STATEMENT. + 00670000
* * 3. AN INPUT LINE CONSISTS OF CHARACTERS FROM + 00680000
* COLUMNS 1-INPUT. IF AN INPUT STATEMENT SPANS + 00690000
* OVER MULTIPLE LINES, THE LINES ARE CONCATENATED + 00700000
* AND BLANKS ARE REMOVED SUCH THAT ONLY ONE + 00710000
* BLANK OCCURS BETWEEN WORDS. + 00720000
* * MODULE TYPE = PROCEDURE + 00740000
* PROCESSOR = + 00750000
* ADMF PRECOMPILED + 00760000
* PL/I MVS/VM (FORMERLY PL/I SAA AD/Cycle) + 00770000
* MODULE SIZE = 2K + 00780000
* ATTRIBUTES = RE-ENTERABLE + 00790000
* * ENTRY POINT = DSN8EPI + 00800000
* PURPOSE = SEE FUNCTION + 00810000
* LINKAGE = STANDARD MVS PROGRAM INVOCATION, ONE PARAMETER. + 00820000
* INPUT = PARAMETERS EXPLICITLY PASSED TO THIS FUNCTION: + 00830000
* SYMBOLIC LABEL/NAME = SYSIN + 00840000
* DESCRIPTION = DDNAME OF SEQUENTIAL DATA SET CONTAINING + 00850000
* DB2 COMMANDS TO BE EXECUTED. + 00860000
* OUTPUT = PARAMETERS EXPLICITLY RETURNED: + 00870000
* SYMBOLIC LABEL/NAME = SYSPRINT + 00880000
* DESCRIPTION = DDNAME OF SEQUENTIAL OUTPUT DATA SET TO + 00890000
* CONTAIN RESULTS OF THE COMMANDS EXECUTED. + 00900000
* * EXIT-NORMAL = + 00910000
* * NO ERRORS WERE FOUND IN THE SOURCE AND NO + 00920000
* ERRORS OCCURRED DURING PROCESSING. + 00930000
* * NORMAL MESSAGES = + 00940000
* * 1. THE FOLLOWING MESSAGE WILL BE GENERATED FOR ALL INPUT + 00950000
* STATEMENTS: + 00960000
* ***INPUT STATEMENT: DB2 COMMAND INPUT STATEMENT + 00970000
* * ERRORS WERE FOUND IN THE SOURCE, OR OCCURRED DURING + 00980000
* PROCESSING. + 00990000
* * RETURN CODE = 4 - WARNING-LEVEL ERRORS DETECTED. + 01000000
* SQLWARNING OR IFI WARNING FOUND DURING EXECUTION. + 01010000
* REASON CODE = 0 OR IFI REASON CODE + 01020000
* * RETURN CODE = 8 - ERRORS DETECTED. + 01030000
* SQLError OR IFI ERROR FOUND DURING EXECUTION. + 01040000
* REASON CODE = 0 OR IFI REASON CODE + 01050000
* * RETURN CODE = 12 - SEVERE ERRORS DETECTED. + 01060000
* ONE OF THE FOLLOWING ERRORS OCCURRED: + 01070000
* UNABLE TO OPEN FILES. + 01080000
* INTERNAL ERROR, ERROR MESSAGE ROUTINE RETURN CODE. + 01090000
* STATEMENT IS TOO LONG. + 01100000
* SQL OR IFI BUFFER OVERFLOW. + 01110000

1102 Db2 12 for z/OS: Application Programming and SQL Guide
*  **REASON CODE = 0 OR IFI REASON CODE**  *  01279000
*  **ABEND CODES = NONE**  *  01280000
*  **ERROR MESSAGES =**  *  01300000
*  **1. THE FOLLOWING MESSAGE WILL BE GENERATED WHEN A DB2**  *  01300000
*  **COMMAND DOES NOT BEGIN WITH A HYPHEN "-".**  *  01300000
*  ***** SYNTAX FOR DB2 COMMAND IS NOT VALID.**  *  01300000
*  **A VALID COMMAND MUST BEGIN WITH A HYPHEN "-".**  *  01300000
*  **2. THE FOLLOWING MESSAGE WILL BE GENERATED WHEN AN INPUT**  *  01300000
*  **STATEMENT IS GREATER THAN STMTMAX SIZE:**  *  01300000
*  ****ERROR: DB2 COMMAND GREATER THAN NNN CHARACTERS.**  *  01300000
*  **STMT:**  *  01300000
*  **DB2 COMMAND.**  *  01300000
*  **NNU IS MAXIMUM COMMAND SIZE**  *  01300000
*  **DB2 COMMAND IS THE CURRENT DB2 COMMAND BEING**  *  01300000
*  **PROCESSED.**  *  01300000
*  **EXTERNAL REFERENCES =**  *  01300000
*  **ROUTINES/SERVICES = NONE**  *  01300000
*  **DSNBIAR - SQL COMMUNICATION AREA FORMATTING**  *  01300000
*  **DATA-AREAS = NONE**  *  01300000
*  **CONTROL-BLOCKS = SQLCA - SQL COMMUNICATION AREA**  *  01300000
*  **PSEUDOCODE =**  *  01300000
*  **DSNAME: PROCEDURE.**  *  01300000
*  **DECLARATIONS.**  *  01300000
*  **INITIALIZE VARIABLES.**  *  01300000
*  **CALL READRTN TO READ IN A DB2 COMMAND STATEMENT.**  *  01300000
*  **DO UNTIL END-OF-FILE.**  *  01300000
*  **CALL READRTN TO READ A NEW DB2 COMMAND STATEMENT.**  *  01300000
*  **END.**  *  01300000
*  **HEX2CHAR: PROCEDURE.**  *  01300000
*  **CONVERT THE RETURN CODE AND REASON CODE THAT ARE RETURNED**  *  01300000
*  **FROM THE IFI CALL FROM BINARY TO HEXADECIMAL.**  *  01300000
*  **END HEX2CHAR.**  *  01300000
*  **PRINTCA: PROCEDURE.**  *  01300000
*  **CALL DSNBIAR TO FORMAT ANY MESSAGES.**  *  01300000
*  **IF A RETURN CODE WAS PASSED FROM DSNBIAR, INDICATE IT.**  *  01300000
*  **PRINT THE DATA FORMATTED FORMATTED BY DSNBIAR.**  *  01300000
*  **SET THE RETURN CODE TO 8.**  *  01300000
*  **END PRINTCA.**  *  01300000
*  **READRTN: PROCEDURE.**  *  01300000
*  **SET ENDDR = "NO".**  *  01300000
*  **SET REREAD = "NO".**  *  01300000
*  **DO WHILE (ENDDR = NO).**  *  01300000
*  **FILL THE STATEMENT BUFFER FROM THE CURRENT INPUT LINE.**  *  01300000
*  **TERMINATE A STATEMENT WHEN A SEMICOLON IS FOUND.**  *  01300000
*  **VERIFY THAT COMMAND IS VALID.**  *  01300000
*  **DO SQL TO CALL DSNAME.**  *  01300000
*  **PROCESS THE COMMAND RESULTS.**  *  01300000
*  **SET REREAD FLAG.**  *  01300000
*  **RETURN TO CALLER.**  *  01300000
*  **END COMMAND.**  *  01300000
*  **END READRTN.**  *  01300000
*  **RESULTS: PROCEDURE.**  *  01300000
*  **PROCESS THE RETURN CODE, REASON CODE, THE NUMBER OF**  *  01300000
*  **BYTES IN THE RETURN BUFFER, AND THE RETURN BUFFER**  *  01300000
*  **THAT ARE RETURNED FROM THE IFI CALL.**  *  01300000
*  **END RESULTS.**  *  01300000
*  **CHANGE ACTIVITY =**  *  01300000
*  **6/29/95 UPDATED THE REMOTE LOCATION NAME VARIABLES (DB2LOC &**  *  01300000
*  **PARMS) TO ACCEPT A SIXTEEN CHARACTER NAME.**  *  01300000
*  **7/05/95 CHANGED THE OUTPUT STRING LENGTH FROM VARYING**  *  01300000
*  **& FIXED 80 BYTE STRINGS (PN69303).**  *  01300000
*  **8/28/95 ADDED ROLLBACK WORK STATEMENT TO ENSURE THAT DB2**  *  01300000
*  **WORK ISROLLED BACK IN ERROR SITUATIONS (PN74842).**  *  01300000
1104  Db2 12 for z/OS: Application Programming and SQL Guide
Chapter 12. Sample data and applications supplied with Db2 for z/OS 1105
PROCEDURE(INPUT) RETURNS(CHAR(8)); /* RESULTS RETURNED IN CHARACTER FORMAT */
DECLARE INPUT BIT(31),
/* RETURN CODE IN BINARY */
I1 BIT(4) DEF INPUT,
I2 BIT(4) DEF INPUT POSITION(4),
I3 BIT(4) DEF INPUT POSITION(8),
I4 BIT(4) DEF INPUT POSITION(12),
I5 BIT(4) DEF INPUT POSITION(16),
I6 BIT(4) DEF INPUT POSITION(20),
I7 BIT(4) DEF INPUT POSITION(24),
I8 BIT(4) DEF INPUT POSITION(28),
HEXES CHAR(16) INIT('0123456789ABCDEF'),
OUTPUT CHAR(8),
OUTPUT1(8) CHAR(1) DEFINED(OUTPUT);
OUTPUT1(1)=SUBSTR(HEXES,I1+1,1); /* 1ST BYTE OF RET CODE IN HEX */
OUTPUT1(2)=SUBSTR(HEXES,I2+1,1); /* 2ND BYTE OF RET CODE IN HEX */
OUTPUT1(3)=SUBSTR(HEXES,I3+1,1); /* 3RD BYTE OF RET CODE IN HEX */
OUTPUT1(4)=SUBSTR(HEXES,I4+1,1); /* 4TH BYTE OF RET CODE IN HEX */
OUTPUT1(5)=SUBSTR(HEXES,I5+1,1); /* 5TH BYTE OF RET CODE IN HEX */
OUTPUT1(6)=SUBSTR(HEXES,I6+1,1); /* 6TH BYTE OF RET CODE IN HEX */
OUTPUT1(7)=SUBSTR(HEXES,I7+1,1); /* 7TH BYTE OF RET CODE IN HEX */
OUTPUT1(8)=SUBSTR(HEXES,I8+1,1); /* 8TH BYTE OF RET CODE IN HEX */
RETURN (OUTPUT); /* RETURN THE OUTPUT RESULT */
END HEX2CHAR;
%PAGE;
/* PROCEDURE TO PRINT THE IFI RETURN CODE IN HEX */
PROCEDURE(INPUT) RETURNS(CHAR(8)); /* RESULTS RETURNED IN CHARACTER FORMAT */
DECLARE INPUT BIT(31),
/* RETURN CODE IN BINARY */
I1 BIT(4) DEF INPUT,
I2 BIT(4) DEF INPUT POSITION(4),
I3 BIT(4) DEF INPUT POSITION(8),
I4 BIT(4) DEF INPUT POSITION(12),
I5 BIT(4) DEF INPUT POSITION(16),
I6 BIT(4) DEF INPUT POSITION(20),
I7 BIT(4) DEF INPUT POSITION(24),
I8 BIT(4) DEF INPUT POSITION(28),
HEXES CHAR(16) INIT('0123456789ABCDEF'),
OUTPUT CHAR(8),
OUTPUT1(8) CHAR(1) DEFINED(OUTPUT);
OUTPUT1(1)=SUBSTR(HEXES,I1+1,1); /* 1ST BYTE OF RET CODE IN HEX */
OUTPUT1(2)=SUBSTR(HEXES,I2+1,1); /* 2ND BYTE OF RET CODE IN HEX */
OUTPUT1(3)=SUBSTR(HEXES,I3+1,1); /* 3RD BYTE OF RET CODE IN HEX */
OUTPUT1(4)=SUBSTR(HEXES,I4+1,1); /* 4TH BYTE OF RET CODE IN HEX */
OUTPUT1(5)=SUBSTR(HEXES,I5+1,1); /* 5TH BYTE OF RET CODE IN HEX */
OUTPUT1(6)=SUBSTR(HEXES,I6+1,1); /* 6TH BYTE OF RET CODE IN HEX */
OUTPUT1(7)=SUBSTR(HEXES,I7+1,1); /* 7TH BYTE OF RET CODE IN HEX */
OUTPUT1(8)=SUBSTR(HEXES,I8+1,1); /* 8TH BYTE OF RET CODE IN HEX */
RETURN (OUTPUT); /* RETURN THE OUTPUT RESULT */
END HEX2CHAR;
%PAGE;
/* PROCEDURE TO PRINT THE SQLCA ERROR INDICATION AND CLEAR OUT THE */
/* SQLCA. OUTPUT MOST OF THE DATA ON AN EXCEPTION BASIS */
PROCEDURE;
PRINTCA: PROCEDURE;
CALL DSNTIAR (SQLCA, MESSAGE, MAXPAGWD); /* FORMAT ANY MESSAGES */
IF PLIRETV ^= ZERO THEN /* IF THE RETURN CODE ISN'T ZERO */
    DO; /* ISSUE AN ERROR MESSAGE */
        PUT EDIT ( ' *** RETURN CODE ', PLIRETV, /*@35*/
                  ' FROM MESSAGE ROUTINE DSNTIAR.' )
        (COL(1), A(17), F(8), A(30)); /* ISSUE THE MESSAGE */
        RETCODE = SEVERE; /* SET THE RETURN CODE */
    END; /* END ISSUE AN ERROR MESSAGE */
DO I = ONE TO MSGBLEN /* PRINT OUT THE DSNTIAR BUFFER */
WHILE (MESSAGET(I) ^= BLANK); /* PRINT NON BLANK LINES */
    PUT EDIT ( MESSAGET(I) ) (COL(2), A(MAXPAGWD));
END;
RETCODE = SEVERE; /* SET THE RETURN CODE */
END PRINTCA;

%PAGE;

READRTN: PROCEDURE;

DCL CONTLINE FIXED BIN(15) /* CONTINUATION LINE - INPUT STMT */
             INIT(0), /* IS MORE THAN 72 CHARACTERS */
             DQUOTFLAG BIT(1) /* DOUBLE QUOTE (") ENCOUNTERED? */
                    INIT('0'B),
FIRSTCHAR BIT(1) /* FIRST NON BLANK CHAR? */
              INIT('0'B),
LASTCHAR CHAR(1) /* LAST CHARACTER IN THE BUFFER */
             INIT(' '),
MOVECHAR BIT(1) /* MOVE CHAR INTO STMT BUFFER? */
              INIT('0'B),
NBLK FIXED BIN(15) /* NUMBER OF BLANKS FOUND */
             INIT( 0 ),
NEWOFSET FIXED BIN(15) /* FIRST POSITION OF THE COMMAND */
             INIT( 0 ),
NEWSTMT BIT(1) /* NEW STMT TO BE PROCESSED? */
              INIT('0'B),
QUOTEFLAG BIT(1) /* QUOTE ('') ENCOUNTERED? */
               INIT('0'B);

ON ENDFILE(SYSIN) /* PROCESS EOF ON INPUT FILE */
    BEGIN; /* END OF FILE */
    IF LENGTH(STMTBUF) = 0 THEN /* LENGTH(STMTBUF) = 0 */
        DO; /* LENGTH(STMTBUF) = 0 */
            EXIT = YES; /* NO STMT TO PROCESS */
            GOTO ENDRD; /* SO END THE PROGRAM */
        END; /* PROCESS THE CURRENT STATEMENT */
    ELSE /* PROCESS THE CURRENT STATEMENT */
        DO; /* LENGTH(STMTBUF) ^= 0 */
            EODIN = YES; /* SIGNAL END_OF_DATA */
            ENDSR = YES; /* SIGNAL END_OF_STRING */
            GOTO CHKCOMM; /* PROCESS CURRENT COMMAND */
        END; /* END LENGTH(STMTBUF) ^= 0 */
    END; /* END END OF FILE */
END;

/* ENDFILE CONDITIONS */

ON ENDFILE(SYSIN) /* PROCESS EOF ON INPUT FILE */
    BEGIN; /* END OF FILE */
    IF LENGTH(STMTBUF) = 0 THEN /* LENGTH(STMTBUF) = 0 */
        DO; /* LENGTH(STMTBUF) ^= 0 */
            EXIT = YES; /* NO STMT TO PROCESS */
            GOTO ENDRD; /* SO END THE PROGRAM */
        END; /* PROCESS THE CURRENT STATEMENT */
    ELSE /* PROCESS THE CURRENT STATEMENT */
        DO; /* LENGTH(STMTBUF) ^= 0 */
            EODIN = YES; /* SIGNAL END_OF_DATA */
            ENDSR = YES; /* SIGNAL END_OF_STRING */
            GOTO CHKCOMM; /* PROCESS CURRENT COMMAND */
        END; /* END LENGTH(STMTBUF) ^= 0 */
    END; /* END END OF FILE */
END;

/* BEGIN READRTN PROCESSING */

NEWSTMT= YES; /* NEW STMT IS BEING PROCESSED */

%PAGE;

/* READ IN THE INPUT STATEMENT */
DO WHILE (NEWSTMT = YES);
/* No more input data (EOF) so return to caller */
dbName = RD;
/* End of data */
EXIT = YES;
/* Exit program */
LEAVE RD;
/* End end of data */

IF EODIN = YES THEN
DO;
  EXIT = YES;
  LEAVE RD;
END;
/* End end of data */

ELSE
  /* More input to process */
  DO;
    CONTLINE = ZERO;
    ENDSTR = NO;
    QUOTEFLAG = NO;
    DQUOTFLAG = NO;
    STMTLEN = ZERO;
    STMTBUF = NULLCHAR;
    LASTCHAR = NULLCHAR;
    NBLK = ZERO;
    NEWSTMT = NO;
    COMMENT = NO;
    FIRSTCHAR = NO;
    endif
  endwhile
  DO
    GET EDIT (INPUT) (COL(1), (INPUTL) A(1));
    INCOL = ONE;
    IF FIRSTCHAR = YES THEN
      CONTLINE = CONTLINE + 1;
    END;
    endif
  endwhile
  IF INCOL > INPUTL THEN
    DO;
      IF INCOL = 1 & (INPUT(1) = ASTERISK
        | (INPUT(1) = HYPHEN & INPUT(2) = HYPHEN)) & STMTLEN = 0 THEN
        IF INCOL = 1 & (INPUT(1) = ASTERISK
          | (INPUT(1) = HYPHEN & INPUT(2) = HYPHEN))
          & STMTLEN = 0 THEN
            DO J = 1 TO INPUTL; /* Put entire line into STMTBUF */
              STMTBUF = STMTBUF || INPUT(J);
            END;
            STMTBUF = STMTBUF || INPUT(J);
            EXIT = YES;
            LEAVE RD;
            END;
            endwhile
          END;
        END;
      ELSE
        endif
      endif
    endwhile
    GET EDIT (INPUT) (COL(1), (INPUTL) A(1));
    INCOL = ONE;
    IF FIRSTCHAR = YES THEN
      CONTLINE = CONTLINE + 1;
    END;
    endif
  endwhile
END;
/* End end of data */

end.
/* MOVE THE CHARACTER FROM THE INPUT DATA INTO THE STATEMENT BUFFER UNTIL AN END OF LINE CHARACTER OR SEMICOLON IS ENCOUNTERED */

DO J = INCOL TO INPUTL WHILE (^ENDSTR);

/* PREPROCESS ANY DOUBLE QUOTATION MARKS ("). IF THE " QUOTATION MARK IS CONTAINED BETWEEN DOUBLE QUOTATION MARKS ("), THE QUOTATION MARK IS NOT CONSIDERED TO BE THE STRING DELIMITER. THE QUOTATION MARK WILL NOT BE SET. IN THIS CASE THE STRING IS CONSIDERED TO BE PART OF THE STATEMENT */

IF INPUT(J) = DQUOTE THEN
  DO;
    IF ^QUOTEFLAG THEN /* NOT DELIMITED BY QUOTES THEN */
      QUOTEFLAG = ^QUOTEFLAG; /* THE DELIMITER */
    END;
  END;

/* PREPROCESS ANY QUOTATION MARKS ('). IF THE QUOTATION MARK IS CONTAINED BETWEEN DOUBLE QUOTATION MARKS ("), THE DOUBLE QUOTATION MARK IS CONSIDERED TO BE THE STRING DELIMITER. THE QUOTEFLAG WILL NOT BE SET. IN THIS CASE THE QUOTATION MARK IS CONSIDERED TO BE PART OF THE STRING. */

IF INPUT(J) = QUOTE THEN
  DO;
    IF ^DQUOTFLAG THEN /* NOT DELIMITED BY DOUBLE QUOTES THEN */
      QUOTEFLAG = ^QUOTEFLAG; /* THE DELIMITER */
    END;
  END;

/* PROCESS A HYPHEN IF FOUND. THE HYPHEN IS CONSIDERED PART OF A STRING IF A DELIMITER FLAG IS SET. IF THE FOLLOWING CHARACTER IS A HYPHEN, MOVE THE REMAINING CHARACTERS TO THE STATEMENT BUFFER. */

IF (INPUT(J) = HYPHEN) & (J < INPUTL) & ^QUOTEFLAG & ^DQUOTFLAG THEN
  DO;
    IF INPUT(J+1) = HYPHEN THEN DO; MOVECHAR = NO; /* INTO THE STATEMENT BUFFER*/
      STMTBUF = STMTBUF || INPUT(J);
    END;
    DO J = INPUTL + ONE; /* TO FORCE THE NEXT STATEMENT */
      STMTBUF = STMTBUF || INPUT(J);
    END;
    ENDSTR = YES; /* INDICATE END OF A STRING */
    NEWSTMT = YES; /* NEW STMT SHOULD BE READ */
    MOVECHAR = NO; /* INTO THE STATEMENT BUFFER*/
    STMTLEN = LENGTH(STMTBUF);
    END;
  END;

END;
IF (INPUT(J) = SEMICOLON) & ^DQUOTFLAG & ^QUOTEFLAG THEN/* SEMICOLON & NOT */ ENDSTR = ^ENDSTR; /* DELIMITED THEN SET END */ /* OF STRING */ /* NOT THE END OF THE STRING, PROCESS THE STATEMENT */ /* DELIMITED THEN SET END */

ELSE
  DO;
  /* MOVE ALL NON BLANK CHARACTERS INTO THE DB2 */ /* COMMAND STATEMENT BUFFER */
  IF INPUT(J)^= BLANK THEN
    DO;
      MOVECHAR = YES;
      FIRSTCHAR = YES;
      NBLK = ZERO;
    END;
  ELSE
    DO;
      IF QUOTEFLAG | DQUOTFLAG | (CONTLINE >= 1 & J = 1 & NBLK = 0) THEN
        DO; /* BLANK IS DELIMITED, MOVE */
          MOVECHAR = YES; /* IT INTO STMT BUFFER*/
          NBLK = NBLK + ONE; /* & INC BLANK COUNT */
        END;
      ELSE
        DO;
          NBLK = NBLK + ONE; /* INCREASE BLANK CTR */
          IF (NBLK = ONE) & (FIRSTCHAR = YES) THEN
            MOVECHAR = YES;
          ELSE
            DO;
              MOVECHAR = NO;
            END;
        END;
      END;
    END;
  END;
ELSE /* BLANK CHARACTER FOUND */
  IF MOVECHAR = YES THEN
    DO;
    /* WHEN THE STATEMENT LENGTH IS TOO LONG, THE STATEMENT CANNOT BE PROCESSED. A RETURN CODE IS SET TO INDICATE NO FURTHER PROCESSING SHOULD BE DONE. AN ERROR MESSAGE WILL BE PUT OUT. */
    STMTLEN = LENGTH(STMTBUF);
    IF STMTLEN = STMTMAX THEN /* STMT TOO LONG */
      DO;
        RETCODE = SEVERE; /* SET RETURN CODE */
        PUT EDDIT(' *** ERROR: STATEMENT GREATER THAN ',STMTMAX,' CHARACTERS. STMT: ')(COL(1),A(31),A(5),F(4),A(13),A(7)); /* @35*/
        PUT EDDIT((SUBSTR(STMTBUF,KK,MIN(100,STMTLEN-KK+1))) /* @35*/
          (COL(1),A(9),A(13),A(7)));
        END;
      END;
  END;
ELSE /* BLANK NOT DELIMITED */
  IF QUOTEFLAG | DQUOTFLAG | (CONTLINE >= 1 & J = 1 & NBLK = 0) THEN
    DO; /* BLANK IS DELIMITED, MOVE */
      MOVECHAR = YES; /* IT INTO STMT BUFFER*/
      NBLK = NBLK + ONE; /* & INC BLANK COUNT */
    END;
  ELSE
    DO;
      NBLK = NBLK + ONE; /* INCREASE BLANK CTR */
      IF (NBLK = ONE) & (FIRSTCHAR = YES) THEN
        MOVECHAR = YES;
      ELSE
        DO;
          MOVECHAR = NO;
        END;
      END;
    END;
  END;
ELSE /* END BLANK NOT DELIMITED */
  IF MOVECHAR IS SET THEN MOVE THE INPUT CHARACTER INTO STATEMENT BUFFER AREA /* CHARACTER INTO STATEMENT BUFFER AREA */
ELSE /* END BLANK CHARACTER FOUND */
  IF MOVECHAR = YES THEN
    DO;
    /* WHEN THE STATEMENT LENGTH IS TOO LONG, THE STATEMENT CANNOT BE PROCESSED. A RETURN CODE IS SET TO INDICATE NO FURTHER PROCESSING SHOULD BE DONE. AN ERROR MESSAGE WILL BE PUT OUT. */
    STMTLEN = LENGTH(STMTBUF);
    IF STMTLEN = STMTMAX THEN /* STMT TOO LONG */
      DO;
        RETCODE = SEVERE; /* SET RETURN CODE */
        PUT EDDIT(' *** ERROR: STATEMENT GREATER THAN ',STMTMAX,' CHARACTERS. STMT: ')(COL(1),A(31),A(5),F(4),A(13),A(7)); /* @35*/
        PUT EDDIT((SUBSTR(STMTBUF,KK,MIN(100,STMTLEN-KK+1))) /* @35*/
DO KK = 1 TO STMTLEN BY 100) /* END STAR */

LEAVE RD;
END; /* END COMMAND */

STMTBUF = STMTBUF || INPUT(J); /* MOVE CHARACTER INTO BUFFER */

LASTCHAR = INPUT(J); /* SAVE THIS CHARACTER */
END; /* END CHARACTER NOT A SEMICOLON */

END; /* END DO WHILE (ENDSTR = NO) */

IF END = 0 THEN /* NO HYPHEN */

DO;

CHKCOMM:

IF ^COMMENT THEN

DO;

OSTMTLN = OSTMTLN + ONE;

IF OSTMTLN * INPLLEN ^= STMTLEN THEN /* INPUT STMT */

OSTMTLN = OSTMTLN + ONE;

NEWOFSET = ONE;

STMTLEN = LENGTH(STMTBUF);

END;

END;

END;

LEAVE RD; /* LEAVE PROGRAM */

Chapter 12. Sample data and applications supplied with Db2 for z/OS 1111
EXEC SQL CONNECT TO :DB2LOC2; /* CONNECT TO REMOTE LOCATION */

CALL PRINTCA; /* PRINT ERROR MSG */
GOTO STOPRUN; /* END PROGRAM */

CALL RESULTS; /* PROCESS THE RESULTS */
NEWOFSET = ZERO; /* RESET CHARACTER PTR */
NEWSTMT = YES; /* RESET FOR NEW STMT */

RESULTS: PROCEDURE;

IFCA_RET_CODE = HEX2CHAR(IFCA_RET_HEX); /* RETURN CODE IN HEX */
IFCA_RES_CODE = HEX2CHAR(IFCA_RES_HEX); /* REASON CODE IN HEX */
TOTBYTES = 0; /* INITIALIZE COUNTER */
BEGINST = 1; /* INITIALIZE POINTER */

IF IFCA_RET_HEX ^= 0 THEN  /* IF THE RETURN CODE ISN'T ZERO */
   DO; /* ISSUE AN ERROR MESSAGE */
      PUT EDIT(’ *** RETURN CODE=’,SUBSTR(IFCA_RET_CODE,7,2), /*@35*/
               ‘ ’); /* PRINT MESSAGE */
      IF IFCA_RES_CODE = ‘-911’ | IFCA_RES_CODE = ‘-918’ | IFCA_RES_CODE = ‘-965’ THEN /* CHECK FOR SPECIFIC ERRORS */
         DO; /* YES, ROLL BACK REQUIRED */
            CALL PRINTCA; /* PRINT ERROR MSG */
            PUT EDIT(’ *** ISSUE ROLLBACK WORK’, /*@35*/
                     ‘BECAUSE STORED PROCEDURE CALL NOT SUCCESSFUL’) /* PRINT ROLLBACK WORK MESSAGE */;
            EXEC SQL ROLLBACK WORK; /* EXECUTE ROLLBACK */
         END; /* END ROLL BACK REQUIRED */
      END; /* END ELSE MORE INPUT */
   END; /* END DO WHILE NEW STMT */

ENDRD:; /* END RD SUB-PROC */
END READRTN; /* END READRTN PROC */

%PAGE;
Related reference

“Sample applications in TSO” on page 1027

A set of Db2 sample applications run in the TSO environment.

DSN8EP2

USING THE INSTRUMENTATION FACILITY INTERFACE (IFI), PROCESS A Db2 COMMAND WHICH IS PASSED FROM THE DSN8EP1 REQUESTER PROGRAM.

DSN8EP2:  PROCEDURE(INPUTCMD, IFCA_RET_HEX, IFCA_RES_HEX, BUFF_OVERFLOW, RETURN_BUFF, NULL_ARRAY) OPTIONS(MAIN NOEXECOPS);

*******************************************************************************/
/*   MODULE NAME = DSN8EP2 (SAMPLE PROGRAM)                                  */
/*   DESCRIPTIVE NAME = STORED PROCEDURE SERVER PROGRAM                       */
/*******************************************************************************/

STOPRUN: IF RETCODE >= SEVERE THEN /* SET PL/I RETURN CODE AND TERMINATE PROCESSING */
  CALL PLIRETC(RETCODE); /* SET PL/I RETURN CODE */
END DSN8EP1; /* END PROGRAM */
* LICENSED MATERIALS - PROPERTY OF IBM  * 00100000
* 5675-DB2  * 00110000
* (C) COPYRIGHT 1982, 2000 IBM CORP. ALL RIGHTS RESERVED.  * 00120000
*  * 00130000
* STATUS = VERSION 7  * 00140000
*  * 00150000
* FUNCTION =  * 00160000
* USING THE INSTRUMENTATION FACILITY INTERFACE (IFI),  * 00170000
* PROCESS A DB2 COMMAND WHICH IS PASSED FROM THE DSNBEP1  * 00180000
* REQUESTER PROGRAM.  * 00190000
*  * 00200000
* NOTES =  * 00210000
* DEPENDENCIES = NONE  * 00220000
*  * 00230000
* RESTRICTIONS =  * 00240000
* 1. THE INSTRUMENTATION FACILITY COMMUNICATION AREA  * 00250000
* (IFCA) CONTAINS INFORMATION REGARDING THE SUCCESS  * 00260000
* OF THE CALL AND PROVIDES FEEDBACK.  * 00270000
* THIS AREA MUST BE MAINTAINED TO INCLUDE ANY CHANGES  * 00280000
* TO THE MAPPING MACRO DSNDFCA.  * 00290000
*  * 00300000
* MODULE TYPE = PROCEDURE  * 00310000
* PROCESSOR =  * 00320000
* ADMF PRECOMPILER  * 00330000
* PL/I MVS/VM (FORMERLY PL/I SAA AD/CYCLE)  * 00340000
* MODULE SIZE = 1K  * 00350000
* ATTRIBUTES = RE-ENTERABLE  * 00360000
*  * 00370000
* ENTRY POINT = DSNBEP2  * 00380000
* PURPOSE = SEE FUNCTION  * 00390000
* LINKAGE = INVOKED VIA EXEC SQL CALL.  * 00400000
* INPUT = PARAMETERS EXPLICITLY PASSED TO THIS FUNCTION:  * 00410000
* SYMBOLIC LABEL/NAME = INPUTCMD  * 00420000
* DESCRIPTION = DB2 COMMAND TO BE PROCESSED BY IFI.  * 00430000
* INPUT STATEMENTS FROM THE INPUTCMD  * 00440000
* PARAMETER WILL BE PASSED TO THE  * 00450000
* TEXT_OR_COMMAND FIELD OF THE OUTPUT_AREA  * 00460000
* IN THE DSNBEP1 PROGRAM.  * 00470000
*  * 00480000
* OUTPUT = PARAMETERS EXPLICITLY RETURNED:  * 00490000
* SYMBOLIC LABEL/NAME = IFCA_RET_HEX  * 00500000
* SYMBOLIC LABEL/NAME = IFCA_RES_HEX  * 00510000
* DESCRIPTION = COMMUNICATION AREAS BETWEEN THE  * 00520000
* APPLICATION PROGRAM AND IFI  * 00530000
* APPLICATION PROGRAM AND IFI  * 00540000
* SYMBOLIC LABEL/NAME = RETURN_BUFF  * 00550000
* DESCRIPTION = DB2 COMMAND RESPONSE FROM IFI  * 00560000
* THE RETURN CODE, REASON CODE, AND THE  * 00570000
* BYTES MOVED FROM THE IFCA AND THE  * 00580000
* RTRN_BUFF FIELD FROM THE IFI RETURN AREA  * 00590000
* WILL BE PASSED VIA THE IFCA_RET_HEX,  * 00600000
* IFCA_RES_HEX, IFCA_BYTES_MOVED, AND  * 00610000
* RETURN_BUFF PARAMETERS.  * 00620000
*  * 00630000
* EXIT NORMAL =  * 00640000
* NO ERRORS WERE FOUND IN THE SOURCE AND NO  * 00650000
* ERRORS OCCURRED DURING PROCESSING.  * 00660000
*  * 00670000
* NORMAL MESSAGES =  * 00680000
*  * 00690000
* EXIT-ERROR =  * 00700000
* ERRORS WERE FOUND IN THE SOURCE, OR OCCURRED DURING  * 00710000
* PROCESSING.  * 00720000
*  * 00730000
* RETURN CODE = 4 - WARNING-LEVEL ERRORS DETECTED.  * 00740000
* WARNING FOUND DURING EXECUTION.  * 00750000
* REASON CODE = NONE OR FROM IFI  * 00760000
*  * 00770000
* RETURN CODE = 8 - ERRORS DETECTED.  * 00780000
* ERROR FOUND DURING EXECUTION.  * 00790000
* REASON CODE = NONE OR FROM IFI  * 00800000
*  * 00810000
* RETURN CODE = 12 - SEVERE ERRORS DETECTED.  * 00820000
* UNABLE TO OPEN FILES.  * 00830000
* INTERNAL ERROR, ERROR MESSAGE ROUTINE RETURN CODE.  * 00840000
* STATEMENT IS TOO LONG.  * 00850000
* BUFFER OVERFLOW.  * 00860000
* REASON CODE = NONE OR FROM IFI  * 00870000
*  * 00880000
* ABEND CODES =  * 00890000
*  * 00900000
* ERROR MESSAGES =  * 00910000
Chapter 12. Sample data and applications supplied with Db2 for z/OS 1115

/*
 * 05/22/03  FIX CODE HOLE CLOSED BY VA AND ENTERPRISE PL/I PQ44916
 *            REASON CODE=00E60804 FROM IFI
 * 7/05/95  CHANGED THE OUTPUT STRING LENGTH FROM VARYING
 * TO FIXED 80 BYTE STRINGS
 * 04/17/00  INITIALIZE STORAGE TO PREVENT RETURN CODE=04,
 * REASON CODE=00E608B4 FROM IFI
 * 05/22/03  FIX CODE HOLE CLOSED BY VA AND ENTERPRISE PL/I PQ44916

*********************************************************************/

*DCL COMMAND CHAR(8) INIT(' '); /* USER SPECIFIED DB2 COMMAND */

%PAGE;

*DCL

 ADDR BUILTIN, /* ADDRESS OF A DATA AREA */
 LENGTH BUILTIN, /* LENGTH OF A STRING */
 MOD BUILTIN, /* RETURNS MODULO VALUE */
 STORAGE BUILTIN, /* FUNCTION TO GET SOME SPACE */
 SUBSTR BUILTIN, /* FUNCTION TO RETURN SUBSTRING */
 UNSPEC BUILTIN; /* IGNORES VARIABLE TYPING */

; /* DECLARATION FOR INPUT AND OUTPUT PARAMETERS */

*DCL

 INPUTCMD CHAR(4096) VARYING, /* DB2 COMMAND */
 FUNCTION CHAR(8) INIT(' '), /* FUNC PARM FOR IFI */
 IFCA_RET_HEX FIXED BIN(31), /* IFI RETURN CODE */
 IFCA_RES_HEX FIXED BIN(31), /* IFI REASON CODE */
 BUFF_OVERFLOW FIXED BIN(31), /* RETURN BUFF BYTES */
 NULL_ARRAY(5) FIXED BIN(15), /* INDICATOR ARRAY */
 RETURN_BUFF CHAR(8320) VARYING, /* PASSED BUFFER */
 RETURN_LEN FIXED BIN(15) INIT(8320); /* LENGTH */

; /* WORKING VARIABLES */

*DCL

 REMBYTES FIXED BIN(15) INIT(0), /* NUM BYTES TO BE */
 CMDLEN FIXED BIN(15) INIT(0), /* NUM BYTES IN A */
 01 FIXED_BUFF BASED(ADDR(RETURN_BUFF)), /* OVERLAY OF PASSED BUF */
 02 FIXED_LEN FIXED BIN(15), /* MOVING DATA FROM RETURN AREA */
 02 FIXED_TEXT CHAR(4160), /* TO MAKE RECORD LENGTHS IN */

; /* OVERLAY OF PASSED BUF FOR */

*DCL

 FILLBYTES FIXED BIN(15) INIT(0), /* NUMBER OF FILL BYTES NEED */
 NUMFULL FIXED BIN(15) INIT(0), /* NUMBER OF FULL LINES IN */
 PARTROW FIXED BIN(15) INIT(0), /* LENGTH OF NON-FULL LINE */
 J FIXED BIN(15) INIT(0), /* LOOP COUNTER */
 BUFPOSIX FIXED BIN(15) INIT(0), /* POSITION IN RETURN BUFFER */
 01 FIXED_BUFF FIXED BIN(15) INIT(0), /* POSITION IN PASSED BUFFER */
 LEN_CHAR CHAR(2), /* LENGTH BYTES IN COMMAND */

; /* RESULT STRING */

*********************************************************************/

*DCL

 NULL_ARRAY(5) FIXED BIN(15), /* INDICATOR ARRAY */
 RETURN_BUFF CHAR(8320) VARYING, /* PASSED BUFFER */
 RETURN_LEN FIXED BIN(15) INIT(8320); /* LENGTH */

; /* OF PASSED BUFFER */

*DCL

 INPUTCMD CHAR(4096) VARYING, /* DB2 COMMAND */
 FUNCTION CHAR(8) INIT(' '), /* FUNC PARM FOR IFI */
 IFCA_RET_HEX FIXED BIN(31), /* IFI RETURN CODE */
 IFCA_RES_HEX FIXED BIN(31), /* IFI REASON CODE */
 BUFF_OVERFLOW FIXED BIN(31), /* RETURN BUFF BYTES */
 NULL_ARRAY(5) FIXED BIN(15), /* INDICATOR ARRAY */

; /* DECLARATION FOR INPUT AND OUTPUT PARAMETERS */

*DCL

 ADDR BUILTIN, /* ADDRESS OF A DATA AREA */
 LENGTH BUILTIN, /* LENGTH OF A STRING */
 MOD BUILTIN, /* RETURNS MODULO VALUE */
 STORAGE BUILTIN, /* FUNCTION TO GET SOME SPACE */
 SUBSTR BUILTIN, /* FUNCTION TO RETURN SUBSTRING */
 UNSPEC BUILTIN; /* IGNORES VARIABLE TYPING */

; /* DECLARATION FOR INPUT AND OUTPUT PARAMETERS */

*DCL

 INPUTCMD CHAR(4096) VARYING, /* DB2 COMMAND */
 FUNCTION CHAR(8) INIT(' '), /* FUNC PARM FOR IFI */
 IFCA_RET_HEX FIXED BIN(31), /* IFI RETURN CODE */
 IFCA_RES_HEX FIXED BIN(31), /* IFI REASON CODE */
 BUFF_OVERFLOW FIXED BIN(31), /* RETURN BUFF BYTES */
 NULL_ARRAY(5) FIXED BIN(15), /* INDICATOR ARRAY */

; /* DECLARATION FOR INPUT AND OUTPUT PARAMETERS */
LEN_BIT  BIT(16) /* LENGTH IN BITS */@47+ 01730001
LEN_BIN  FIXED BIN(15) /* LENGTH IN BINARY */@47+ 01750000
SPACE_LEFT  FIXED BIN(15) /* BYTES LEFT IN BUFFER */@47+ 01770000

DCL 01 OUTPUT_AREA, 02540000
       CHAR(80) INIT(' '); 02510000
       DIAGNOS_DATA     /* DIAGNOSTIC DATA. */ 02500000
       FIXED BIN(15) INIT(0), 02490000
       UNUSED_4                                                02480000
       DIAGNOS_LNGTH    /* DIAGNOSTIC LENGTH. */ 02460000
       DIAGNOS_AREA,     /* DIAGNOSTIC AREA. */ 02450000
       CHAR(2)       INIT(' '), 02440000
       ARRAY_TRACE_NOS(8)/* AREA FOR TRACE NUMBERS RETURNED */ 02430000
       FIXED BIN(15) INIT(0), 02420000
       UNUSED_3                                                02410000
       TRACE_LNGTH      /* LENGTH OF TRACE NO.S RETURNED + 4*/ 02390000
       TRACE_NOS_AREA,     /* AREA CONTAINING UP TO 8 TRACE #'S */ 02380000
       CHAR(4)       INIT(' '), 02370000
       ARRAY_OPN_NAMES(8) /* AREA FOR OPN NAMES RETURNED    */ 02360000
       FIXED BIN(15) INIT(0), 02350000
       UNUSED_2                                                02340000
       OPN_LNGTH        /* LENGTH OF OPN NAMES RETURNED + 4.*/ 02320000
       OPN_NAME_FOR_READA  /* OPN NAME USED FOR READA REQUEST  */ 02300000
       FIXED BIN(31) INIT(0), 02280000
       NUM_RECDS_LOST      /* RECORDS LOST INDICATOR.          */ 02270000
       FIXED BIN(31) INIT(0), 02250000
       BYTES_MOVED         /* BYTES OF RECORD WHICH WERE MOVED.*/ 02210000
       IFCARC2             /* REASON CODE FOR THE IFC API CALL.*/ 02190000
       IFCARC1             /* RETURN CODE FOR THE IFC API CALL.*/ 02170000
       OWNER_ID            /* TO ESTAB OWNERSHIP OF AN OPN DEST*/ 02150000
       EYE_CATCHER         /* USED TO VERIFY THE IFCA BLOCK.   */ 02130000
       LNGTH               /* LENGTH OF IFCA, INCL LENGTH FIELD*/ 02090000
DCL 01 IFCA,
   02 LNGTH /* LENGTH OF IFCA, INCL LENGTH FIELD*/@47+ 02090000
   02 FIXED BIN(15) INIT(0), 02100000
   02 UNUSED 02110000
   02 FIXED BIN(15) INIT(0), 02120000
   02 EYE_CATCHER /* USED TO VERIFY THE IFCA BLOCK. */@47+ 02130000
   02 CHAR(4) INIT('IFCA'), 02140000
   02 OWNER_ID /* TO ESTAB OWNERSHIP OF AN OPN DEST*/@47+ 02150000
   02 CHAR(4) INIT(' '), 02160000
   02 IFCARC1 /* RETURN CODE FOR THE IFC API CALL. */@47+ 02170000
   02 FIXED BIN(31) INIT(0), 02180000
   02 IFCARC2 /* REASON CODE FOR THE IFC API CALL. */@47+ 02190000
   02 FIXED BIN(31) INIT(0), 02200000
   02 BYTES_MOVED /* BYTES OF RECORD WHICH WERE MOVED.*/@47+ 02210000
   02 FIXED BIN(31) INIT(0), 02220000
   02 EXCESS_RECS /* BYTES OF RECORD WHICH DID NOT FIT*/@47+ 02230000
   02 FIXED BIN(31) INIT(0), 02240000
   02 OPN_WRIT_SEQ_NUM /* LAST OPN WRTR SEQ# FOR READA FUNC*/@47+ 02250000
   02 FIXED BIN(31) INIT(0), 02260000
   02 NUM_RECS_LOST /* RECORDS LOST INDICATOR. */@47+ 02270000
   02 FIXED BIN(31) INIT(0), 02280000
   02 OPN_NAME_FOR_READA /* OPN NAME USED FOR READA REQUEST */@47+ 02290000
   02 CHAR(4) INIT(' '), 02300000
   02 OPN_NAMES_AREA,   /* AREA CONTAINING UP TO 8 OPN NAMES*/@47+ 02310000
   02 OPN_NAME_LENGTH /* LENGTH OF OPN NAMES RETURNED + 4.*/@47+ 02320000
   02 FIXED BIN(15) INIT(0), 02330000
   02 UNUSED_2 02340000
   02 FIXED BIN(15) INIT(0), 02350000
   02 ARRAY_OPN_NAM E(8) /* AREA FOR OPN NAMES RETURNED */@47+ 02360000
   02 CHAR(4) INIT(' '), 02370000
   02 TRACE_NOS_AREA,   /* AREA CONTAINING UP TO 8 TRACE #'S*/@47+ 02380000
   02 TRACE_LENGTH /* LENGTH OF TRACE NO.S RETURNED + 4.*/@47+ 02390000
   02 FIXED BIN(15) INIT(0), 02400000
   02 UNUSED_3 02410000
   02 FIXED BIN(15) INIT(0), 02420000
   02 ARRAY_TRACE_NOS(8)/* AREA FOR TRACE NUMBERS RETURNED */@47+ 02430000
   02 CHAR(2) INIT(' '), 02440000
   02 DIAGNOS_AREA /* DIAGNOSTIC AREA. */@47+ 02450000
   02 DIAGNOS_LENGTH /* DIAGNOSTIC LENGTH. */@47+ 02460000
   02 IMPORTED BIN(15) INIT(0), 02470000
   02 UNUSED_4 02480000
   02 URLs(15) INIT(0), 02490000
   02 DIAGNOS_DATA /* DIAGNOSTIC DATA. */@47+ 02500000
   02 CHAR(80) INIT(' '), 02510000
DCL 01 OUTPUT_AREA,
   02 LNGTH /* LENGTH OF APPL PGM REC TO WRITE */@47+ 02530000
Chapter 12. Sample data and applications supplied with Db2 for z/OS 1117
SUBSTR(FIXED_BUFF.FIXED_TEXT,BUFPOSO,BUFROWLN) = /*@47*/ 03370000
SUBSTR(RETURN_AREA.RTRN_BUFF,BUFPOSI,BUFROWLN); /*@47*/ 03380000
BUFPOSO = BUFPOSO + BUFROWLN; /* MOVE PAST STRG IN OUTP@47*/ 03390000
BUFPOSI = BUFPOSI + BUFROWLN; /* MOVE PAST STRG IN INPT@47*/ 03400000
REMBYTES = REMBYTES - BUFROWLN; /* CALCULATE BYTES LEFT@47*/ 03410000
END;                                                   /*@47*/ 03420000
IF PARTROW > 0 THEN                                      /*@47*/ 03430000
DO;                             /* MOVE PARTIAL LINE     @47*/ 03440000
SUBSTR(FIXED_BUFF.FIXED_TEXT,BUFPOSO,PARTROW) =      /*@47*/ 03450000
SUBSTR(RETURN_AREA.RTRN_BUFF,BUFPOSI,PARTROW);     /*@47*/ 03460000
BUFPOSI = BUFPOSI + PARTROW;  /* MOVE PAST STR IN INPUT@47*/ 03470000
BUFPOSO = BUFPOSO + PARTROW - 1; /* MOVE TO END OF     @47*/ 03480000
/* STRING IN OUTPUT   @47*/ 03490000
SUBSTR(FIXED_BUFF.FIXED_TEXT,BUFPOSO,1) = BLANK;     /*@47*/ 03500000
/* REPLACE THE NEW LINE  @47*/ 03510000
/* CHARACTER IN THE LAST */ 03520000
BUFPOSO = BUFPOSO + 1;        /* MOVE PAST STRG IN OUTP@47*/ 03530000
REMBYTES = REMBYTES - PARTROW; /* CALCULATE BYTES LEFT @47*/ 03540000
END;                                                   /*@47*/ 03550000
IF PARTROW > 0 THEN               /* FILL UP SPACE WITH BLK@47*/ 03560000
DO;                                                    /*@47*/ 03570000
DO J = BUFPOSO TO (BUFPOSO + FILLBYTS - 1);          /*@47*/ 03580000
SUBSTR(FIXED_BUFF.FIXED_TEXT,J,1) = ' ';            /*@47*/ 03590000
END;                                                 /*@47*/ 03600000
BUFPOSO = BUFPOSO + FILLBYTS; /* MOVE PAST BLANKS      @47*/ 03610000
END;                                                   /*@47*/ 03620000
END;                              /* COPY TEXT TO PASSED BUF @47*/ 03630000
FIXED_BUFF.FIXED_LEN = BUFPOSO - 1;                        /*@47*/ 03640000
/* GET BYTES IN PASSED BUF @47*/ 03650000
END;                                /* IF ANYTHING TO COPY     @47*/ 03660000
END DSN8EP2;                          /* END PROGRAM                */ 03670000

Related reference
“Sample applications in TSO” on page 1027
A set of Db2 sample applications run in the TSO environment.

DSN8EPU
PASS Db2 UTILITY STATEMENTS TO BE EXECUTED BY THE STORED PROCEDURE PROGRAM DSNUTILS.

DSN8EPU: PROC OPTIONS(MAIN);                                           00010000
/%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%****  00020000
*   MODULE NAME = DSN8EPU (SAMPLE PROGRAM)                         *  00030000
*                                                                  *  00040000
*   DESCRIPTIVE NAME = STORED PROCEDURE REQUESTER PROGRAM          *  00050000
*                                                                  *  00060000
*    LICENSED MATERIALS - PROPERTY OF IBM                          *  00070000
*    5625-DB2                                                      *  0008001
*    (C) COPYRIGHT 1992, 2003 IBM CORP.                            *  0009001
*                                                                  *  00100000
*    STATUS = VERSION 8                                            *  00110001
*                                                                  *  00120000
*   FUNCTION =                                                     *  00130000
*                                                                  *  00140000
*      PASS DB2 UTILITY STATEMENTS TO BE EXECUTED BY THE STORED    *  00150000
*      PROCEDURE PROGRAM DSNUTILS.  GET INPUT FROM 'SYSIN'.        *  00160000
*      PASS THE STATEMENT AND RECEIVE THE OUTPUT RESULTS           *  00170000
*      VIA A RETURNED CURSOR.  WRITE THE RESULTS TO 'SYSPRINT'.    *  00180000
*                                                                  *  00190000
*      DEPENDENCIES = NONE                                         *  00200000
*                                                                  *  00210000
*      RESTRICTIONS =                                              *  00220000
*                                                                  *  00230000
*      INPUT =                                                     *  00240000
*                                                                  *  00250000
*      1. INPUT MUST BE OF THE FORM                                 *  00260000
*                                                                  *  00270000
*      Uid='',Restart='',Utility='REORG TABLESPACE',               *  00280000
*      CopyDSN1='SYSADM.COPYDDN.DSN8D51A.DSN8S51E',                *  00290000
*      CopyDev1='SYSDA',CopySpace1=10,                             *  00300000
*      Utstmt=                                                     *  00310000
*      ' REORG TABLESPACE DSN8D51A.DSN8S51E                       *  00320000
*      SORTDev SYSDA SORTNUM 2                                      *  00330000
*      SHRLEVEL CHANGE                                             *  00340000
*      DEADLINE 2010-2-4-03.15.00                                 *  00350000
*      MAXRO 240 LONGLOG DRAIN DELAY 900                           *  00360000
END DSN8EPU2; /* END PROGRAM */ 03680000
Chapter 12. Sample data and applications supplied with Db2 for z/OS

DCL UTSTMT CHAR(32704) VARYING;
DCL RESTART CHAR(8) VARYING; /* RESTART */
DCL UID CHAR(16) VARYING; /* UTILITY ID */
DCL NULL BUILTIN;
DCL ADDR BUILTIN;

DCL UTIL_OPTS_BUFF VARYING CHAR(32760) INIT( '' );

SYSIN_EOF = '1'B;
ON ENDFILE( SYSIN )
DCL SYSIN_EOF BIT( 01 ) INIT( '0'B );

SEQ_NOS CHAR( 08 );
UTIL_OPTS CHAR( 72 ),

DCL SYSIN FILE INPUT STREAM ENV( F RECSIZE(80) );
DCL SYSPRINT FILE OUTPUT STREAM;

EXEC SQL FETCH FROM RESULT SET.
DO UNTIL SQLCODE > 0.
EXEC SQL CALL SYSPROC.DSNUTILS.
INITIALIZE VARIABLES.
DECLARATIONS.

DSN8EPU: PROCEDURE.

PSEUDOCODE =

DSNBEPU: PROCEDURE.
DECLARATIONS.
INITIALIZE VARIABLES.
EXEC SQL CALL SYSPROC.DSNUTILS.
DO UNTIL SQLCODE > 0.
EXEC SQL FETCH FROM RESULT SET.
PRINT RESULT SET TO SYSPRINT.
END.

NOTICE =
THIS SAMPLE PROGRAM USES DB2 UTILITIES. SOME UTILITY FUNCTIONS ARE ELEMENTS OF SEPARATELY ORDERABLE PRODUCTS. SUCCESSFUL USE OF A PARTICULAR SAMPLE MAY BE DEPENDENT UPON THE OPTIONAL PRODUCT BEING LICENSED AND INSTALLED IN YOUR ENVIRONMENT.

CHANGE ACTIVITY =

* PQ24728 - Add FILTRDSN and Fix I/O for seq qed input
* PQ44916 - Fix code hole closed by VA and Enterprise PL/I
* d54292 - Check for unexpected SQLCODE in FETCH loop

*****************************/

CALL SYSPROC.DSNUTILS( :UID, :RESTART, /* Call DSNUTILS stored procedure to process the inputted settings */
01890000
01880000
GET STRING( UTIL_OPTS_BUFF ) DATA; /* Assign DSNUTILS options from inputted settings in UTIL_OPTS_BUFF */
01870000
GET COPY EDIT( UTIL_OPTS,SEQ_NOS )( A(72),A(8) ); UTIL_OPTS_BUFF = UTIL_OPTS_BUFF || UTIL_OPTS;
01860000
DO WHILE( ^SYSIN_EOF ); /* Collect DSNUTILS options from SYSIN records, columns 1-72 */
01850000
FiltrDSN=''; FiltrDEVT=''; FiltrSPACE=0; ErrDSN=''; ErrDEVT=''; ErrSpace=0; MapDSN=''; MapDEVT=''; MapSpace=0; WorkDSN2=''; WorkDEVT2=''; WorkSpace2=0; RcpyDSN2=''; RcpyDEVT2=''; RcpySpace2=0; CopyDSN2=''; CopyDEVT2=''; CopySpace2=0; PnchDSN=''; PnchDEVT=''; PnchSpace=0; RecDSN=''; RecDEVT=''; RecSpace=0; Utility=''; RetCode = 0; Utstmt='';
01840000
EXEC SQL INCLUDE SQLCA;
01830000
DCL TEXT CHAR(122) VARYING;
01820000
DCL SEQNO FIXED BIN(31);
01810000
DCL RESULTS SQL TYPE IS RESULT_SET_LOCATOR VARYING;
01800000
FILTRSPACE FIXED BIN(15);
01790000
DCL ERRDSN     CHAR(44) VARYING,
01780000
ERRDEVT    CHAR(8),
01770000
ERRSPACE   FIXED BIN(15),
01760000
DCL MAPDSN     CHAR(44) VARYING,
01750000
MAPDEVT    CHAR(8),
01740000
MAPSPACE   FIXED BIN(15),
01730000
DCL RCPYDSN1   CHAR(44) VARYING,
01720000
RCPYDEVT1  CHAR(8),
01710000
RCPYSPACE1 FIXED BIN(15),
01700000
DCL COPYDSN2 CHAR(44) VARYING,
01690000
COPYDEVT2  CHAR(8),
01680000
COPYSPACE2 FIXED BIN(15),
01670000
COPYDSN1=''; COPYDEVT1=''; COPYSPACE1=0; RCPYDSN1=''; RCPYDEVT1=''; RCPYSPACE1=0; COPYDSN2=''; COPYDEVT2=''; COPYSPACE2=0; RCPYDSN2=''; RCPYDEVT2=''; RCPYSPACE2=0;
Related reference

“Sample applications in TSO” on page 1027

A set of Db2 sample applications run in the TSO environment.

**DSN8ED1**

Pass Db2 commands received from standard input to stored procedure DSN8ED2 for execution.

```sql
EXEC SQL
ASSOCIATE LOCATOR (:RESULTS) WITH PROCEDURE SYSPROC.DSNUTILS;
IF SQLCODE < 0 THEN
DO;
PUT SKIP EDIT('ASSOCIATE LOCATOR SQLCA')(A);
PUT SKIP DATA(SQLCA);
CALL PLIRETC(8);    /* SET PLI RETURN CODE */
RETURN;
END;
EXEC SQL
ALLOCATE SYSPRINT CURSOR FOR RESULT SET :RESULTS;
IF SQLCODE < 0 THEN
DO;
PUT SKIP EDIT('ALLOCATE SYSPRINT SQLCA')(A);
PUT SKIP DATA(SYSPRINT);
CALL PLIRETC(8);    /* SET PLI RETURN CODE */
END;
FETCHLOOP:
DO UNTIL(SQLCODE < 0 | SQLCODE = 100 );
EXEC SQL
FETCH SYSPRINT INTO :SEQNO, :TEXT;
IF (SQLCODE >= 0) & (SQLCODE ^= 100) THEN
DO;
PUT SKIP EDIT(TEXT)(A);
END;
IF (SQLCODE ^= 0) & (SQLCODE ^= 100) THEN
DO;
CALL PLIRETC('FETCH SYSPRINT SQLCA')(A);
PUT SKIP DATA(SYSPRINT);
END;
END FETCHLOOP;
IF SQLCODE < 0 THEN
DO;
CALL PLIRETC(RetCode);    /* SET PLI RETURN CODE */
END DSN8EPU;
```
/* Function = */
/* Pass DB2 commands received from standard input to */
/* stored procedure DSN8ED2 for execution. Receive the */
/* command results from DSN8ED2 as result set. Unload the */
/* result set and print the contents to standard output. */
/* Dependencies = None */
/* Restrictions = */
/* 1. DB2 commands must be preceded by a hyphen and */
/* followed by a semicolon. Lines with an asterisk */
/* in the first column or two hyphens as the first */
/* nonblank characters are interpreted as comment */
/* lines. Two hyphens placed after the command text in */
/* in a line indicate that the rest of the line is */
/* comments only. */
/* 2. A command may be no more than 4096 bytes. */
/* Input = */
/* 1. A single input parameter that indicates the location */
/* of the stored procedure. The contents must be a */
/* valid DB2 location name of at most 16 characters. */
/* 2. Lines of length INPUTL from standard input. */
/* Only the first INPUTSED bytes are used. Lines are */
/* considered to be either command text or comments. */
/* Command text begins with a hyphen and ends with a */
/* semicolon. Comments begin with an asterisk in */
/* column one or two hyphens as the first two nonblank */
/* characters. Command text may span lines, but comment */
/* text may not. */
/* Output = */
/* Lines of length OUTLEN to standard output. */
/* Each line contains one of the following: */
/* a. Command text. */
/* b. Results returned by the DB2 for MVS/ESA command */
/* processor after the command is issued. */
/* Module type = C program */
/* Processor = */
/* ADMF Precompiler */
/* C/370 */
/* Module size = See linkedit output */
/* Attributes = Not reentrant or reusable */
/* Entry point = DSN8ED1 */
/* Purpose = See Function */
/* Linkage = Standard MVS program invocation, one parameter. */
/* Exit normal = */
/* Return code 0 on normal completion. */
/* Normal messages = */
/* *** Input statement: <DB2 command input statement text> */
/* *** IFI return area: <Results of DB2 command execution> */
/* Exit-error = */
/* Return code = 4 - Warnings occurred. */
/* - The DB2 for MVS/ESA Instrumentation Facility Interface */
/* (IFI) invocation of the DB2 command resulted in a */
/* return code 4. The accompanying reason code indicates */
/* the specific problem. */
/* Return code = 8 - Errors occurred. */
/* - The DB2 for MVS/ESA Instrumentation Facility Interface */
/* (IFI) invocation of the DB2 command resulted in a */
/* return code 8. The accompanying reason code indicates */
/* the specific problem. */
/* Return code = 12 - Severe errors occurred. */
/* - Input parameter 1 did not contain the name of the */
/* DB2 server where the stored procedure resides. */
/* - The input dataset (SYSIN) did not contain any data. */
/* Command input did not begin with a hyphen. */
/* Command input was not ended with a semicolon. */
/* An input statement contained more than STMTMAX bytes. */
/* Connection to the stored procedure location failed. */
/* The SQL CALL statement to the stored procedure failed. */
/* The DB2 for MVS/ESA Instrumentation Facility Interface */
/* (IFI) invocation of the DB2 command resulted in a */
/* return code 12. The accompanying reason code indicates */
/* the specific problem. */
/* The call to the stored procedure, DSN8ED2, succeeded */
/* but DSN8ED2 experienced SQL problems. The formatted */
/* SQL error message appears in SYSPRINT. */
/* The call to the stored procedure, DSN8ED2, succeeded */
/* but no result set was returned. SYSPRINT messages */
/* should provide more information. */
/* A result set was returned by DSN8ED2 but one of the */
/* following occurred (see SYSPRINT messages for details):*/
/* - The locator variable could not be associated with */
/*   the result set. */
/* - The result set cursor could not be allocated */
/* - No data could be fetched from the result set cursor. */
/* Abend codes = None */

Error messages =
- *** ERROR: No server name provided - DSN8ED1 ended.
- *** ERROR: No input records found - DSN8ED1 ended.
- *** ERROR: Syntax for DB2 command is invalid.
- A valid command ends with a semicolon.
- *** ERROR: Syntax for DB2 command is invalid.
- *** ERROR: Statement length is greater than the ___
  character maximum.
- *** ERROR: Connection to server <location> was unsuccessfull.
- *** ERROR: Call to stored procedure DSN8ED2 failed; diagnoses follow.
- *** ERROR: The following diagnostics were returned by
  stored procedure DSN8ED2.
- *** ERROR: DSNTIAR could not format the message.
  SQLCODE is ____ SQLERRM is ________ ...
- *** WARNING: Call to stored procedure DSN8ED2 succeeded
  but no result set was returned.
- *** WARNING: IFI error codes returned by DSN8ED2.
  Return code=___, reason code=____ from IFI request.
- *** WARNING: ____ records were lost because the IFI
  return area in stored procedure DSN8ED2
  is too small to accomodate this request.
- Increase the IFI return area (RETURN_LEN) in DSN8ED2 and then
  recompile/relink/rebind before resubmitting this request.
- *** Syntax for DB2 command is invalid.
- A valid command must begin with a hyphen.
- *** Statement length is greater than the <STMTMAX>
  character maximum.
- *** Connection to <location> unsuccessful.
- SQLCODE is <sqlcode>.
- *** Call to DSN8ED2 unsuccessful.
- SQLCODE is <sqlcode>.
- *** Insufficient space to receive all output from IFI
  return area.
- *** Return code=<return code>, reason code=<reason code> from IFI request.
- *** Severe error occurred. Program is terminating.

External references =
Routines/services =
  none
Data areas =
  none
Control blocks =
  SQLCA - SQL communication area
Pseudocode =
DSN8ED1:
- Extract the name of the DB2 server where stored procedure
  DSN8ED2 resides from input parameter number 1.
build_DB2_command:
  /* - Read a record from SYSIN */
  /* - Do the following until either a full command is built, end */
  /* of file is reached, or an error occurs: */
  /* - if the first byte of the record is '*' or the first two */
  /* nonblank bytes of the record are '---' then the whole */
  /* record is a comment. Disregard it. */
  /* - if '---' is encountered after nonblanks are found then the */
  /* rest of the record is a comment and can be disregarded. */
  /* - else if a semicolon is found inside a delimited string */
  /* call copy_byte_to_cmd_buf to add it to the command string. */
  /* - a delimited string is one that starts but has not yet */
  /* terminated with a single quote or a double quote */
  /* - else if a semicolon is found outside a delimited string */
  /* then the command is complete. */
  /* - else if a nonblank is found then call copy_byte_to_cmd_buf */
  /* to add it to the command string. */
  /* - else if a blank is found inside a delimited string then */
  /* call copy_byte_to_cmd_buf to add it to the command string. */
  /* - else if a blank is found outside a delimited string and */
  /* the preceding byte was nonblank then call copy_byte_to_*/
  /* cmd_buf to add it to the command string. */
  /* - else if a blank is found outside a delimited string and */
  /* the preceding byte was blank then disregard the blank. */
  /* - if the input record is exhausted before a terminating */
  /* semicolon is found, read the next input record. */
  /* - When a command is created successfully, call echo_DB2_command*/
  /* to output the reformatted command. */
  /* - Check the command to ensure that it starts with a hyphen. */
  End build_DB2_command
  */
  copy_byte_to_cmd_buf
  /* - append the current byte of the input record to the end of */
  /* the command string and update length of command string. */
  /* - if command string exceeds buffer size, issue a message and */
  /* end DSN8ED1. */
  End copy_byte_to_cmd_buf
  /* echo_DB2_command */
  /* - output the reformatted DB2 command to SYSPRINT */
  End echo_DB2_command
  /* connect_to_sp_server */
  /* - invoke SQL to CONNECT to the DB2 server where stored proc-*/
  /* edure DSN8ED2 resides. */
  /* - if the CONNECT fails, issue a message and end DSN8ED1. */
  End connect_to_sp_server
  /* send_DB2_command_to_sp */
  /* - invoke SQL to call stored procedure DSN8ED2 to process the */
  /* contents of the command buffer. */
  /* - analyze the resultant SQLCODE, IFI return and result codes, */
  /* and buffer overload and error parameters returned by DSN8ED2. */
  End send_DB2_command_to_sp
  /* output_results_from_sp */
  /* - associate a DB2 locator variable with the result set from */
  /* stored procedure DSN8ED2 */
  /* - allocate a cursor to the result set */
  /* - fetch each row from the result set and output it to SYSPRINT */
  End output_results_from_sp
  /* sql_error */
  /* - invoke DSNNTAR to format the current SQL code and print the */
  /* messages to SYSPRINT */
  /* - if DSNNTAR cannot detail the code, output the SQLCODE and */
  /* SQLERRM to SYSPRINT */
  End sql_error
/*
*/
/* Change activity =
*/
/*
none
*/
/********************************************************************/
/********************** C library definitions ***********************/
#include
<stdio.h>
#include
<stdlib.h>
#include
<string.h>
/**************************** Constants *****************************/
#define
INPUTL
81
/* Length of input line
*/
#define
INPUSED
72
/* Bytes used in an input line*/
#define
LOCLEN
16
/* Length of input parm (loc) */
#define
OUTLEN
81
/* Length of output line
*/
#define
RETWRN
4
/* Warning return code
*/
#define
RETERR
8
/* Error return code
*/
#define
RETSEV
12
/* Severe error return code
*/
#define
STMTMAX
4096
/* Maximum statement length
*/
#define
ASTERISK
'*'
/* Comment indicator
*/
#define
BLANK
' '
/* Blank
*/
#define
HYPHEN
'-'
/* Hyphen
*/
#define
NULLCHAR
'\0'
/* Null character
*/
#define
QUOTE
'\''
/* Quotation mark
*/
#define
DQUOTE
'"'
/* Double quote
*/
#define
SEMICOLON
';'
/* SQL stmt terminator
*/
enum flag

{No, Yes};

/* Settings for flags

*/

/********************** Program Argument List ***********************/
char *parms[];
/* Contains input parameter
*/
/********************** Standard Input/Output ***********************/
FILE
*sysin;
/* Input statements
*/
char
input[INPUTL];
/* Current input data
*/
char
*inres;
/* Result of gets invocation */
FILE

*sysprint;

/* Command results/error msgs */

/************************ Working variables *************************/
short int
c;
/* pointer to command buffer */
enum flag
dquotflag;
/* '"' delimiter status
*/
short int
i;
/* pointer to input buffer
*/
short int
j;
/* miscellaneous counter, ptr */
enum flag
endstr;
/* End of statement flag
*/
enum flag
input_eof;
/* End of input data flag
*/
enum flag
quoteflag;
/* "'" delimiter status
*/
/************************ DB2 Host Variables ************************/
EXEC SQL BEGIN DECLARE SECTION;
char
db2loc2[17];
/* Remote DB2 location name
*/
long int
ifca_ret_hex;
/* Return code from IFI call */
long int
ifca_res_hex;
/* Reason code from IFI call */
long int
xs_bytes_hex;
/* No. of bytes not returned */
long int
rc;
/* All-purpose return var
*/
struct {
short int
sp_err_blen;
/* Error msg buffer length
*/
char
sp_err_txt[880];
/* Error msg text
*/
}
sp_err_buf;
/* Error message buffer from */
/* stored procedure
*/
short int
sperind1;
/* Indicator vars for parm 1 */
short int
sperind2;
/* Indicator vars for parm 2 */
short int
sperind3;
/* Indicator vars for parm 3 */
short int
sperind4;
/* Indicator vars for parm 4 */
short int
sperind5;
/* Indicator vars for parm 5 */
struct {
short int
cmdlen;
char
cmdtxt[4096];
}
cmdbuf;

/* Statement length
*/
/* Statement text
*/
/* Statement buffer passed to */
/* stored procedure
*/

/* Result set locator
static volatile SQL TYPE IS RESULT_SET_LOCATOR *DSN8ED2_rs_loc;
long int
char

rs_sequence;
rs_data[80];

EXEC SQL END DECLARE SECTION;

*/

/* Result set table data sequ */
/* Result set data buffer
*/
/* - length is OUTLEN - 1
*/

/******************** DB2 SQL Communication Area ********************/
EXEC SQL INCLUDE SQLCA;

Chapter 12. Sample data and applications supplied with Db2 for z/OS 1125


```c
int main( int argc, char *argv[] ) {
  cmdbuf.cmdlen = 0; /* Nothing in command buf yet */
  input_eof = No;  /* Not at end of input */
  rc = 0;         /* No errors yet */
  sperind1 = -1;  /* Clear null indicator var 1 */
  sperind2 = -1;  /* Clear null indicator var 2 */
  sperind3 = -1;  /* Clear null indicator var 3 */
  sperind4 = -1;  /* Clear null indicator var 4 */
  sperind5 = -1;  /* Clear null indicator var 5 */

  for( j=1; j<argc; j++ ) /* break out the input parms */
    parms[j] = argv[j];

  for( j=0; j<LOCLEN; j++ ) /* Extract name of DB2 server */
    db2loc2[j] = *(parms[1]+j); /* where sp resides */
  if( db2loc2[1] == BLANK ) /* If no server specified, */
    { /* issue error */
      printf( " *** ERROR: No server name provided - DSN8ED1 ended.\n" );
      rc = RETSEV;
    }
  db2loc2[j]=NULLCHAR; /* Null-terminate the string */

  if( rc < RETSEV )
    { build_DB2_command();
      if( input_eof == Yes && rc < RETSEV )
        { printf( " *** ERROR: No input records found - DSN8ED1 ended.\n" );
          rc = RETSEV;
        }
      while( input_eof == No && rc < RETSEV )
        { connect_to_sp_server(); /* connect to the server */
          if( rc < RETSEV ) /* if successful */
            { send_DB2_command_to_sp(); /* invoke the stored proc */
              if( rc < RETSEV ) /* if successful */
                { output_results_from_sp(); /* out the results */
                  if( rc < RETSEV ) /* if successful */
                    { build_DB2_command(); /* process the next input */
                      printf( " \n \n *** DSN8ED1 completed; highest return code was %d\n", rc );
                      return( rc ); /* put return code in ctl blk */
                    }
                }
            }
        }
    }
  printf( " \n \n *** DSN8ED1 completed; highest return code was %d\n", rc );
  return( rc ); /* put return code in ctl blk */
}
```
**Build a DB2 command from one or more physical input records**

```c
build_DB2_command() /*proc*/
{
    /* initialize working variables */
    for( i=0; i<INPUTL; i++ ) /* Blank the input array */
        input[i] = ' ';
    c = 0; /* marks pos'n in command buf */
    cmdbuf.cmdlen = 0; /* no. of bytes in cmd buffer */
    dquotflag = No; /* flags delimiter stat of '"'*/
    endstr = No; /* flags end of log inp record*/
    i = 0; /* marks pos'n in input buffer*/
    quoteflag = No; /* flags delimiter stat of '*'*/

    /* read the first physical record of the command from input */
    inres = gets( input );
    if( inres == NULL ) /* If end of file reached */
        input_eof = Yes; /* then all finished */

    /* parse the current input record for DB2 command parts */
    while( endstr == No  &&  input_eof == No  &&  rc < RETSEV )
    {
        /*******************************
        * first char in a line is '*' -OR- the 1st two non-blank chars
        * in a line are '--', this is a comment line. Don't copy it to
        * the command buffer; request next line.
        * *******************************
        if( i == 0  &&  input[0] == ASTERISK )
            i = INPUSED;
        else if( cmdbuf.cmdlen == 0
            && input[0] == HYPHEN  &&  input[1] == HYPHEN )
            i = INPUSED;
        /*******************************
        * Otherwise, this must be a command line. Parse it into the com-
        * mand buffer while looking for delimiters and the end of stmt. 
        * *******************************
        else
            while( i < INPUSED  &&  endstr == No  &&  rc < RETSEV )
            {
                if( input[i] == DQUOTE ) /* if double quote found */
                    if( dquotflag == Yes ) /* and is already a delimiter*/
                        dquotflag = No; /* note end of delimited str*/
                    else if( quoteflag == No ) /* else if it's not delimited*/
                        dquotflag = Yes; /* then it's a delimiter */
                if( input[i] == QUOTE ) /* if single quote found */
                    if( quoteflag == Yes ) /* and is already a delimiter*/
                        quoteflag = No; /* -note end of delimited str*/
                    else if( dquotflag == No ) /* else if it's not delimited*/
                        quoteflag = Yes; /* then it's a delimiter */
                if( input[i] == HYPHEN /* if '...' found in current */
                    && input[i+1] == HYPHEN /* and next byte */
                    && i < INPUSED /* not at end of input line */
                    && quoteflag == No /* not in a delimited */
                    && dquotflag == No /* string then rest is comment*/
                    i = INPUSED; /* ignore it; request next line*/
                else if( input[i] == SEMICOLON/* else if semicolon found */
                    && quoteflag == No /* and it's not delimited */
                    && dquotflag == No /* then command is complete */
                    endstr = Yes; /* fall through */
                else if( input[i] != BLANK ) /* else if non-blank found */
                    copy_byte_to_cmd_buf(); /* copy it to command buffer */
                else if( input[i] == BLANK /* else if blank found */
                    && ( quoteflag == Yes /* and it's in a delimited */
                        || dquotflag == Yes ) /* string */
                    copy_byte_to_cmd_buf(); /* copy it to command buffer*/
                else if( input[i] == BLANK /* else if blank found */
    ```
&& c > 0                    /* and something's in cmd buf*/
&& cmdbuf.cmdtxt[c-1]=BLANK  /* and prev cmd byte nonblank */
copy_byte_to_cmd_buf();      /* copy it to command buffer*/
else;
    /* swallow all other blanks */
i++;
    /* bump pos'n in input record */
}
} /* end while( i<INPUSED && endstr == No && rc<RETSEV ) */

if( i >= INPUSED && endstr == No && rc < RETSEV )
{
    for( i=0; i<INPUTL; i++ )    /* Blank the input array */
        input[i] = '\0';
    i = 0;                        /* reset pointer to input buff*/
inres = gets( input );        /* Read the next physical rec */
if( inres == NULL )            /* If end of file reached */
{
    printf( " *** ERROR: Syntax for DB2 command is invalid.
              A valid command ends with a" );
    printf( " semicolon.\n" );
    rc = RETSEV;                  /* stop the program */
}
} /* end while( endstr == No && input_eof == No && rc < RETSEV ) */

if( cmdbuf.cmdlen > 0 )
    echo_DB2_command();

if( endstr == Yes && input_eof == No && rc < RETSEV )
{
    if( cmdbuf.cmdtxt[0] != HYPHEN )
    {
        printf( " *** ERROR: Syntax for DB2 command is invalid.
                A valid command begins with a hyphen.\n" );
        rc = RETSEV;
    }
} /* end of build_DB2_command() */

/* Copy the current byte of current input line to command buffer */
copy_byte_to_cmd_buf();
{
    cmdbuf.cmdtxt[c++] = input[i];
    cmdbuf.cmdlen = c;
}

if( cmdbuf.cmdlen >= STMTMAX )
{
    printf( " *** ERROR: Statement length is greater than the" );
    printf( " %d character maximum.\n",STMTMAX );
    rc = RETSEV;
}
} /* end of copy_byte_to_cmd_buf() */

connect_to_sp_server()     /*proc*/
{
    EXEC SQL CONNECT TO :db2loc2;
    if( SQLCODE != 0 )
    {

1128 Db2 12 for z/OS: Application Programming and SQL Guide
printf( " *** ERROR: Connection to server %s was unsuccessful.\n", 
db2loc2 );
sql_error( " *** Connection to server unsuccessful" );
rc = RETSEV;
}
} /* end of connect_to_sp_server() */

/******************************************************
** Process the current DB2 command built from the input file **
******************************************************
* send_DB2_command_to_sp()                                      /*proc*/
{
  sperind1 = 0;                          /* tell DB2 to transmit    */
  EXEC SQL CALL DSN8.DSN8ED2(       
              :cmdbuf :sperind1,
              :ifca_ret_hex :sperind2,
              :ifca_res_hex :sperind3,
              :xs_bytes_hex :sperind4,
              :sp_err_buf :sperind5 );

  /*******************************************************************
  * verify the SQL return code returned by the stored procedure      *
  *******************************************************************/
  if( SQLCODE == 0 )
  {
    printf( " *** WARNING: Call to stored procedure DSN8ED2" );
    printf( "           succeeded\n" );
    printf( " but no result set was returned.\n" );
    if( rc < RETERR )
      rc = RETERR;
  } else if( SQLCODE == 466 )
  {
    printf( " *** A result set was returned by stored procedure" );
    printf( " DSN8ED2.\n" );
  } else
  {
    printf( " *** ERROR: Call to stored procedure DSN8ED2 failed;" );
    printf( " diagnostics follow.\n" );
    sql_error( " *** Stored procedure call unsuccessful." );
    rc = RETSEV;
  }

  /*******************************************************************
  * verify the IFI return code returned by the stored procedure      *
  *******************************************************************/
  if( sperind2 != -1  &&  ifca_ret_hex != 0 )
  {
    printf( " *** WARNING: IFI error codes returned by DSN8ED2.\n" );
    printf( "           Return code=%0X ",  ifca_ret_hex );
    printf( " reason code=%0X from IFI request.\n" , ifca_res_hex );
    if( ifca_ret_hex > rc )
      rc = ifca_ret_hex;
  }

  /*******************************************************************
  * if IFI return buffer was too small, output a message             *
  *******************************************************************/
  if( sperind4 != -1  &&  xs_bytes_hex != 0 )
  {
    printf( " *** WARNING: %d bytes were lost",  xs_bytes_hex );
    printf( " because the IFI return area in stored\n" );
    printf( " procedure DSN8ED2 is too small" );
    printf( " to accomodate this request.\n" );
    printf( " *** Increase the IFI return area\n" );
    printf( " (RETURN_LEN) in DSN8ED2 and then\n" );
    printf( " recompile/relink/rebind before\n" );
    if( rc < RETWRN )
      rc = RETWRN;
  }

  /*******************************************************************
  * output any data from the error message buffer                    *
  *******************************************************************/
  if( sperind5 != -1 )
  {
    printf( " *** ERROR: The following diagnostics were returned by" );
  }

Chapter 12. Sample data and applications supplied with Db2 for z/OS 1129
printf( " stored procedure DSN8ED2.
\n" );
for( j = 0; j < sp_err_buf.sp_err_blen; j++ )
   printf( "%c",sp_err_buf.sp_err_txt[j] );
printf( "\n" );
if( rc < RETSEV )
   rc = RETSEV;
}
} /* end of send_DB2_command_to_sp() */

/***************************************************************************/
/*** Write out the DB2 command that has been built from input records ***/
/***************************************************************************/
echo_DB2_command()                                            /*proc*/
{
   short int  c;                       /* local ptr to command buffer*/
   short int  k,kk,l;                  /* counters and loop control  */

   printf( "\n\n *** Input Statement:\n" );

c = 0;
/***************************************************************************/
/* calculate no. of full print lines the cmd uses and print them */
/***************************************************************************/
   kk = cmdbuf.cmdlen / (OUTLEN - 1);
   for( k=1; k<=kk; k++ )
   {
      printf( "\n" );
      for( l=0; l<(OUTLEN-1); l++ )
      {
         printf( "%c",cmdbuf.cmdtxt[c++] );
      }
      printf( "\n" );
   }

/***************************************************************************/
/* calculate no. of partial print lines the cmd uses; print them */
/***************************************************************************/
   kk = cmdbuf.cmdlen % (OUTLEN - 1);
   if( kk > 0 )
   {
      printf( "\n" );
      for( k=1; k<=kk; k++ )
      {
         printf( "%c",cmdbuf.cmdtxt[c++] );
      }
      printf( "\n" );
   }
} /* end of echo_DB2_command() */

/***************************************************************************/
/*** Output the contents of the result set returned by the stored ***/
/*** procedure. ***/
/***************************************************************************/
output_results_from_sp()                                      /*proc*/
{
   /*******************************************************************/
   /* local initialization */
   /*******************************************************************/
   for(j=0; j<(OUTLEN-1); j++)       /* Blank the input array */
   rs_data[j] = BLANK;               /* Initialize result string */
   rs_sequence = 0;                  /* Initialize data sequence */

   printf( "\n\n *** IFI return area:\n" );

   /*******************************************************************/
   /* associate a locator variable with the result */
   /*******************************************************************/
   EXEC SQL ASSOCIATE LOCATOR        /* Associate the result set */
   (DSN8ED2_rs_loc)  /* locator with a host var. */
   WITH PROCEDURE DSN8.DSN8ED2;      /*                            */
   if (SQLCODE != 0 )                  /* If unsuccessful then       */
   {                                   /* - Say so                   */
      sql_error( "*** Associate result set locator call unsuccessful." );
      /* - Print the sqlcode        */
   }
}
rc = RETSEV;                      /* - Flush remainder of proc */
} /* end of output_results_from_sp() */

/***************************************************************
* allocate the result set cursor
***************************************************************
if( rc < RETSEV )                   /* Or if okay so far then */
{
    EXEC SQL ALLOCATE DSN8ED2_RS_CSR /* - Allocate a cursor to read*/
     CURSOR FOR                      /* - Allocate a cursor to read*/
     RESULT SET :DSN8ED2_rs_loc;     /* the result set locator */
    if (SQLCODE != 0 )              /* - If unsuccessful then */
        {                                 /* - Say so */
            sql_error( "*** Allocate result set cursor call unsuccessful." );
            rc = RETSEV;                    /* - Flush remainder of proc*/
        }                                 /* - Print the sqlcode */
}                                   /* - Flush remainder of proc*/

/***************************************************************
* fetch first row from the result set
***************************************************************
if( rc < RETSEV )                   /* Or if okay so far then */
{
    EXEC SQL FETCH DSN8ED2_RS_CSR     /* - Fetch first row (if any) */
     INTO :rs_sequence, :rs_data;    /* - Get the next one from */
    if (SQLCODE != 0 )                /* - If unsuccessful then */
        {                                 /* - Say so */
            sql_error( "*** Priming fetch of result set cursor unsuccessful." );
            rc = RETSEV;                    /* - Set return code */
        }                                 /* -Flush remainder of proc*/
}                                   /* - Print the sqlcode */

/***************************************************************
* output the contents of the result set
***************************************************************
while(SQLCODE == 0 && rc < RETSEV)  /* Or if okay so far then */
{                                   /* until all lines processed */
    printf( " %s
", rs_data );       /* -- Output current line */
    EXEC SQL FETCH DSN8ED2_RS_CSR     /* -- Get the next one from */
     INTO :rs_sequence, :rs_data;    /* -- Next line */
}                                   /* next line */

/***************************************************************
* check for successful processing of result set
***************************************************************
if (SQLCODE != 100 && rc < RETSEV)  /* If unsuccessful then */
{
    sql_error( "*** Fetch of result set cursor unsuccessful." );
    rc = RETSEV;                    /* - Set return code */
}                                   /* -Print the sqlcode */

/***************************************************************
* SQL error handler
***************************************************************
** SQL error handler
**
***************************************************************
** DSNTIAR message structure
**
#define DATA_DIM 10       /* Number of message lines */
struct error_struct {          /* DSNTIAR message structure */
    short int error_len;
    char     error_text[DATA_DIM][OUTLEN-1];
} error_message = {DATA_DIM * (OUTLEN-1)};

extern short int dsntiar( struct sqlca, struct error_struct *msg,
    int *len );

short int rc;                     /* DSNTIAR Return code */
int j;                            /* Loop control */
static int lrecl = OUTLEN - 1;     /* Width of message lines */
Related reference

“Sample applications in TSO” on page 1027
A set of Db2 sample applications run in the TSO environment.

DSN8ED2

Use the Instrumentation Facility Interface (IFI) to process a Db2 command which has been passed from DSN8ED1, the requester program.

---

1132  Db2 12 for z/OS: Application Programming and SQL Guide
/* Description = DB2 command to be processed by IFI. * Input statements from this parameter will be passed to the text_or_command field of the output_area of the IFI */
* utility for processing.
* Output = Parameters explicitly returned:
Symbolic label/name = ARGV[2] (gets ifca_ret_hex)
- IFI return code, in hex
Symbolic label/name = ARGV[3] (gets ifca_res_hex)
- IFI reason code, in hex
Symbolic label/name = ARGV[4] (gets xs_bytes_hex)
- Excess bytes not returned, in hex
Symbolic label/name = ARGV[5] (gets ermsg_buf)
- Formatted SQL error messages
Symbolic label/name = ARGV[6] (gets indvar)
- DB2 indicator variables
Output = Result set returned:
- Result set cursor name = DSN8ED2_RS_CSR
- Formatted responses from IFI for input command
* Exit normal =
* No errors were found in the passed DB2 command and no errors occurred during processing.
* Normal messages =
* Exit-error =
* Errors were found in the passed DB2 command or occurred during processing.
* Return messages = see under output
* External references =
* Routines/services = none
* Data areas = none
* Control blocks = none
* Pseudocode = DSN8ED2: Main
- get the passed DB2 command.
- calculate the return area size for command requests.
- allocate the requested return area.
- format the output area with the requested command.
- issue the command request to IFI.
- create the temporary table to hold the result set.
- if sql_error if an unexpected SQLCODE is encountered
- extract the responses from the IFI return buffer and insert them to the result set table.
- if sql_error if an unexpected SQLCODE is encountered
- open the cursor to the result set table and exit.
- if sql_error if an unexpected SQLCODE is encountered
End DSN8ED2
- sql_error
- invoke DSNTIAR to format the current SQL code and put the messages to output parameter ARGV[5].
- if DSNTIAR cannot detail the code, put the SQLCODE and the SQLERRM to output parameter ARGV[5].
End sql_error
********************************************************************/
*/

Chapter 12. Sample data and applications supplied with Db2 for z/OS 1133
/* ******************* Program Argument List ***********************/
struct inp {
    short int incmlen; /* Arg1 (in): Command stmt */
    char incmtxt[4096]; /* - Input stmt length */
    /* - Input stmt text */
} inputcmd; /* Pointer to input stmt */
struct inp *inpptr; /* Pointer to input struct */

long int ifca_ret_hex; /* Arg2 (out): IFI return code*/
long int ifca_res_hex; /* Arg3 (out): IFI reason code*/
long int xs_bytes_hex; /* Arg4 (out): # records lost */

char errmsg[DATA_DIM+1][BUFROWLN]; /* Arg5 (out): error messages */
short int locind[5]; /* Arg6 (out): indicator vars */

/* *********************** Working variables ************************/ 
char *parg1[]; /* Pointer to argument 1 */
int *parg2; /* Pointer to argument 2 */
int *parg3; /* Pointer to argument 3 */
int *parg4; /* Pointer to argument 4 */
char *parg5[][BUFROWLN]; /* Pointer to argument 5 */
short int *parg6; /* Pointer to argument 6 */

long int rc, lastrc; /* Return codes */

EXEC SQL BEGIN DECLARE SECTION;
EXEC SQL INCLUDE SQLCA;
EXEC SQL DECLARE DSN8ED2_RS_CSR
CURSOR WITH RETURN WITH HOLD FOR
SELECT RS_SEQUENCE, RS_DATA
FROM DSN8.DSN8ED2_RS_TBL /* <- Created in job DSNTEJ6T */
ORDER BY RS_SEQUENCE;
EXEC SQL END DECLARE SECTION;
EXEC SQL INCLUDE SQLCA;
EXEC SQL INCLUDE SQLCA;

int main( int argc, char *argv[] ) /* Argument count and list */
{
    /* ******************* Constants ******************************/
    char eye[4] /* Const for IFI eye catcher */
        = {'I','F','C','A'};
    char loc[4] /* Const for IFI location */
        = {'L','O','C','2'};
    char function[9]; /* First parm for IFI call */

    short int numfull; /* No. of lines in area passed*/
        /* from IFI that have BUFROWLN*/
    /* or more bytes */
    short int partrow; /* No. of lines in area passed*/
        /* from IFI that have less */
        /* than BUFROWLN bytes */
    short int i, j, k; /* Loop control vars */
    char *curbyte; /* Pointer to current byte in */
        /* return area */
    char len_bin; /* Length of buffer, in binary*/
    char lenbyt1; /* 1st byte of length */
    char lenbyt2; /* 2nd byte of length */
    /* *********************** IFI Argument List ***********************/

    /**************************************************************/
    /*************************** main routine ***************************/
    /**************************************************************/
while( *(curbyte) != NULLCHAR   &&    i < 4096 )
    
i = 0;
    inputcmd.incmlen = 0;

*******************************************************************/

* determine the length of the command text
*******************************************************************/

02890000
02880000
02870000
02860000
02850000
02840000
02830000
02820000
02810000
02800000
02790000
02780000
02770000
02760000
02750000
02740000
02730000
02720000
02710000
02700000
02690000
02680000
02670000
02660000
02650000
02640000
02630000
02620000
02610000
02600000
02590000
02580000
02570000
02560000
02550000
02540000
02530000
02520000
02510000
02500000
02490000
02480000
02470000
02460000
02450000
02440000
02430000
02420000
02410000
02400000
02390000
02380000
02370000
02360000
02350000
02340000
02330000
02320000
02310000
02300000
02290000
02280000
02270000
02260000
02250000
02240000
02230000
02220000
02210000
02200000
02190000
02180000
02170000
02160000
02150000
02140000
02130000
02120000
02110000
02100000
02090000
02080000
02070000
02060000
02050000
02040000
02030000
02020000
02010000
02000000
01990000
01980000
01970000
01960000
01950000
01940000
01930000
01920000
01910000
01900000
01890000
01880000
01870000
01860000
01850000
01840000
01830000
01820000
01810000
01800000
01790000
01780000
01770000
01760000
01750000
01740000
01730000
01720000
01710000
01700000
01690000
01680000
01670000
01660000
01650000
01640000
01630000
01620000
01610000
01600000
01590000
01580000
01570000
01560000
01550000
01540000
01530000
01520000
01510000
01500000
01490000
01480000
01470000
01460000
01450000
01440000
01430000
01420000
01410000
01400000
01390000
01380000
01370000
01360000
01350000
01340000
01330000
01320000
01310000
01300000
01290000
01280000
01270000
01260000
01250000
01240000
01230000
01220000
01210000
01200000
01190000
01180000
01170000
01160000
01150000
01140000
01130000
01120000
01110000
01100000
01090000
01080000
01070000
01060000
01050000
01040000
01030000
01020000
01010000
01000000
00990000
00980000
00970000
00960000
00950000
00940000
00930000
00920000
00910000
00900000
00890000
00880000
00870000
00860000
00850000
00840000
00830000
00820000
00810000
00800000
00790000
00780000
00770000
00760000
00750000
00740000
00730000
00720000
00710000
00700000
00690000
00680000
00670000
00660000
00650000
00640000
00630000
00620000
00610000
00600000
00590000
00580000
00570000
00560000
00550000
00540000
00530000
00520000
00510000
00500000
00490000
00480000
00470000
00460000
00450000
00440000
00430000
00420000
00410000
00400000
00390000
00380000
00370000
00360000
00350000
00340000
00330000
00320000
00310000
00300000
00290000
00280000
00270000
00260000
00250000
00240000
00230000
00220000
00210000
00200000
00190000
00180000
00170000
00160000
00150000
00140000
00130000
00120000
00110000
00100000
00090000
00080000
00070000
00060000
00050000
00040000
00030000
00020000
00010000
00000000
{  
  inputcmd.incmtxt[i] = *curbyte;  
  i++;  
  curbyte++;  
  inputcmd.incmlen++;  
}

/*******************************************************************/  
* initialize the IFI parameters  
*******************************************************************/

strncpy( function,"COMMAND \0",9 ); /* Set constant */ 03000000

pi = malloc( sizeof(ifca) ); /* Point to IFCA structure */ 03010000
pi->lngth = sizeof(ifca); /* Note length of IFCA area */ 03020000
for(i=0; i<4; i++)  
  {  
    pi->eye_catcher[i] = eye[i]; /* Initialize eye catcher */ 03030000
    pi->owner_id[i] = loc[i]; /* DB2 Loc: 1=Local, 2=Remote */ 03040000
  }

pr = malloc( sizeof(return_area) ); /* Point to IFI return area */ 03050000
for(i=0; i<RETURN_LEN; i++)  
  {  
    pr->rtrn_buff[i] = BLANK; /* Clear the return buffer */ 03060000
    pr->lngth = RETURN_LEN; /* Length of return buffer */ 03070000
  }

po = malloc( sizeof(output_area) ); /* Point to IFI command area */ 03080000
po->lngth = inputcmd.incmlen+4; /* Note length of command text*/ 03090000
for(i=0; i<254; i++)  
  {  
    po->text_or_command[i] = inputcmd.incmtxt[i]; /* Copy in command */ 03100000
  }

po->lngth = inputcmd.incmlen+4; /* Note length of command text*/ 03110000
for( i=0; i<254; i++ )              /* Copy in command            */ 03120000
  {  
    po->text_or_command[i] = inputcmd.incmtxt[i]; /* Copy in command */ 03130000
  }

po = malloc( sizeof(output_area) ); /* Point to IFI command area */ 03140000
po->lngth = inputcmd.incmlen+4; /* Note length of command text*/ 03150000
for(i=0; i<254; i++)  
  {  
    po->text_or_command[i] = inputcmd.incmtxt[i]; /* Copy in command */ 03160000
  }

po->lngth = inputcmd.incmlen+4; /* Note length of command text*/ 03170000
for(i=0; i<254; i++)  
  {  
    po->text_or_command[i] = inputcmd.incmtxt[i]; /* Copy in command */ 03180000
  }

po = malloc( sizeof(output_area) ); /* Point to IFI command area */ 03190000
po->lngth = inputcmd.incmlen+4; /* Note length of command text*/ 03200000
for(i=0; i<254; i++)  
  {  
    po->text_or_command[i] = inputcmd.incmtxt[i]; /* Copy in command */ 03210000
  }

dsnwli( function,pi,pr,po ); /* Make the IFI call via the DSNWLI macro */ 03220000

*******************************************************************/  
* copy IFI command status code to output parms */ 03230000
*******************************************************************/

ifca_res_hex = pi->ifcarc2; /* IFI Reason code in binary */ 03240000
ifca_hex = pi->ifcarc1; /* IFI Return code in binary */ 03250000
ifca_res_hex = pi->ifcarc2; /* IFI Reason code in binary */ 03260000
xs_bytes_hex = pi->excess_bytes; /* Bytes that did not fit */ 03270000

*******************************************************************/  
* table for transmission to the caller via a result set */ 03280000
*******************************************************************/  
if( pi->bytes_moved != 0 ) /* If data was returned by IFI*/ 03290000
{  
  *******************************************************************/  
* First, clear any residue from the result set table */ 03300000
*******************************************************************/  
EXEC SQL DELETE 03310000
FROM DSN8.DSN8ED2_RS_TBL; 03320000
if( SQLCODE != 0 ) /* +88 because all rows del'd */ 03330000
  SQLERROR(" *** SQL error when clearing temp table ... "); 03340000
  if( SQLCODE != +88 ) /* +88 because all rows del'd */ 03350000
    sql_error( *** SQL error when clearing temp table ... "); 03360000
  rs_sequence = 0; /* Init result set sequence no*/ 03370000
for( k=0; k<BUFROWLN; k++ ) /* Clear result set data var */ 03380000
  rs_data[k] = BLANK; 03390000

*******************************************************************/  
* the IFI return buffer contains one or more variable length */ 03400000
*******************************************************************/  
* records. Each record consists of a 4-byte length component */ 03410000
* followed by a text component. The length component contains */ 03420000
* the length of the text component plus 4 to account for its */ 03430000
* own length. */ 03440000
*******************************************************************/  
* Extract the length of the 1st record in the buffer and sub-*/ 03450000
* tract 4 bytes to obtain the length of just the text portion. */ 03460000
*******************************************************************/  
len_bin = ( (short int)lenbyt1 ) * 16 + ( (short int)lenbyt2 ); 03470000
len_bin = len_bin - 4; /* Discount size of length fld*/ 03480000

*******************************************************************/  
* For each IFI record returned, create one or more records of */ 03490000
* length BUFROWLN and insert them to the result set table */ 03500000
*******************************************************************/  

1136 Db2 12 for z/OS: Application Programming and SQL Guide
while( ( rc < RETSEV ) && ( pi->bytes_moved - len_bin ) > 2 )
{
    curbyte = curbyte + 4;  /* Update position in buffer */
    if( ((short int)( curbyte - &(pr->rtrn_buff[0]) ) + len_bin - 1) > pi->bytes_moved )
        break;  /* At end of buffer */
    numfull = len_bin / BUFSIZE;  /* No. rows of BUFSIZE bytes */
    partrow = len_bin % BUFSIZE;  /* No. of bytes leftover */
    /***************************************************************
    * Move all complete lines
    ***************************************************************/
    if( numfull > 0 )
        for( j=0; j<numfull; j++ )
        {
            for( i=0; i<BUFSIZE; i++ )  /* Clear result set tbl buffer*/
                rs_data[i] = BLANK;
            for( i=0; i<BUFSIZE; i++ )
            {
                rs_data[i] = *curbyte;  /* Build result set table rec */
                curbyte++;  /* Bump ptr into IFI rtrn buff*/
            }
            rs_sequence++;  /* Bump result set tbl sequ no*/
            EXEC SQL INSERT  /* Insert to the table */
            INTO  DSN8.DSN8ED2_RS_TBL
            VALUES(:rs_sequence,:rs_data );
            if( SQLCODE != 0 )
                sql_error( "*** SQL error when inserting full line ..." );
        }
    /***************************************************************
    * Move leftover bytes to one last result set table record
    ***************************************************************/
    if( rc < RETSEV && partrow > 0 )
        for( i=0; i<BUFSIZE; i++ )  /* Clear result set tbl buffer*/
            rs_data[i] = BLANK;
        for( i=0; i<partrow; i++ )
        {
            rs_data[i] = *curbyte;  /* Build result set table rec */
            curbyte++;  /* Bump ptr into IFI rtrn buff*/
        }
        rs_data[i-1] = BLANK;  /* Discard linefeed char */
        rs_sequence++;  /* Bump result set tbl sequ no*/
        EXEC SQL INSERT  /* Insert to the table */
        INTO  DSN8.DSN8ED2_RS_TBL
        VALUES(:rs_sequence,:rs_data );
        if( SQLCODE != 0 )
            sql_error( "*** SQL error when inserting partial line ..." );
    
    /***************************************************************
    * Advance to next record in the IFI buffer, extract its length,*
    * and subtract 4 bytes to get length of text portion
    ***************************************************************/
    lenbyt1 = *(curbyte);  /* Set 1st byte of length */
    lenbyt2 = *(curbyte+1);  /* Set 2nd byte of length */
    len_bin  = ( (short int)lenbyt1) * 10 + ((short int)lenbyt2 );
    len_bin = len_bin - 4;  /* Discount for length field */
}  /* End of copying IFI return text to result set table */

*****************************************************************
* Open the cursor to the result set table on the way out
*****************************************************************
if( rc < RETSEV )
{
    EXEC SQL OPEN  DSN8.DSN8ED2_RS_CSR;
    if( SQLCODE != 0 )
        sql_error( "*** SQL error when opening result set cursor ..." );
}
}  /* End of if data was returned by IFI */
/* Set output arguments and DB2 locator variables */
parg2 = (int *)argv[2]; /* locate and recast 2nd arg */
locind[1] = 0; /* tell DB2 to transmit it */

/* locate and recast 3rd arg */
locind[2] = 0; /* tell DB2 to transmit it */

/* locate and recast 4th arg */
locind[3] = 0; /* tell DB2 to transmit it */

if( errmsg[0][0] == BLANK ) /* if no error message exists*/
locind[4] = -1; /* tell DB2 not to send one */
else /* otherwise copy it over and */
{
  locind[4] = 0; /* tell DB2 to transmit it */
  curbyte = ((char *)argv[5]); /* set helper pointer */
  for( i=0; i<DATA_DIM+1; i++ ) /* -parse a row, looking for */
  {
    *curbyte = errmsg[i][j++]; /* -copy nonnull bytes */
    curbyte++; /* End of for( i=0; i<DATA_DIM+1; i++ ) */
  }
  *curbyte = NULLCHAR; /* -null-terminate the buffer*/
  locind[4] = 0; /* -tell DB2 to transmit it */
} /* End of if( errmsg[0][0] != BLANK ) */

locind[2] = 0; /* tell DB2 to transmit it */
locind[1] = 0; /* tell DB2 to transmit it */
locind[4] = -1; /* -tell DB2 not to send one */

/* DSINT message structure */
struct error_struct {
  int error_len;
  char error_text[DATA_DIM][BUFROWLN];
} error_message = {DATA_DIM * BUFROWLN};

extern short int dsntiar( struct sqlca *sqlca,
                         struct error_struct *msg,
                         int *len );

char *curbyte; /* Pointer to current byte in */

short int tiar_rc; /* DSINT Return code */
int / * Loop control */
static int lrecl = BUFROWLN; /* Width of message lines */

rc = RETSEV; /* indicate that a fatal eror has occurred */

/* copy locator message to the error message return buffer */
strcpy( errmsg[0],locmsg );

/* format the SQL message and move it to the error message return buffer */
Related reference

“Sample applications in TSO” on page 1027
A set of Db2 sample applications run in the TSO environment.

DSN8EC1

Demonstrates how a Db2 stored procedure can use IMS Open Database Access (ODBA) to connect to IMS DBCTL and access IMS data.
*******
* Attributes = Re-entrant
**
* Entry Point = DSNBEC1
**
* Purpose = See function
**
* Linkage = Standard MVS program invocation
**
* Input = Parameters explicitly passed to this function:
**
*   TDBCTLID ...... PIC X(8)  * 00005700
*   - IMS subsystem id  * 00005800
*   COMMAND ....... PIC X(8)  * 00005900
*   - Action to perform: ADD, UPD, DIS, DEL  * 00006000
*   LAST-NAME .... PIC X(10)  * 00006100
*   FIRST-NAME .... PIC X(10)  * 00006200
*   EXTENSION .... PIC X(10)  * 00006300
*   ZIP-CODE ...... PIC X(7)  * 00006400
*   - Action performed: ADD, UPD, DIS, DEL  * 00006500
* Output = Parameters explicitly passed by this function:
*   COMMAND ....... PIC X(8)  * 00006600
*   - Action performed: ADD, UPD, DIS, DEL  * 00006700
*   LAST-NAME .... PIC X(10)  * 00006800
*   FIRST-NAME .... PIC X(10)  * 00006900
*   EXTENSION .... PIC X(10)  * 00007000
*   ZIP-CODE ...... PIC X(7)  * 00007100
*   AIBREASN ...... PIC S9(9) COMP  * 00007200
*   - Reason code from IMS AIB call  * 00007300
*   AIBRETRN ...... PIC S9(9) COMP  * 00007400
*   - Return code from IMS AIB call  * 00007500
*   ERROR-CALL .... PIC X(4)  * 00007600
*   - DL/I command that failed  * 00007700
* Exit-Normal = Return Code 0 Normal Completion
**
* Exit-Error = Return Code 0 Abnormal Completion
**
* Error Messages = None: Errors are signaled by means of SQLCODEs and DL/I codes returned to the client.
**
* External References =
**
* Routines/Services =
*   AERTDLI - DL/I interface for ODBA  * 00008900
* Data areas = None  * 00009100
* Control Blocks =
*   AIB - DL/I Application Interface Block  * 00009300
* Tables = None  * 00009500
* Change Activity = None  * 00009900
* *Pseudocode*
**
* PROCEDURE A00000-ODBA-SP
* Call B10000-ALLOCATE-AIB to allocate the IMS AIB  * 00010400
* Call B20000-PREPARE-REQUEST to format input from the client  * 00010500
* Call B30000-PROCESS-REQUEST to access data on IMS  * 00010600
* Call C23000-ADD-ENTRY if client passed ADD request  * 00010900
*   Call D31000-INSERT-TO-DB to process IMS ISRT  * 00010900
* Call C22000-UPDATE-ENTRY if client passed UPD request  * 00011000
* Call D31000-GET-HOLD-UNIQUE-FROM-DB for IMS GUH  * 00011100
* Call D32200-REPLACE-IN-DB for IMS REPL  * 00011200
* Call C33000-DELETE-ENTRY if client passed DEL request  * 00011300
* Call D32100-GET-HOLD-UNIQUE-FROM-DB for IMS GUH  * 00011400
* Call D32200-DELETE-FROM-DB for IMS DLET  * 00011500
* Call C21000-DISPLAY-ENTRY if client passed DIS request  * 00011600
* Call D33100-GET-UNIQUE-FROM-DB for IMS GU  * 00011700
* Call B40000-DEALLOCATE-AIB to pend unit of work on IMS  * 00011900
* *
DATA DIVISION.
WORKING-STOREAGE SECTION.

*****************************************************************
** Miscellaneous variables
*****************************************************************
03  FILLER     PIC  X(5).
03  DB2TEMP-IOCMD PIC  X(3).
02  DB2TEMP-COMMAND REDEFINES DB2IW-COMMAND.
02  DB2IW-COMMAND    PIC  X(8).
01  DB2IN-COMMAND.

02  IO-COMMAND   PIC  X(8) VALUE SPACES.
02  IO-FILLER    PIC  X(3) VALUE SPACES.
03  IO-ZIP-CODE    PIC  X(7).
03  IO-EXTENSION   PIC  X(10).
03  IO-FIRST-NAME  PIC  X(10).
03  IO-LAST-NAME   PIC  X(10).
02  IO-DATA REDEFINES IO-BLANK.
02  IO-BLANK  PIC X VALUE ')'.
02  SSA-KEY       PIC X(10).
02  SEG-KEY-NAME  PIC X(11) VALUE '(A1111111 ='.
02  SEGMENT-NAME  PIC X(8)  VALUE 'A1111111'.
02  SSA-KEY-VALUE PIC X(10).
02  FILLER        PIC X VALUE ')'.
02  SSA1          PIC X(9) VALUE 'A1111111'.
02  APSBNAME      PIC X(8) VALUE 'DFSIVP6'.
02  PCDBNAME      PIC X(8) VALUE 'TELEPCB1'.
02  SSA10         PIC X(9) VALUE 'A1111111'.
02  GET-UNIQUE    PIC X(4) VALUE 'GU'.
02  GET-HOLD-UNIQUE PIC X(4) VALUE 'GHU'.
02  GET-NEXT      PIC X(4) VALUE 'GN'.
02  ISRT          PIC X(4) VALUE 'ISRT'.
02  DLET          PIC X(4) VALUE 'DLET'.
02  REPL          PIC X(4) VALUE 'REPL'.
02  APSB          PIC X(4) VALUE 'APSB'.
02  DPSB          PIC X(4) VALUE 'DPSB'.
02  APPERR        PIC X(3) VALUE '264'.
02  INVCMD        PIC X(3) VALUE '440'.
02  NOKEY         PIC X(3) VALUE '218'.

*****************************************************************
** Application Interface Block(AIB) mapping
*****************************************************************
01 AIB.
02 AIBID PIC X(8).
02 AIBLEN PIC 9(9) USAGE BINARY.
02 AIBSFUNC PIC X(8).
02 AIBRSNM1 PIC X(8).
02 AIBRSNM2 PIC X(8).
02 AIBRESV1 PIC X(8).
02 AIBAILEN PIC 9(9) USAGE BINARY.
02 AIBAOUSE PIC 9(9) USAGE BINARY.
02 AIBRESV2 PIC X(12).
02 AIBRETRN PIC 9(9) USAGE BINARY.
02 AIBREASN PIC 9(9) USAGE BINARY.
02 AIBRESV3 PIC X(4).
02 AIBRESA1 USAGE POINTER.
02 AIBRESA2 USAGE POINTER.
02 AIBRESA3 USAGE POINTER.
02 AIBRESV4 PIC X(40).
02 AIBSAVE OCCURS 18 TIMES USAGE POINTER.
02 AIBTKNM OCCURS 6 TIMES USAGE POINTER.
02 AIBTKNC PIC X(16).
02 AIBTKV PIC X(16).
02 AIBTOKA OCCURS 2 TIMES PIC 9(9) USAGE BINARY.

*****************************************************************
** DL/I-related declarations
*****************************************************************

WORKING-STORAGE SECTION.

Chapter 12. Sample data and applications supplied with Db2 for z/OS 1141
**Linkage Section.**

- **Data area for DB2 Stored Procedures input/output**
  - `DB2IO-TDBCTLID` PIC X(8).
  - `DB2IO-COMMAND` PIC X(8).
  - `DB2IO-LAST-NAME` PIC X(10).
  - `DB2IO-FIRST-NAME` PIC X(10).
  - `DB2IO-EXTENSION` PIC X(10).
  - `DB2IO-ZIP-CODE` PIC X(7).
  - `DB2OUT-AIBRETRN` PIC S9(9) COMP.
  - `DB2OUT-AIBREASN` PIC S9(9) COMP.
  - `DC-ERROR-CALL` PIC X(4).

- **Data area for DB2 Stored Procedures output**
  - `DB2OUT-AIBRETRN` PIC S9(9) COMP.
  - `DB2OUT-AIBREASN` PIC S9(9) COMP.
  - `DC-ERROR-CALL` PIC X(4).

**Procedure Division**

- **Main Driver: Process data passed by client and apply the data to the IMS IVP phone book database, DFSIVD1.**

```plaintext
MOVE 'GOOD' TO RUN-STATUS.
PERFORM B10000-ALLOCATE-AIB.
IF OKAY THEN PERFORM B20000-PREPARE-REQUEST.
IF OKAY THEN PERFORM B30000-PROCESS-REQUEST.
IF OKAY THEN PERFORM B40000-DEALLOCATE-AIB.
STOP RUN.
```

- **Initialize and allocate the Application Interface Block**

```plaintext
B10000-ALLOCATE-AIB.
INITIALIZE AIB.
SET AIBRESA1 TO NULLS.
SET AIBRESA2 TO NULLS.
SET AIBRESA3 TO NULLS.
MOVE ZEROES TO AIBRETRN.
MOVE ZEROES TO AIBREASN.
MOVE VAIBID TO AIBID.
```
Chapter 12. Sample data and applications supplied with Db2 for z/OS

MOVE LENGTH OF AIB to AIBLEN.
MOVE SPACES to IOAREA.
MOVE LENGTH OF IOAREA to AIBOALEN.
MOVE SPACES to AIBSFUNC.
MOVE APSBNME to AIBRSN1.
MOVE DB2IO-TDBCTLD to AIBRSN2.

Allocate the PSB for the AIB
CALL 'AERTDLI' USING APSB, AIB.

IF AIBRETRN EQUAL ZEROES THEN
  MOVE 0 TO SET-DATA-FLAG
  MOVE 0 TO TADD-FLAG
ELSE
  MOVE 'BAD' TO RUN-STATUS
  MOVE AIBRETRN TO DB2OUT-AIBRETRN
  MOVE AIBREASN TO DB2OUT-AIBREASN.

Allocate data passed from client for processing by ODBA

* Check the leading space in input command and trim it off
* Check the leading space in input LAST NAME and trim it off
* Check the leading space in input FIRST NAME and trim it off
* Check the leading space in input EXTENSION and trim it off
* Check the leading space in input ZIP CODE and trim it off
MOVE SPACES TO TEMP-TWO.

* Move the data to IO area for IMS
MOVE DB2IO-LAST-NAME TO IO-LAST-NAME.
MOVE DB2IO-COMMAND TO IO-COMMAND.
MOVE DB2IO-COMMAND TO DB2IN-COMMAND.
DISPLAY 'TE>DB2TEMP-IOCMD=' DB2TEMP-IOCMD.

* If no command specified, issue error
IF IO-COMMAND EQUAL SPACES THEN
    MOVE 'BAD' TO RUN-STATUS
    MOVE APPERR TO DB2OUT-AIBRETRN
    MOVE INVCMD TO DB2OUT-AIBREASN

* If no LAST NAME specified, issue error
ELSE IF IO-LAST-NAME EQUAL SPACES THEN
    MOVE 'BAD' TO RUN-STATUS
    MOVE APPERR TO DB2OUT-AIBRETRN
    MOVE NOKEY TO DB2OUT-AIBREASN.

*****************************************************************
* Process the request from the client
*****************************************************************
B30000-PROCESS-REQUEST.

* If command is ADD, insert a new record
IF DB2TEMP-IOCMD EQUAL 'ADD' THEN
    PERFORM C31000-ADD-ENTRY

* If command is TAD, insert a new record and trace with WTO
ELSE IF DB2TEMP-IOCMD EQUAL 'TAD' THEN
    MOVE 1 TO TADD-FLAG
    PERFORM C31000-ADD-ENTRY

* If command is UPD, update existing record for LAST NAME
ELSE IF DB2TEMP-IOCMD EQUAL 'UPD' THEN
    PERFORM C32000-UPDATE-ENTRY

* If command is DEL, delete record for LAST NAME
ELSE IF DB2TEMP-IOCMD EQUAL 'DEL' THEN
    PERFORM C33000-DELETE-ENTRY

* If command is DIS, display record for LAST NAME
ELSE IF DB2TEMP-IOCMD EQUAL 'DIS' THEN
    PERFORM C34000-DISPLAY-ENTRY

* Otherwise, issue error for unexpected command
ELSE
    MOVE 'BAD' TO RUN-STATUS
    MOVE APPERR TO DB2OUT-AIBRETRN
    MOVE NOKEY TO DB2OUT-AIBREASN.

*****************************************************************
* Deallocate the ODBA Application Interface Block
*****************************************************************
B40000-DEALLOCATE-AIB.

* Prep keyword, below, tells IMS to move in-flight transactions
  to in-doubt state, so checkpoint or rollback can be deferred
* until DB2 stored procedure client issues COMMIT or ROLLBACK
* or SFPREP to AIBSFUNC.

MOVE APSBNME to AIBRSNM1.
MOVE SFPREP to AIBSFUNC.

* Deallocaete the PSB for the AIB
CALL 'AERTDLI' USING DPSB, AIB.

*****************************************************************
* Addition request handler
*****************************************************************
C31000-ADD-ENTRY.
MOVE DB2IO-FIRST-NAME TO IO-FIRST-NAME. 00046000
MOVE DB2IO-EXTENSION TO IO-EXTENSION. 00046100
MOVE DB2IO-ZIP-CODE TO IO-ZIP-CODE. 00046200
MOVE IO-COMMAND TO DB2IO-COMMAND. 00046300

00046400
IF DB2IO-FIRST-NAME EQUAL SPACES
  OR DB2IO-EXTENSION EQUAL SPACES
  OR DB2IO-ZIP-CODE EQUAL SPACES THEN
  MOVE 'BAD' TO RUN-STATUS
  MOVE APPERR TO DB2OUT-AIBRETNR
  MOVE INVCMD TO DB2OUT-AIBREAASN
ELSE
  PERFORM D31100-INSERT-TO-DB.
END-IF

*****************************************************************
* Update request handler                                          *
*****************************************************************

C32000-UPDATE-ENTRY.

MOVE 0 TO SET-DATA-FLAG.
MOVE IO-LAST-NAME TO SSA-KEY.
PERFORM D32100-GET-HOLD-UNIQUE-FROM-DB.
IF AIBRETRN = ZEROES THEN
  IF DB2IO-FIRST-NAME NOT = SPACES THEN
    MOVE 1 TO SET-DATA-FLAG
    MOVE DB2IO-FIRST-NAME TO IO-FIRST-NAME
  END-IF
  IF DB2IO-EXTENSION NOT = SPACES THEN
    MOVE 1 TO SET-DATA-FLAG
    MOVE DB2IO-EXTENSION TO IO-EXTENSION
  END-IF
  IF DB2IO-ZIP-CODE NOT = SPACES THEN
    MOVE 1 TO SET-DATA-FLAG
    MOVE DB2IO-ZIP-CODE TO IO-ZIP-CODE
  END-IF
MOVE IO-COMMAND TO DB2IO-COMMAND.
IF NO-SET-DATA THEN
  PERFORM D32200-REPLACE-IN-DB
ELSE
  MOVE 'BAD' TO RUN-STATUS
  MOVE APPERR TO DB2OUT-AIBRETNR
  MOVE INVCMD TO DB2OUT-AIBREAASN.
END-IF

*****************************************************************
* Delete request handler                                          *
*****************************************************************

C33000-DELETE-ENTRY.

MOVE IO-LAST-NAME TO SSA-KEY.
PERFORM D32100-GET-HOLD-UNIQUE-FROM-DB.
IF AIBRETRN = ZEROES THEN
  MOVE IO-COMMAND TO DB2IO-COMMAND
  PERFORM D33200-DELETE-FROM-DB.
ELSE
  MOVE 'BAD' TO RUN-STATUS
  MOVE APPERR TO DB2OUT-AIBRETNR
  MOVE INVCMD TO DB2OUT-AIBREAASN.
END-IF

*****************************************************************
* Display request handler                                         *
*****************************************************************

C34000-DISPLAY-ENTRY.

MOVE IO-LAST-NAME TO SSA-KEY.
DISPLAY 'TE>SSA-KEY=' SSA-KEY.
PERFORM D34100-GET-UNIQUE-FROM-DB.
IF AIBRETRN = ZEROES THEN
  MOVE IO-LAST-NAME TO DB2IO-LAST-NAME
  MOVE IO-FIRST-NAME TO DB2IO-FIRST-NAME
  MOVE IO-EXTENSION TO DB2IO-EXTENSION
  MOVE IO-ZIP-CODE TO DB2IO-ZIP-CODE
  MOVE IO-COMMAND TO DB2IO-COMMAND.
ELSE
  MOVE 'BAD' TO RUN-STATUS
  MOVE APPERR TO DB2OUT-AIBRETNR
  MOVE INVCMD TO DB2OUT-AIBREAASN.
END-IF

*****************************************************************
* Data base segment insert request handler                        *
*****************************************************************

D31100-INSERT-TO-DB.

MOVE DPCBNME to AIBRSNM1.
CALL 'AERTDLI' USING ISRT, AIB, IOAREA, SSA1.
IF AIBRETRN = ZEROES THEN
  MOVE 0 TO TADD-FLAG
  END-IF

Chapter 12. Sample data and applications supplied with Db2 for z/OS 1145
ELSE
  MOVE 'BAD' TO RUN-STATUS
  DISPLAY 'ISRT AIBRETRN=' AIBRETRN
  DISPLAY 'ISRT AIBREASN=' AIBREASN
  DISPLAY 'ISRT AIBRESA1=' AIBRESA1
  DISPLAY 'ISRT AIBRESA2=' AIBRESA2
  DISPLAY 'ISRT AIBRESA3=' AIBRESA3
  MOVE APPERR TO DB2OUT-AIBRETRN
  MOVE INVCMD TO DB2OUT-AIBREASN
  MOVE ISRT TO DC-ERROR-CALL.

*****************************************************************
* Data base segment request handler                                 *
*****************************************************************
D32100-GET-HOLD-UNIQUE-FROM-DB.
  MOVE DPCBNME to AIBRSNM1.
  CALL 'AERTDLI' USING GET-HOLD-UNIQUE, AIB, IOAREA, SSA.
  IF AIBRETRN NOT EQUAL ZEROES THEN
    MOVE 'BAD' TO RUN-STATUS
    MOVE APPERR TO DB2OUT-AIBRETRN
    MOVE INVCMD TO DB2OUT-AIBREASN
    MOVE GET-HOLD-UNIQUE TO DC-ERROR-CALL.

*****************************************************************
* Data base segment replace request handler                        *
*****************************************************************
D32200-REPLACE-IN-DB.
  MOVE DPCBNME to AIBRSNM1.
  CALL 'AERTDLI' USING REPL, AIB, IOAREA.
  IF AIBRETRN NOT EQUAL ZEROES THEN
    MOVE 'BAD' TO RUN-STATUS
    MOVE APPERR TO DB2OUT-AIBRETRN
    MOVE INVCMD TO DB2OUT-AIBREASN
    MOVE REPL TO DC-ERROR-CALL.

*****************************************************************
* Data base segment delete request handler                         *
*****************************************************************
D33200-DELETE-FROM-DB.
  MOVE DPCBNME to AIBRSNM1.
  CALL 'AERTDLI' USING DLET, AIB, IOAREA.
  IF AIBRETRN NOT EQUAL ZEROES THEN
    MOVE 'BAD' TO RUN-STATUS
    MOVE APPERR TO DB2OUT-AIBRETRN
    MOVE INVCMD TO DB2OUT-AIBREASN
    MOVE DLET TO DC-ERROR-CALL.

*****************************************************************
* Data base segment GET-UNIQUE request handler                     *
*****************************************************************
D34100-GET-UNIQUE-FROM-DB.
  MOVE DPCBNME to AIBRSNM1.
  CALL 'AERTDLI' USING GET-UNIQUE, AIB, IOAREA, SSA.
  IF AIBRETRN NOT EQUAL ZEROES THEN
    MOVE 'BAD' TO RUN-STATUS
    DISPLAY 'GU AIBRETRN=' AIBRETRN
    DISPLAY 'GU AIBREASN=' AIBREASN
    DISPLAY 'GU AIBRESA1(ADDR PCB)=' AIBRESA1
    DISPLAY 'GU AIBRESA2=' AIBRESA2
    DISPLAY 'GU AIBRESA3=' AIBRESA3
    MOVE APPERR TO DB2OUT-AIBRETRN
    MOVE INVCMD TO DB2OUT-AIBREASN
    MOVE GET-UNIQUE TO DC-ERROR-CALL.

Related reference
“Sample applications in TSO” on page 1027
A set of Db2 sample applications run in the TSO environment.

**DSN8EC2**

Demonstrates how to CALL the Db2 sample ODBA stored procedure, DSN8.

**Related reference**
“Sample applications in TSO” on page 1027
A set of Db2 sample applications run in the TSO environment.
PROCEDURE A00000-ODBA-SP-CLIENT
  Call A30000-ADD-ENTRY to generate add request
  Call C31000-CALL-ODBA-SP to handle add request
  Call DSNB.ODSBE1 to perform add request
  Call D31100-CHECK-SQLCODE to verify DB2 call
  Call E31110-DETAIL-SQL-ERROR to format err
  Call F31111-PRINT-SQL-ERROR-MSG
  Call D31200-CHECK-AIBCODE to verify IMS state
  Call B40000-COMMIT-WORK to commit DB2 work unit
  Call B50000-DISPLAY-ENTRY to generate display request
  Call C31000-CALL-ODBA-SP to handle display request
  Call DSNB.ODSBE1 to perform display request
  Call D31100-CHECK-SQLCODE to verify DB2 call
  Call E31110-DETAIL-SQL-ERROR to format err
  Call F31111-PRINT-SQL-ERROR-MSG
  Call D31200-CHECK-AIBCODE to verify IMS state

ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
SOURCE-COMPUTER. IBM-370.
OBJECT-COMPUTER. IBM-370.

DATA DIVISION.
WORKING-STORAGE SECTION.

* Fields for receiving
  01 DB2-SERVER-LOCATION-NAME PIC X(16).
  01 IMS-SUBSYSTEM-NAME       PIC X(8).

* Parameter list for invoking sample DB2 stored procedure DSN8EC1
  01 DB2IO-TDBCTLID           PIC X(8).
  01 DB2IO-COMMAND            PIC X(8).
  01 DB2IO-LAST-NAME          PIC X(10).
  01 DB2IO-FIRST-NAME         PIC X(10).
  01 DB2IO-EXTENSION          PIC X(10).
  01 DB2IO-ZIP-CODE           PIC X(7).
  01 DB2OUT-AIBRETRN          PIC S9(9) COMP.
  01 DB2OUT-AIBREASN          PIC S9(9) COMP.
  01 DC-ERROR-CALL            PIC X(4).

* Buffer for receiving SQL error messages
  01 ERROR-MESSAGE.
    02 ERROR-LEN            PIC S9(4) COMP VALUE +960.
    02 ERROR-TEXT           PIC X(120) OCCURS 10 TIMES INDEXED BY ERROR-INDEX.

* Job status indicator
  01 RUN-STATUS               PIC X(4). 88 NOT-OKAY VALUE 'BAD'. 88 OKAY VALUE 'GOOD'.

* Include Cobol standard language global variables
  EXEC SQL INCLUDE SQLCA END-EXEC.

LINKAGE SECTION.

* DSNBEC2 invocation parameter list
  01 PARMS.
    05 PARMS-LEN     PIC 9(4) USAGE BINARY.
    05 PARMS-DATA    PIC X(25).

1148 Db2 12 for z/OS: Application Programming and SQL Guide
PROCEDURE DIVISION
USING PARMS.

*****************************************************************
* Main driver: Use ODBA to add to and display from the IMS IVP DB
*****************************************************************
A00000-ODBA-SP-CLIENT.

DISPLAY '****************************************'
'DS8EC2: Sample Client for IMS/ODBA'
'DB2 stored procedure sample (DSN8.DSN8EC1)'.
DISPLAY '*'.
MOVE 'GOOD' TO RUN-STATUS.

PERFORM B10000-PROCESS-PARMS.

PERFORM B20000-CONNECT-TO-SERVER.

IF OKAY THEN
PERFORM B30000-ADD-ENTRY.
IF OKAY THEN
PERFORM B40000-COMMIT-WORK.
IF OKAY THEN
PERFORM B50000-DISPLAY-ENTRY.

DISPLAY '****************************************'

IF NOT-OKAY THEN
MOVE 8 to RETURN-CODE.
STOP RUN.

B10000-PROCESS-PARMS.

*****************************************************************
* Process DSN8EC2 invocation parameters
*****************************************************************
UNSTRING PARMS-DATA
DELIMITED BY SPACE
INTO DB2-SERVER-LOCATION-NAME
IMS-SUBSYSTEM-NAME.
MOVE IMS-SUBSYSTEM-NAME TO DB2IO-TDBCTLID.

B20000-CONNECT-TO-SERVER.

*****************************************************************
* Connect to the remote server
*****************************************************************
DISPLAY '*    Now connecting to ' DB2-SERVER-LOCATION-NAME.
DISPLAY '*    for access to IMS node ' IMS-SUBSYSTEM-NAME.

EXEC SQL CONNECT TO :DB2-SERVER-LOCATION-NAME END-EXEC.

IF SQLCODE IS NOT EQUAL TO ZERO THEN
PERFORM D31100-CHECK-SQLCODE.

B30000-ADD-ENTRY.

*****************************************************************
* Generate and add an entry to the IMS IVP database DFSIVD1
*****************************************************************
MOVE 'ADD' TO DB2IO-COMMAND.
MOVE 'DOE' TO DB2IO-LAST-NAME.
MOVE 'JOHN' TO DB2IO-FIRST-NAME.
MOVE '9-876-5432' TO DB2IO-EXTENSION.
MOVE 0 TO DB2OUT-AIBRETRN.
MOVE 0 TO DB2OUT-AIBREASN.
MOVE ' ' TO DC-ERROR-CALL.

PERFORM C31000-CALL-ODBA-SP.

IF OKAY THEN
DISPLAY '*' Entry for:
DISPLAY '*' - Last Name .......... ' DB2IO-LAST-NAME
DISPLAY '*' - First Name .......... ' DB2IO-FIRST-NAME

Chapter 12. Sample data and applications supplied with Db2 for z/OS 1149
DISPLAY 'Entry for:'
DISPLAY 'Last Name ........: ' DB2IO-LAST-NAME
DISPLAY 'First Name .......: ' DB2IO-FIRST-NAME
DISPLAY 'Extension Number ...: ' DB2IO-EXTENSION
DISPLAY 'Internal Zip Code ..: ' DB2IO-ZIP-CODE

EXEC SQL CALL DSN8.DSN8EC1 (:DB2IO-TDBCTLID,
:DB2IO-COMMAND,
:DB2IO-LAST-NAME,
:DB2IO-FIRST-NAME,
:DB2IO-EXTENSION,
:DB2IO-ZIP-CODE,
:DB2OUT-AIBRETRN,
:DB2OUT-AIBREASN,
:DC-ERROR-CALL)
END-EXEC.

EXEC SQL COMMIT
END-EXEC.

EXEC SQL CALL DSN8.DSN8EC1 (:DB2IO-TDBCTLID,
:DB2IO-COMMAND,
:DB2IO-LAST-NAME,
:DB2IO-FIRST-NAME,
:DB2IO-EXTENSION,
:DB2IO-ZIP-CODE,

EXEC SQL CALL DSN8.DSN8EC1 (:DB2IO-TDBCTLID,
:DB2IO-COMMAND,
:DB2IO-LAST-NAME,
:DB2IO-FIRST-NAME,
:DB2IO-EXTENSION,
:DB2IO-ZIP-CODE,

EXEC SQL CALL DSN8.DSN8EC1 (:DB2IO-TDBCTLID,
:DB2IO-COMMAND,
:DB2IO-LAST-NAME,
:DB2IO-FIRST-NAME,
:DB2IO-EXTENSION,
:DB2IO-ZIP-CODE,

EXEC SQL CALL DSN8.DSN8EC1 (:DB2IO-TDBCTLID,
:DB2IO-COMMAND,
:DB2IO-LAST-NAME,
:DB2IO-FIRST-NAME,
:DB2IO-EXTENSION,
:DB2IO-ZIP-CODE,

EXEC SQL CALL DSN8.DSN8EC1 (:DB2IO-TDBCTLID,
:DB2IO-COMMAND,
:DB2IO-LAST-NAME,
:DB2IO-FIRST-NAME,
:DB2IO-EXTENSION,
:DB2IO-ZIP-CODE,

EXEC SQL CALL DSN8.DSN8EC1 (:DB2IO-TDBCTLID,
:DB2IO-COMMAND,
:DB2IO-LAST-NAME,
:DB2IO-FIRST-NAME,
:DB2IO-EXTENSION,
:DB2IO-ZIP-CODE,

EXEC SQL CALL DSN8.DSN8EC1 (:DB2IO-TDBCTLID,
:DB2IO-COMMAND,
:DB2IO-LAST-NAME,
:DB2IO-FIRST-NAME,
:DB2IO-EXTENSION,
:DB2IO-ZIP-CODE,

EXEC SQL CALL DSN8.DSN8EC1 (:DB2IO-TDBCTLID,
:DB2IO-COMMAND,
:DB2IO-LAST-NAME,
:DB2IO-FIRST-NAME,
:DB2IO-EXTENSION,
:DB2IO-ZIP-CODE,

EXEC SQL CALL DSN8.DSN8EC1 (:DB2IO-TDBCTLID,
:DB2IO-COMMAND,
:DB2IO-LAST-NAME,
:DB2IO-FIRST-NAME,
:DB2IO-EXTENSION,
:DB2IO-ZIP-CODE,
FROM 1 BY 1 UNTIL ERROR-INDEX GREATER THAN 10.

* **MESSAGE FORMAT**
* **ROUTE ERROR**
* **PRINT ERROR MESSAG**

F31111-PRINT-SQL-ERROR-MSG.

****************************************************************
** Print message text
****************************************************************

DISPLAY ERROR-TEXT (ERROR-INDEX).

D31200-CHECK-AIBCODE.

****************************************************************
** Verify that the IMS operation via ODBA succeeded
****************************************************************

IF DB2OUT-AIBRETRN NOT = 0 OR DB2OUT-AIBREASN NOT = 0 THEN

MOVE 'BAD' TO RUN-STATUS

DISPLAY '* Unexpected return code from ODBA:'
DISPLAY '* Command ..............' DB2IO-COMMAND
DISPLAY '* AIB return code ......' DB2OUT-AIBRETRN
DISPLAY '* AIB reason code ......' DB2OUT-AIBREASN
DISPLAY '* DC error call ........' DC-ERROR-CALL
DISPLAY '*'.

Related reference
“Sample applications in TSO” on page 1027
A set of Db2 sample applications run in the TSO environment.

**DSN8ES1**

Accepts a department number from the caller and returns parameters containing the total earnings (salaries and bonuses) for employees in that department, as well as the number of employees who got a bonus.

-- DSN8ES1: SOURCE MODULE FOR THE SAMPLE SQL PROCEDURE
-- LICENSED MATERIALS - PROPERTY OF IBM
-- (C) COPYRIGHT 2000, 2006 IBM CORP. ALL RIGHTS RESERVED.
-- STATUS = VERSION 9
-- Function: Accepts a department number from the caller and returns
-- parameters containing the total earnings (salaries and
-- bonuses) for employees in that department, as well as the
-- number of employees who got a bonus.
-- In addition, DSN8ES1 generates a result set that contains
-- the serial no, first and last name, salary, and bonus for
-- each employee in the department who got a bonus. The
-- result set also contains a sequence number so that it can
-- be read in the order it was generated.
-- Notes:
-- Dependencies:
-- Requires DB2 precompiler support for SQL procedures (DSNHPSM)
-- Requires a global temporary table (created in sample job
-- DSNTEJ63) for returning the result.
-- Restrictions:
-- Module Type: SQL Procedure
-- Processor: DB2 for OS/390 precompiler and IBM C/C++ for OS/390
-- or a subsequent release
-- Attributes: Re-entrant and re-usable
-- Entry Point: DSN8ES1
-- Purpose: See Function, above
-- Parameters:
-- Input: DEPTNO CHAR(3)
-- Output: DEPTSAL DECIMAL(15,2)
-- BONUSCNT INTEGER
--
-- Normal Exit: 00410000
-- Error Exit: 00420000
-- 00430000
-- 00440000
-- External References:
-- 00450000
-- - EMP : DB2 Sample Employee Table 00460000
-- - DSN8.DSNBES1_RS_TBL: Global Temporary Table for result set 00470000
-- 00480000
-- Pseudocode:
-- 00490000
-- - Clear any residual from result set table 00500000
-- - Open cursor on EMP table for employees in department DEPTNO 00510000
-- - While more rows:
-- 00520000
-- 00530000
-- 00540000
-- 00550000
-- 00560000
-- 00570000
-- 00580000
-- 00590000
-- 00600000
-- 00610000
-- 00620000
-- CREATE PROCEDURE DSN8.DSNBES1 00630000
-- ( IN DEPTNO           CHAR(3), 00640000
-- OUT DEPTSAL          DECIMAL(15,2), 00650000
-- OUT BONUSCNT         INT ) 00660000
-- 00670000
-- 00680000
-- 00690000
-- 00700000
-- 00710000
-- 00720000
-- 00730000
-- 00740000
-- 00750000
-- 00760000
-- 00770000
-- 00780000
-- 00790000
-- 00800000
-- 00810000
-- 00820000
-- 00830000
-- 00840000
-- 00850000
-- 00860000
-- 00870000
-- 00880000
-- 00890000
-- 00900000
-- 00910000
-- 00920000
-- 00930000
-- 00940000
-- 00950000
-- 00960000
-- 00970000
-- 00980000
-- 00990000
-- 01000000
-- 01010000
-- 01020000
-- 01030000
-- 01040000
-- 01050000
-- 01060000
-- 01070000
-- 01080000
-- 01090000
-- 01100000
-- 01110000
-- 01120000
-- 01130000
-- 01140000
-- 01150000
-- 01160000
-- 01170000
-- 01180000
-- 01190000
-- 01200000
-- 01210000
-- 01220000
-- 01230000
-- 01240000
-- 01250000
-- 01260000
-- 01270000
-- 01280000
-- 01290000
-- 01300000
-- CREATE PROCEDURE DSN8.DSNBES1 ( IN DEPTNO CHAR(3),
-- OUT DEPTSAL DECIMAL(15,2),
-- OUT BONUSCNT INT )
-- PARAMETER CCSID EBCDIC
-- FENCED
-- RESULT SET 1
-- LANGUAGE SQL
-- NOT DETERMINISTIC
-- MODIFIES SQL DATA
-- COLLID DSNBES!!
-- WLM ENVIRONMENT WLMENV
-- ASUTIME NO LIMIT
-- COMMIT ON RETURN NO
-- P1: BEGIN NOT ATOMIC
-- DECLARE EMPLOYEE_NUMBER CHAR(6) CCSID EBCDIC;
-- DECLARE EMPLOYEE_FIRSTNME CHAR(12) CCSID EBCDIC;
-- DECLARE EMPLOYEE_LASTNAME CHAR(15) CCSID EBCDIC;
-- DECLARE EMPLOYEE_SALARY DECIMAL(9,2) DEFAULT 0;
-- DECLARE EMPLOYEE_BONUS DECIMAL(9,2) DEFAULT 0;
-- DECLARE TOTAL_SALARY DECIMAL(15,2) DEFAULT 0;
-- DECLARE BONUS_COUNTER INT DEFAULT 0;
-- DECLARE END_TABLE INT DEFAULT 0;
-- -- Cursor for result set of employees who got a bonus
-- DECLARE DSNBES1_RS_CSR CURSOR WITH RETURN WITH HOLD FOR
-- SELECT RS_SEQUENCE, RS_EMPNO, RS_FIRSTNME, RS_LASTNAME, RS_SALARY, RS_BONUS
-- FROM DSN8.DSNBES1_RS_TBL ORDER BY RS_SEQUENCE;
-- -- Cursor to fetch department employees
-- DECLARE C1 CURSOR FOR
-- SELECT EMPNO, FIRSTNME, LASTNAME, SALARY, BONUS
-- FROM EMP WHERE WORKDEPT = DEPTNO;
-- DECLARE CONTINUE HANDLER FOR NOT FOUND
-- SET END_TABLE = 1;
-- DECLARE EXIT HANDLER FOR SQLEXCEPTION
-- SET DEPTSAL = NULL;
-- -- Clean residual from the result set table
-- DELETE FROM DSN8.DSNBES1_RS_TBL;
-- OPEN C1;
-- FETCH C1
-- INTO EMPLOYEE_NUMBER,
-- EMPLOYEE_FIRSTNME,
-- Process each employee in the department
WHILE END_TABLE = 0 DO
  -- Update department total salary
  SET TOTAL_SALARY = TOTAL_SALARY + EMPLOYEE_SALARY + EMPLOYEE_BONUS;
  -- If the current employee received a bonus
  IF EMPLOYEE_BONUS > 0.00 THEN
    -- Update department bonus count
    SET BONUS_COUNTER = BONUS_COUNTER + 1;
  -- Add the employee's data to the result set
  INSERT INTO DSN8.DSN8ES1_RS_TBL (RS_SEQUENCE, RS_EMPNO, RS_FIRSTNME, RS_LASTNAME, RS_SALARY, RS_BONUS ) VALUES( P1.BONUS_COUNTER, P1.EMPLOYEE_NUMBER, P1.EMPLOYEE_FIRSTNME, P1.EMPLOYEE_LASTNAME, P1.EMPLOYEE_SALARY, P1.EMPLOYEE_BONUS );
END IF;
FETCH C1 INTO EMPLOYEE_NUMBER, EMPLOYEE_FIRSTNME, EMPLOYEE_LASTNAME, EMPLOYEE_SALARY, EMPLOYEE_BONUS;
END WHILE;
CLOSE C1;
-- Set return parameters
SET DEPTSAL = TOTAL_SALARY;
SET BONUSCNT = BONUS_COUNTER;
-- Open the cursor to the result set
OPEN DSN8ES1_RS_CSR;
END P1

Related reference
“Sample applications in TSO” on page 1027
A set of Db2 sample applications run in the TSO environment.

DSN8ED3
Demonstrates how to call the sample PSM stored procedure DSN8ES1 using static SQL.

/*****************************************************************************/
* Module name = DSN8ED3 (DB2 sample program)            * 00010000
* DESCRIPTIVE NAME = Client for sample PSM Stored Procedure DSN8ES1  * 00020000
* LICENSED MATERIALS - PROPERTY OF IBM  * 00030000
* 5675-DB2  * 00040000
* (C) COPYRIGHT 2000 IBM CORP.  ALL RIGHTS RESERVED.  * 00050000
* STATUS = VERSION 7  * 00060000
* Function: Demonstrates how to call the sample PSM stored procedure DSN8ES1 using static SQL.  * 00070000
* Notes:  * 00080000
* Dependencies: Requires IBM C/C++ for OS/390 V1R3 or higher  * 00090000
* Restrictions:  * 00100000
* Module type: C program  * 00110000
Pseudocode:

DSN8ED3:

- Verify that number of input parameters passed is either:
  - 2 (program name and department number) or
  - 3 (program name, department number, and (remote) server name)
- Other: issue diagnostic message and end with code 0008
- Connect to server location, if one was passed in.
- Call sample stored procedure DSN8ES1, passing the department number as the argument of the first (input) parameter.
- if unsuccessful, call sql_error to issue a diagnostic message, then end with code 0008.
- Report the following parameters, passed back from DSN8ES1:
  - Total of salary and bonuses for department members
  - Number of employees in the department who received a bonus
  - If a result set was returned, call processResultSet to handle it
End DSN8ED3

processResultSet:

- Associate a locator with the result set passed from DSN8ES1, which contains the serial number, first and last name, salary, and bonus for each department member who got a bonus.
- if unsuccessful, call sql_error to issue a diagnostic message, then end with code 0008.
- Allocate DSN8ES1_RS_CSR as a cursor for the locator
- if unsuccessful, call sql_error to issue a diagnostic message, then end with code 0008.
- Do while not end of cursor
  - Read the cursor
    - If successful, print the row as a report line item
      - else if not end of cursor, call sql_error to issue a diagnostic message, then end with code 0008.
    - Close the cursor
    - if unsuccessful, call sql_error to issue a diagnostic message, then end with code 0008.
End processResultSet

sql_error:
- call DSNTIAR to format the unexpected SQLCODE.
End sql_error

*******************************************************************************/
```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <decimal.h>

#define NULLCHAR '\0' /* Null character */
#define OUTLEN 80      /* Length of output line */
#define DATA_DIM 10      /* Number of message lines */
#define NOT_OK 0      /* Run status indicator: Error*/
#define OK 1      /* Run status indicator: Good */

EXEC SQL INCLUDE SQLCA;

EXEC SQL BEGIN DECLARE SECTION;
char hvDeptNo[4];          /* ID of department to query */
short int niDeptNo = 0;  /* Indic var for dept number */
char hvServerName[17];     /* Location name of server */
decimal(15,2) hvDeptEarnings = 0;  /* Total dept salaries & bonus*/
short int niDeptEarnings = 0;  /* Indic var for dept salary */
long int hvDeptBonusCount= 0;  /* Total no. of bonuses in dpt*/
short int niDeptBonusCount= 0;  /* Indic var for dpt bonus cnt*/
long int hvSequence;           /* Result set row sequence no.*/
char hvEmpno[7];           /* Employee number */
char hvFirstName[13];      /* Employee first name */
char hvLastName[16];       /* Employee last name */
decimal(9,2) hvSalary = 0;  /* Employee salary */
decimal(9,2) hvBonus = 0;  /* Employee bonus */
EXEC SQL END DECLARE SECTION;

EXEC SQL BEGIN DECLARE SECTION;
static volatile SQL TYPE IS RESULT_SET_LOCATOR *DSN8ES1_rs_loc;
EXEC SQL END DECLARE SECTION;

struct error_struct /* DSNTIAR message structure */{ short int error_len; char error_text[DATA_DIM][OUTLEN]; }
error_message = {{DATA_DIM * (OUTLEN)};
#pragma linkage( dsntiar, OS )
extern short int dsntiar( struct sqlca *sqlca,
struct error_struct *msg,
int *len );

EXEC SQL INCLUDE SQLCA;
EXEC SQL BEGIN DECLARE SECTION;
EXEC SQL END DECLARE DECLARE SECTION;
EXEC SQL BEGIN DECLARE SECTION;
EXEC SQL END DECLARE DECLARE SECTION;
EXEC SQL BEGIN DECLARE SECTION;
EXEC SQL END DECLARE DECLARE SECTION;
EXEC SQL BEGIN DECLARE SECTION;
EXEC SQL END DECLARE DECLARE SECTION;
EXEC SQL BEGIN DECLARE SECTION;
EXEC SQL END DECLARE DECLARE SECTION;
EXEC SQL BEGIN DECLARE SECTION;
EXEC SQL END DECLARE DECLARE SECTION;
EXEC SQL BEGIN DECLARE SECTION;
EXEC SQL END DECLARE DECLARE SECTION;
EXEC SQL BEGIN DECLARE SECTION;
EXEC SQL END DECLARE DECLARE SECTION;
EXEC SQL BEGIN DECLARE SECTION;
EXEC SQL END DECLARE DECLARE SECTION;
EXEC SQL BEGIN DECLARE SECTION;
EXEC SQL END DECLARE DECLARE SECTION;
EXEC SQL BEGIN DECLARE SECTION;
EXEC SQL END DECLARE DECLARE SECTION;
EXEC SQL BEGIN DECLARE SECTION;
EXEC SQL END DECLARE DECLARE SECTION;
EXEC SQL BEGIN DECLARE SECTION;
EXEC SQL END DECLARE DECLARE SECTION;
EXEC SQL BEGIN DECLARE SECTION;
EXEC SQL END DECLARE DECLARE SECTION;
EXEC SQL BEGIN DECLARE SECTION;
EXEC SQL END DECLARE DECLARE SECTION;
EXEC SQL BEGIN DECLARE SECTION;
EXEC SQL END DECLARE DECLARE SECTION;
EXEC SQL BEGIN DECLARE SECTION;
EXEC SQL END DECLARE DECLARE SECTION;
EXEC SQL BEGIN DECLARE SECTION;
EXEC SQL END DECLARE DECLARE SECTION;
EXEC SQL BEGIN DECLARE SECTION;
EXEC SQL END DECLARE DECLARE SECTION;
EXEC SQL BEGIN DECLARE SECTION;
EXEC SQL END DECLARE DECLARE SECTION;
EXEC SQL BEGIN DECLARE SECTION;
EXEC SQL END DECLARE DECLARE SECTION;
EXEC SQL BEGIN DECLARE SECTION;
EXEC SQL END DECLARE DECLARE SECTION;
EXEC SQL BEGIN DECLARE SECTION;
EXEC SQL END DECLARE DECLARE SECTION;
EXEC SQL BEGIN DECLARE SECTION;
EXEC SQL END DECLARE DECLARE SECTION;
EXEC SQL BEGIN DECLARE SECTION;
EXEC SQL END DECLARE DECLARE SECTION;
EXEC SQL BEGIN DECLARE SECTION;
EXEC SQL END DECLARE DECLARE SECTION;
EXEC SQL BEGIN DECLARE SECTION;
EXEC SQL END DECLARE DECLARE SECTION;
EXEC SQL BEGIN DECLARE SECTION;
EXEC SQL END DECLARE DECLARE SECTION;
EXEC SQL BEGIN DECLARE SECTION;
EXEC SQL END DECLARE DECLARE SECTION;
EXEC SQL BEGIN DECLARE SECTION;
EXEC SQL END DECLARE DECLARE SECTION;
EXEC SQL BEGIN DECLARE SECTION;
EXEC SQL END DECLARE DECLARE SECTION;
EXEC SQL BEGIN DECLARE SECTION;
EXEC SQL END DECLARE DECLARE SECTION;
EXEC SQL BEGIN DECLARE SECTION;
EXEC SQL END DECLARE DECLARE SECTION;
EXEC SQL BEGIN DECLARE SECTION;
EXEC SQL END DECLARE DECLARE SECTION;
EXEC SQL BEGIN DECLARE SECTION;
EXEC SQL END DECLARE DECLARE SECTION;
EXEC SQL BEGIN DECLARE SECTION;
EXEC SQL END DECLARE DECLARE SECTION;
EXEC SQL BEGIN DECLARE SECTION;
EXEC SQL END DECLARE DECLARE SECTION;
EXEC SQL BEGIN DECLARE SECTION;
EXEC SQL END DECLARE DECLARE SECTION;
EXEC SQL BEGIN DECLARE SECTION;
EXEC SQL END DECLARE DECLARE SECTION;
EXEC SQL BEGIN DECLARE SECTION;
EXEC SQL END DECLARE DECLARE SECTION;
EXEC SQL BEGIN DECLARE SECTION;
EXEC SQL END DECLARE DECLARE SECTION;
EXEC SQL BEGIN DECLARE SECTION;
EXEC SQL END DECLARE DECLARE SECTION;
EXEC SQL BEGIN DECLARE SECTION;
EXEC SQL END DECLARE DECLARE SECTION;
EXEC SQL BEGIN DECLARE SECTION;
EXEC SQL END DECLARE DECLARE SECTION;
EXEC SQL BEGIN DECLARE SECTION;
EXEC SQL END DECLARE DECLARE SECTION;
EXEC SQL BEGIN DECLARE SECTION;
EXEC SQL END DECLARE DECLARE SECTION;
EXEC SQL BEGIN DECLARE SECTION;
EXEC SQL END DECLARE DECLARE SECTION;
EXEC SQL BEGIN DECLARE SECTION;
EXEC SQL END DECLARE DECLARE SECTION;
EXEC SQL BEGIN DECLARE SECTION;
EXEC SQL END DECLARE DECLARE SECTION;
EXEC SQL BEGIN DECLARE SECTION;
EXEC SQL END DECLARE DECLARE SECTION;
EXEC SQL BEGIN DECLARE SECTION;
EXEC SQL END DECLARE DECLARE SECTION;
EXEC SQL BEGIN DECLARE SECTION;
EXEC SQL END DECLARE DECLARE SECTION;
EXEC SQL BEGIN DECLARE SECTION;
EXEC SQL END DECLARE DECLARE SECTION;
EXEC SQL BEGIN DECLARE SECTION;
EXEC SQL END DECLARE DECLARE SECTION;
EXEC SQL BEGIN DECLARE SECTION;
EXEC SQL END DECLARE DECLARE SECTION;
EXEC SQL BEGIN DECLARE SECTION;
EXEC SQL END DECLARE DECLARE SECTION;
EXEC SQL BEGIN DECLARE SECTION;
EXEC SQL END DECLARE DECLARE SECTION;
EXEC SQL BEGIN DECLARE SECTION;
EXEC SQL END DECLARE DECLARE SECTION;
EXEC SQL BEGIN DECLARE SECTION;
EXEC SQL END DECLARE DECLARE SECTION;
EXEC SQL BEGIN DECLARE SECTION;
EXEC SQL END DECLARE DECLARE SECTION;
EXEC SQL BEGIN DECLARE SECTION;
EXEC SQL END DECLARE DECLARE SECTION;
EXEC SQL BEGIN DECLARE SECTION;
EXEC SQL END DECLARE DECLARE SECTION;
EXEC SQL BEGIN DECLARE SECTION;
EXEC SQL END DECLARE DECLARE SECTION;
EXEC SQL BEGIN DECLARE SECTION;
EXEC SQL END DECLARE DECLARE SECTION;
EXEC SQL BEGIN DECLARE SECTION;
EXEC SQL END DECLARE DECLARE SECTION;
EXEC SQL BEGIN DECLARE SECTION;
EXEC SQL END DECLARE DECLARE SECTION;
EXEC SQL BEGIN DECLARE SECTION;
EXEC SQL END DECLARE DECLARE SECTION;
EXEC SQL BEGIN DECLARE SECTION;
EXEC SQL END DECLARE DECLARE SECTION;
EXEC SQL BEGIN DECLARE SECTION;
EXEC SQL END DECLARE DECLARE SECTION;
EXEC SQL BEGIN DECLARE SECTION;
EXEC SQL END DECLARE DECLARE SECTION;
EXEC SQL BEGIN DECLARE SECTION;
EXEC SQL END DECLARE DECLARE SECTION;
EXEC SQL BEGIN DECLARE SECTION;
EXEC SQL END DECLARE DECLARE SECTION;
EXEC SQL BEGIN DECLARE SECTION;
EXEC SQL END DECLARE DECLARE SECTION;
EXEC SQL BEGIN DECLARE SECTION;
EXEC SQL END DECLARE DECLARE SECTION;
EXEC SQL BEGIN DECLARE SECTION;
EXEC SQL END DECLARE DECLARE SECTION;
EXEC SQL BEGIN DECLARE SECTION;
EXEC SQL END DECLARE DECLARE SECTION;
EXEC SQL BEGIN DECLARE SECTION;
EXEC SQL END DECLARE DECLARE SECTION;
EXEC SQL BEGIN DECLARE SECTION;
EXEC SQL END DECLARE DECLARE SECTION;
EXEC SQL BEGIN DECLARE SECTION;
EXEC SQL END DECLARE DECLARE SECTION;
EXEC SQL BEGIN DECLARE SECTION;
EXEC SQL END DECLARE DECLARE SECTION;
EXEC SQL BEGIN DECLARE SECTION;
EXEC SQL END DECLARE DECLARE SECTION;
EXEC SQL BEGIN DECLARE SECTION;
EXEC SQL END DECLARE DECLARE SECTION;
EXEC SQL BEGIN DECLARE SECTION;
EXEC SQL END DECLARE DECLARE SECTION;
EXEC SQL BEGIN DECLARE SECTION;
EXEC SQL END DECLARE DECLARE SECTION;
EXEC SQL BEGIN DECLARE SECTION;
EXEC SQL END DECLARE DECLARE SECTION;
EXEC SQL BEGIN DECLARE SECTION;
EXEC SQL END DECLARE DECLARE SECTION;
EXEC SQL BEGIN DECLARE SECTION;
EXEC SQL END DECLARE DECLARE SECTION;
EXEC SQL BEGIN DECLARE SECTION;
EXEC SQL END DECLARE DECLARE SECTION;
EXEC SQL BEGIN DECLARE SECTION;
EXEC SQL END DECLARE DECLARE SECTION;
EXEC SQL BEGIN DECLARE SECTION;
EXEC SQL END DECLARE DECLARE SECTION;
EXEC SQL BEGIN DECLARE SECTION;
EXEC SQL END DECLARE DECLARE SECTION;
EXEC SQL BEGIN DECLARE SECTION;
EXEC SQL END DECLARE DECLARE SECTION;
EXEC SQL BEGIN DECLARE SECTION;
EXEC SQL END DECLARE DECLARE SECTION;
EXEC SQL BEGIN DECLARE SECTION;
EXEC SQL END DECLARE DECLARE SECTION;
EXEC SQL BEGIN DECLARE SECTION;
EXEC SQL END DECLARE DECLARE SECTION;
EXEC SQL BEGIN DECLARE SECTION;
EXEC SQL END DECLARE DECLARE SECTION;
EXEC SQL BEGIN DECLARE SECTION;
EXEC SQL END DECLARE DECLARE SECTION;
EXEC SQL BEGIN DECLARE SECTION;
EXEC SQL END DECLARE DECLARE SECTION;
EXEC SQL BEGIN DECLARE SECTION;
EXEC SQL END DECLARE DECLARE SECTION;
EXEC SQL BEGIN DECLARE SECTION;
EXEC SQL END DECLARE DECLARE SECTION;
EXEC SQL BEGIN DECLARE SECTION;
EXEC SQL END DECLARE DECLARE SECTION;
EXEC SQL BEGIN DECLARE SECTION;
EXEC SQL END DECLARE DECLARE SECTION;
EXEC SQL BEGIN DECLARE SECTION;
EXEC SQL END DECLARE DECLARE SECTION;
EXEC SQL BEGIN DECLARE SECTION;
EXEC SQL END DECLARE DECLARE SECTION;
EXEC SQL BEGIN DECLARE SECTION;
EXEC SQL END DECLARE DECLARE SECTION;
EXEC SQL BEGIN DECLARE SECTION;
EXEC SQL END DECLARE DECLARE SECTION;
EXEC SQL BEGIN DECLARE SECTION;
EXEC SQL END DECLARE DECLARE SECTION;
EXEC SQL BEGIN DECLARE SECTION;
EXEC SQL END DECLARE DECLARE SECTION;
EXEC SQL BEGIN DECLARE SECTION;
EXEC SQL END DECLARE DECLARE SECTION;
EXEC SQL BEGIN DECLARE SECTION;
EXEC SQL END DECLARE DECLARE SECTION;
EXEC SQL BEGIN DECLARE SECTION;
EXEC SQL END DECLARE DECLARE SECTION;
EXEC SQL BEGIN DECLARE SECTION;
EXEC SQL END DECLARE DECLARE SECTION;
EXEC SQL BEGIN DECLARE SECTION;
EXEC SQL END DECLARE DECLARE SECTION;
EXEC SQL BEGIN DECLARE SECTION;
EXEC SQL END DECLARE DECLARE SECTION;
EXEC SQL BEGIN DECLARE SECTION;
EXEC SQL END DECLARE DECLARE SECTION;
EXEC SQL BEGIN DECLARE SECTION;
EXEC SQL END DECLARE DECLARE SECTION;
EXEC SQL BEGIN DECLARE SECTION;
EXEC SQL END DECLARE DECLARE SECTION;
EXEC SQL BEGIN DECLARE SECTION;
EXEC SQL END DECLARE DECLARE SECTION;
EXEC SQL BEGIN DECLARE SECTION;
EXEC SQL END DECLARE DECLARE SECTION;
EXEC SQL BEGIN DECLARE SECTION;
EXEC SQL END DECLARE DECLARE SECTION;
EXEC SQL BEGIN DECLARE SECTION;
EXEC SQL END DECLARE DECLARE SECTION;
EXEC SQL BEGIN DECLARE SECTION;
EXEC SQL END DECLARE DECLARE SECTION;
EXEC SQL BEGIN DECLARE SECTION;
EXEC SQL END DECLARE DECLARE SECTION;
EXEC SQL BEGIN DECLARE SECTION;
EXEC SQL END DECLARE DECLARE SECTION;
EXEC SQL BEGIN DECLARE SECTION;
EXEC SQL END DECLARE DECLARE SECTION;
EXEC SQL BEGIN DECLARE SECTION;
EXEC SQL END DECLARE DECLARE SECTION;
EXEC SQL BEGIN DECLARE SECTION;
EXEC SQL END DECLARE DECLARE SECTION;
EXEC SQL BEGIN DECLARE SECTION;
EXEC SQL END DECLARE DECLARE SECTION;
EXEC SQL BEGIN DECLARE SECTION;
EXEC SQL END DECLARE DECLARE SECTION;
EXEC SQL BEGIN DECLARE SECTION;
EXEC SQL END DECLARE DECLARE SECTION;
EXEC SQL BEGIN DECLARE SECTION;
EXEC SQL END DECLARE DECLARE SECTION;
EXEC SQL BEGIN DECLARE SECTION;
EXEC SQL END DECLARE DECLARE SECTION;
EXEC SQL BEGIN DECLARE SECT
if( argc == 2 )                /* Only dept no. was passed   */
 {          /* Only dept no. was passed */
   strcpy( hvDeptNo, argv[1] );
 }          /* Only dept no. was passed */

else if( argc == 3 )                /* Dept & server name passed */
 {          /* Dept & server name passed */
   strcpy( hvDeptNo, argv[1] );
   strcpy( hvServerName, argv[2] );
   EXEC SQL CONNECT TO :hvServerName;
   if( SQLCODE != 0 )                           /* Dept & server name passed */
     sql_error( " *** Connect to server" );
 }
 else
   {          /* Dept & server name passed */
     printf( "DSN8ED3 failed: Invalid parameter count\n" );
     status = NOT_OK;
   }

if( status == OK )
   printf( "Salary and Bonus Report for Department %s\n", hvDeptNo );

if( status == OK )
 {          /* Dept & server name passed */
   EXEC SQL CALL DSN8.DSN8ES1( :hvDeptNo        :niDeptNo,
                                :hvDeptEarnings  :niDeptEarnings,
                                :hvDeptBonusCount:niDeptBonusCount );
   if( SQLCODE != 0 && SQLCODE != 466 )     /* Dept & server name passed */
     sql_error( " *** Call DSN8ES1" );
   else
     {          /* Dept & server name passed */
       printf( "Total Department Salaries and Bonuses: %D(15,2)\n",  /* Dept & server name passed */
                 hvDeptEarnings );
       printf( "Total Number of Bonuses in Department: %i\n",  /* Dept & server name passed */
                  hvDeptBonusCount );
     }
 }

if( SQLCODE == 0 && status == OK )
 {          /* Dept & server name passed */
   if( hvDeptBonusCount != 0 )
     {          /* Dept & server name passed */
       printf( "*** Error: Result set was expected from DSN8ES1 "  /* Dept & server name passed */
                   "but was not received\n" );
       status = NOT_OK;
     }
 }

if( SQLCODE == 466 && status == OK )
   processResultSet();

if( status != OK )
   completion_code = 8;

return( completion_code );

void processResultSet( void )
 {          /* Dept & server name passed */
   printf( "Bonus Earners are\n" );
   printf( "Serial   First Name     Last Name         "  /* Dept & server name passed */
          "Salary      Bonus\n" );
   printf( "------   ------------   ---------------   "  /* Dept & server name passed */
          "---------   ---------
" );
   EXEC SQL ASSOCIATE LOCATOR( :DSN8ES1_rs_loc )
   WITH PROCEDURE DSN8.DSN8ES1;                           /* Dept & server name passed */
   if( SQLCODE != 0 )                                                   /* Dept & server name passed */
     sql_error( " *** Associate locator DSN8ES1_rs_loc" );
   else
     {          /* Dept & server name passed */
       EXEC SQL ALLOCATE  DSN8ES1_RS_CSR                              /* Dept & server name passed */
                  CURSOR FOR                                                /* Dept & server name passed */
                  RESULT SET :DSN8ES1_rs_loc;                               /* Dept & server name passed */
       if( SQLCODE != 0 )                                                   /* Dept & server name passed */
         sql_error( " *** Allocate cursor for DSN8ES1 result set" );
     }
   if( SQLCODE == 0 && status == OK )                                   /* Dept & server name passed */
   {          /* Dept & server name passed */
     EXEC SQL ALLOCATE  DSN8ES1_RS_CSR                              /* Dept & server name passed */
                 CURSOR FOR                                                /* Dept & server name passed */
                 RESULT SET :DSN8ES1_rs_loc;                               /* Dept & server name passed */
     if( SQLCODE != 0 )                                                   /* Dept & server name passed */
       sql_error( " *** Allocate cursor for DSN8ES1 result set" );
   }

} /* end main */
while( SQLCODE == 0 && status == OK )
{
    EXEC SQL FETCH DSN8ES1_RS_CSR
    INTO :hvSequence,
    :hvEmpno,
    :hvFirstName,
    :hvLastName,
    :hvSalary,
    :hvBonus;
    if( SQLCODE == 0 )
        printf( "%s   %s   %s   %9D(9,2)   %9D(9,2)
”,
        :hvEmpno, :hvFirstName, :hvLastName, :hvSalary, :hvBonus);
    else if( SQLCODE != 100 )
        sql_error( " *** Fetch from DSN8ES1 result set cursor” );
}
} /* end void processResultSet( void ) */

 waste:0268000002690000027000000271000002720000027300000274000002750000027600000277000002780000027900000280000002810000028200000283000002840000028500000286000002870000028800000289000002900000029100000292000002930000029400000295000002960000029700000298000002990000030000003010000030200000303000003040000030500000306000003070000030800000309000003100000031100000312000003130000031400000315000003160000031700000318000003190000032000000321000003220000032300000324000003250000032600000327000003280000032900000330000003310000033200000333000003340000033500000336000003370000033800000339000003400000034100000342000003430000034400000345000003460000034700000348000003490000035000000351000003520000035300000354000003550000035600000357000003580000035900000360000003610000036200000363000003640000036500000366000003670000036800000369000003700000037100000372000003730000037400000375000003760000037700000378000003790000038000000381000003820000038300000384000003850000038600000387000003880000038900000390000003910000039200000393000003940000039500000396000003970000039800000399000004000000

Related reference
“Sample applications in TSO” on page 1027
A set of Db2 sample applications run in the TSO environment.

** DSN8ES2 **
Accepts a bonus base amount (BONUSBAS) to be awarded to employees who are managers.

--- DSN8.DSN8ES2: SOURCE MODULE FOR SAMPLE SQL PROCEDURE
--- LICENSED MATERIALS - PROPERTY OF IBM
--- 5635-DB2
Pseudocode:
- Open a cursor on sample DEPT and EMP tables, that identifies department managers and the number of persons in each department
- For each manager found:
  - Determine the bonus premium according to the number of employees managed: $1000 for more than 10, $500 for 6 to 10, $100 for 1 to 5
  - Update the manager's bonus in the sample EMP table with the sum of the bonus base and the bonus premium
  - Add the manager's bonus to the total bonuses bucket.
- Return total amount of bonuses awarded

CREATE PROCEDURE DSN8.DSN8E2
(IN BONUSBAS DECIMAL(15,2),
OUT BONUSTOT DECIMAL(15,2),
OUT SQLERRCD INTEGER)
PARAMETER CCSID EBCDIC
FENCED
RESULT SETS 0
LANGUAGE SQL
NOT DETERMINISTIC
MODIFIES SQL DATA
COLLID DSN8E2!!
WLM ENVIRONMENT WLMENV
ASUTIME NO LIMIT
COMMIT ON RETURN NO
P1: BEGIN NOT Atomic
DECLARE MANAGER_ID CHAR(6) CCSID EBCDIC;
DECLARE NUM_EMPLOYEES INT DEFAULT 0;
DECLARE BONUSPRM DECIMAL(15,2) DEFAULT 0;
DECLARE BONUSBKT DECIMAL(15,2) DEFAULT 0;
DECLARE END_TABLE INT DEFAULT 0;
DECLARE SQLCODE INT;
-- Cursor gets id and no. of direct reports for each manager.
DECLARE C1 CURSOR FOR
SELECT DEPT.MGRNO,
COUNT(DISTINCT EMP.EMPNO)
FROM DEPT,EMP
WHERE DEPT.DEPTNO = EMP.DEPTNO
  AND EMP.EMPNO = C1.
FOR UPDATE OF BONUS
   RETURN NO
DECLARE CONTINUE HANDLER FOR NOT FOUND
SET END_TABLE = 1;

DECLARE EXIT HANDLER FOR SQLEXCEPTION
SET SQLERRCD = SQLCODE;

SET BONUSTOT = NULL;
SET SQLERRCD = NULL;

OPEN C1;
FETCH C1
INTO MANAGER_ID,
NUM_EMPLOYEES;

WHILE END_TABLE = 0 DO
CASE
WHEN( NUM_EMPLOYEES > 10 ) THEN
SET BONUSPRM = 1000.00;
WHEN( NUM_EMPLOYEES > 5 )  THEN
SET BONUSPRM = 500.00;
WHEN( NUM_EMPLOYEES > 0 )  THEN
SET BONUSPRM = 100.00;
ELSE
SET BONUSPRM = 0.00;
END CASE;

UPDATE EMP
SET BONUS = BONUSBAS + BONUSPRM
WHERE EMPNO = MANAGER_ID;
SET BONUSBKT = BONUSBKT + BONUSBAS + BONUSPRM;

FETCH C1
INTO MANAGER_ID,
NUM_EMPLOYEES;

END WHILE;
CLOSE C1;
SET BONUSTOT = BONUSBKT;
END P1

Related reference
“Sample applications in TSO” on page 1027
A set of Db2 sample applications run in the TSO environment.

DSN8ED6
Demonstrates how to use WLM_REFRESH, the sample stored procedure for refreshing a WLM environment.

Central to Db2 for z/OS is the WLM environment, which provides a framework for managing, controlling, and allocating resources in a large-scale distributed environment. WLM helps ensure that applications receive appropriate resources, and improves resource utilization and system performance. 

The related reference list, "Sample applications in TSO" on page 1027, describes a set of Db2 sample applications run in the TSO environment, illustrating how to use WLM_REFRESH, the sample stored procedure for refreshing a WLM environment.
* Processor: IBM C/C++ for OS/390 V1R3 or higher
* Module size: See linkedit output
* Attributes: Re-entrant and re-usable
* Entry Point: DSN8ED6
* Purpose: See Function
* Linkage: Standard OS/390 linkage
* Parameters: DSN8ED6 uses the C "main" argument convention of
  argv (argument vector) and argc (argument count).
  - ARGV[0] = (input) pointer to a char[9], null-terminated string having the name of
    this program (DSN8ED6)
  - ARGV[1] = (input) pointer to a char[32] null-terminated string having the name of
    the WLM environment to be refreshed.
  - ARGV[2] = (input) pointer to a char[4], null-terminated string having the DB2 sub-
    system id associated with the WLM
    environment to be refreshed
  - ARGV[3] = (input) pointer to a char[8], null-terminated string having the name of a
    secondary authorization id that has
    access to the resource profile <ssid>.
  - WLM_REFRESH.<wlm-environment-name>.
  - in resource class DSNR. WLM_REFRESH
  - requires READ access on that profile
  - in order to fulfill a refresh request.
* Normal Exit: Return Code: 0000
  - Message: none
  - Error Exit: Return Code: 1999
  - Message: Error: Invalid call parameter count
    Specify either 2 or 3 call parameters for DSN8ED6, as follows:
    - ARGV[0] = (input) pointer to a char[9], null-terminated string having the name of
      a WLM environment to be refreshed (1-32 characters)
    - ARGV[1] = (input) pointer to a char[4], null-terminated string having the name of
      a WLM environment (1-4 characters)
    - Set current SQLID to secondary auth id for submitting the refresh request.
    - (Optional. 1-8 characters)
    - Message: <formatted SQL text from DSNTIAR>
    - Message: none
    - External References:
      - Routines/Services: DSNTIAR: DB2 msg text formatter
      - Data areas : None
      - Control blocks : None
    - Pseudocode:
      - DSN8ED6:
        - Verify that number of input parameters passed is either:
        - - 2 (WLM environment name and DB2 ssid); or
        - - 3 (WLM environment name, DB2 ssid, and secondary auth id
        - - Other: issue diagnostic message and end with code 1999
        - - Set current SQLID to secondary auth id, if one was passed in
        - Call sample stored procedure WLM_REFRESH
        - - if unsuccessful, call sql_error to issue a diagnostic mes-
           sage, then end with code 1999.
        - - Report the following parameters, passed back from WLM_REFRESH:
        - - Return code
        - - Return message
        - Set DSN8ED6 return code from WLM_REFRESH return code
        - End DSN8ED6
        - sql_error:
          - call DSNTIAR to format the unexpected SQLCODE.
        - End sql_error
        - Change log:
          - 01/15/04 Pg79759 - Increase authID to 9 bytes
          - 01/04/00
          - 01/05/00
          - 01/06/00
          - 01/07/00
          - 01/08/00
          - 01/09/00
          - 01/10/00
          - 01/11/00
          - 01/12/00
          - 01/13/00
          - 01/14/00
          - 01/15/00
          - 01/16/00
          - 01/17/00
          - 01/18/00
          - 01/19/00
          - 01/20/00
          - 01/21/00
          - 01/22/00
          - 01/23/00
          - 01/24/00
          - 01/25/00
          - 01/26/00
          - 01/27/00
          - 01/28/00
          - 01/29/00
          - 01/30/00
          - 01/31/00
          - 02/01/00
          - 02/02/00
          - 02/03/00
          - 02/04/00
          - 02/05/00
          - 02/06/00
          - 02/07/00
          - 02/08/00
          - 02/09/00
          - 02/10/00
          - 02/11/00
          - 02/12/00
          - 02/13/00
          - 02/14/00
          - 02/15/00
          - 02/16/00
          - 02/17/00
          - 02/18/00
          - 02/19/00
          - 02/20/00
          - 02/21/00
          - 02/22/00
          - 02/23/00
          - 02/24/00
          - 02/25/00
          - 02/26/00
          - 02/27/00
          - 02/28/00
          - 02/29/00
          - 03/01/00
          - 03/02/00
          - 03/03/00
          - 03/04/00
          - 03/05/00
          - 03/06/00
          - 03/07/00
          - 03/08/00
          - 03/09/00
          - 03/10/00
          - 03/11/00
          - 03/12/00
          - 03/13/00
          - 03/14/00
          - 03/15/00
          - 03/16/00
          - 03/17/00
          - 03/18/00
          - 03/19/00
          - 03/20/00
          - 03/21/00
          - 03/22/00
          - 03/23/00
          - 03/24/00
          - 03/25/00
          - 03/26/00
          - 03/27/00
          - 03/28/00
          - 03/29/00
          - 03/30/00
          - 03/31/00
          - 04/01/00
          - 04/02/00
          - 04/03/00
          - 04/04/00
          - 04/05/00
          - 04/06/00
          - 04/07/00
          - 04/08/00
          - 04/09/00
          - 04/10/00
          - 04/11/00
          - 04/12/00
          - 04/13/00
          - 04/14/00
          - 04/15/00
          - 04/16/00
          - 04/17/00
          - 04/18/00
          - 04/19/00
          - 04/20/00
          - 04/21/00
          - 04/22/00
          - 04/23/00
          - 04/24/00
          - 04/25/00
          - 04/26/00
          - 04/27/00
          - 04/28/00
          - 04/29/00
          - 04/30/00
          - 05/01/00
          - 05/02/00
          - 05/03/00
          - 05/04/00
          - 05/05/00
          - 05/06/00
          - 05/07/00
          - 05/08/00
          - 05/09/00
          - 05/10/00
          - 05/11/00
          - 05/12/00
          - 05/13/00
          - 05/14/00
          - 05/15/00
          - 05/16/00
          - 05/17/00
          - 05/18/00
          - 05/19/00
          - 05/20/00
          - 05/21/00
          - 05/22/00
          - 05/23/00
          - 05/24/00
          - 05/25/00
          - 05/26/00
          - 05/27/00
          - 05/28/00
          - 05/29/00
          - 05/30/00
          - 05/31/00
          - 06/01/00
          - 06/02/00
          - 06/03/00
          - 06/04/00
          - 06/05/00
**** Equates ***************]
#define OUTLEN 80 /* Length of output line */
#define DATA_DIM 10 /* Number of message lines */
#define NOT_OK 0 /* Run status indicator: Error */
#define OK 1 /* Run status indicator: Good */

DB2 SQL Communication Area
EXEC SQL INCLUDE SQLCA;
EXEC SQL BEGIN DECLARE SECTION;
char wlmEnvName[33]; /* WLM environment name */
char ssID[5]; /* Subsystem name */
char authID[9]; /* Current authorization id */
char message[123]; /* WLM_REFRESH return message */
EXEC SQL END DECLARE SECTION;

DB2 Message Formatter
struct error_struct /* DSNTIAR message structure */ {
short int error_len;
char error_text[DATA_DIM][OUTLEN];
}
error_message = {DATA_DIM * (OUTLEN)};
#pragma linkage( dsntiar, OS )
extern short int dsntiar( struct sqlca *sqlca,
struct error_struct *msg,
int *len );

DSNB6D Global Variables
short int status = OK; /* DSNB6D run status */
long int completion_code = 0; /* DSNB6D return code */

DSNB6D Function Prototypes
int main( int argc, char *argv[] );
void sql_error( char locmsg[] );

int main( int argc, char *argv[] ){
 printf( "**** DSNB6D: Sample caller of WLM_REFRESH stored procedure (DSNTWR)\n"
); printf( "*
" );
if( argc < 3 || argc > 6 ) {
 printf( "* Error: Invalid call parameter count\n"
); printf( "* Specify either 2 or 3 call parameters "
"for DSNB6D, as follows:\n"
); printf( "* 1. The name of a WLM environment to be "
"refreshed (1-32 characters)\n"
); printf( "* 2. The DB2 subsystem id (1-4 characters)\n"
); printf( "* 3. A secondary authorization id for "
"submitting the refresh request\n"
); printf( "* (Optional. 1-8 characters)\n"
); status = NOT_OK;
}
else if( strlen(argv[1]) < 1 || strlen(argv[1]) > 32 ) {
 printf( "* Error: The WLM environment name must be 1-32 "
"characters in length\n"
); status = NOT_OK;
}
else if( strlen(argv[2]) < 1 || strlen(argv[2]) > 4 ) {
 printf( "* Error: The DB2 subsystem id must be 1-4 "
"characters in length\n"
); status = NOT_OK;
}
else if( argc == 4 && (strlen(argv[3]) < 1 || strlen(argv[3]) > 8) ) {
 printf( "* Error: The secondary authorization id must be 1-8 "
"characters in length\n"
); status = NOT_OK;
}
else...


```c
{ strcpy( wlmEnvName, argv[1] );
  strcpy( ssID, argv[2] );
}

//****************************************************************************
* Change authid if one was passed in
*******************************************************************************/

if( status == OK && argc == 4 ){
  strcpy( authID, argv[3] );
  EXEC SQL SET CURRENT SQLID = :authID;
  if( SQLCODE != 0 )
    sql_error( "Error setting SQLID" );
}

//****************************************************************************
* Call WLM_REFRESH to refresh the specified WLM environment
*******************************************************************************/

if( status == OK ){
  EXEC SQL CALL SYSPROC.WLM_REFRESH( :wlmEnvName,
                                     :ssID,
                                     :message,
                                     :code );
  if( SQLCODE != 0 )
    sql_error( "Error calling SYSPROC.WLM_REFRESH" );
  else
    printf( "* Results: WLM_REFRESH returned code %i 
    and the following message:
    * %s
    
    " , code , message );
}

if( status != OK )
  completion_code = 1999;
else
  completion_code = code;
return( completion_code );

} /* end main */

void sql_error( char locmsg[] ) /* SQL message formatter */
{
  short int   rc; /* DSNTIAR Return code */
  int         j,k; /* Loop control */
  static int  lrecl = OUTLEN; /* Width of message lines */

  /**************************************************************************
  * set status to prevent further processing
  ***************************************************************************/
  status = NOT_OK;

  /**************************************************************************
  * print the locator message
  ***************************************************************************/
  printf( " %.80s
" , locmsg );

  /**************************************************************************
  * format and print the SQL message
  ***************************************************************************/
  rc = dsntiar( &sqlca, &error_message, &lrecl );
  if( rc == 0 )
    for( j=0; j<DATA_DIM; j++ )
    {
      for( k=0; k<OUTLEN; k++ )
        putchar(error_message.error_text[j][k] );
      putchar(\n" );
    }
  else
    {
      printf( " *** ERROR: DSNTIAR could not format the message\n" );
      printf( " *** SQLCODE is %d\n" , SQLCODE );
      printf( " *** SQLERRM is \n" );
      for( j=0; j<sqlca.sqlerrml; j++ )
        printf( "%c", sqlca.sqlerrmc[j] );
      printf( \n" );
    }
} /* end of sql_error */
```
Related reference
“Sample applications in TSO” on page 1027
A set of Db2 sample applications run in the TSO environment.

DSN8ED7
Calls Db2-provided stored procedure ADMIN_INFO_SYSPARM, which returns the current settings of the
Db2 subsystem parameters.

/***************************************************************************/
/* Module name = DSN8ED7 (DB2 sample program)
 /*
 /* DESCRIPTIVE NAME = Caller for SYSPROC.ADMIN_INFO_SYSPARM
 /* (IFCID 106 formatter stored procedure)
 /*
 /* Licensed Materials - Property of IBM
 /* 5635-DB2
 /* (C) COPYRIGHT 1982, 2006 IBM Corp. All Rights Reserved.
 /*
 /* STATUS = Version 11
 /*
 /* Function: Calls DB2-provided stored procedure ADMIN_INFO_SYSPARM,
 /* which returns the current settings of the DB2 subsystem parameters. These settings are then written in
 /* report format to standard output.
 /*
 /* Notes:
 /* Dependencies: Requires IBM C/C++ for z/OS
 /*
 /* Restrictions:
 /* Module type: C program
 /* Processor: IBM C/C++ for z/OS
 /* Module size: See linkedit output
 /* Attributes: Re-entrant and re-usable
 /*
 /* Entry Point: DSN8ED7
 /* Purpose: See Function
 /* Linkage: Standard z/OS linkage
 /*
 /*
 /* Parameters: none
 /*
 /* Normal Exit: Return Code: 0000
 /* - Message: report of DB2 subsystem parameter settings
 /*
 /* Error Exit: Return Code: 0012
 /* - Message: <formatted SQL text from DSNTIAR>
 /*
 /* External References:
 /* - Routines/Services: DSNTIAR: DB2 msg text formatter
 /* - Data areas : None
 /* - Control blocks : None
 /*
 /* Pseudocode:
 /* DSN8ED7:
 /* - Call ADMIN_INFO_SYSPARM
 /* - if unsuccessful, call sql_error to issue a diagnostic mes-
 /*   sage, then end with code 0012.
 /* - Associate a locator variable with the result set
 /* - Allocate the result set cursor
 /* - Fetch first row from the result set
 /* - Print headings
 /* - Output the content of the result set’s current row and fetch
 /*   the next row, until are rows have been fetched
 /* - Check for successful processing of result set
 /* End DSN8ED7
 /*
 /* sql_error:
 /* - call DSNTIAR to format the unexpected SQLCODE.
 /* End sql_error
 /*
 /* Change activity =
 /* 11/07/2012 Convert from SYSPROC.DSNWZP  dn1651_inst1 / dn1651
 /* to SYSPROC.ADMIN_INFO_SYSPARM
 /*
/******************************************************************************/
# Equates

#define NOT_OK 0 /* Run status indicator: Error */
#define OK 1 /* Run status indicator: Good */
#define OUTLEN 80 /* Length of DSNTIAR line */
#define DATA_DIM 10 /* Number of DSNTIAR lines */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

EXEC SQL INCLUDE SQLCA;

EXEC SQL BEGIN DECLARE SECTION;

char hvDB2_MEMBER[9]; /* host var, target DB2 */
short int niDB2_MEMBER = -1; /* indic var for above parm */
long int hvRETURN_CODE = 0; /* host var, return code */
short int niRETURN_CODE = 0; /* indic var for above parm */
char hvMSG[1332]; /* host var, status message */
short int niMSG = 0; /* indic var for above parm */

static volatile SQL TYPE IS RESULT_SET_LOCATOR *DB2_SYSPARM_rs_loc;

char hvROWNUM = 0; /* host var, row number */
char hvMACRO[9]; /* host var, zparm macro name */
char hvPARAMETER[41]; /* host var, zparm name */
short int niINSTALL_PANEL[9]; /* host var, inst panel name */
short int niINSTALL_PANEL = 0; /* indic var for above parm */
short int niINSTALL_FIELD[41]; /* host var, inst field name */
short int niINSTALL_FIELD = 0; /* indic var for above parm */
short int niINSTALL_LOCATION[13]; /* host var, inst field numb */
short int niINSTALL_LOCATION = 0; /* indic var for above parm */
char hvVALUE[2049]; /* host var, zparm setting */
char hvADDITIONAL_INFO[201]; /* host var, zparm setting */
short int niADDITIONAL_INFO = 0; /* indic var for above parm */

EXEC SQL END DECLARE SECTION;

struct error_struct /* DSNTIAR message structure */
{
    short int error_len;
    char error_text[DATA_DIM][OUTLEN];
    error_message = {DATA_DIM * (OUTLEN)};
}

#pragma linkage( dsntiar, OS )
extern short int dsntiar( struct sqlca *sqlca,
                        struct error_struct *msg,
                        int *len );

short int status = OK; /* DSN8ED7 run status */

int main( int argc, char *argv[] );
void sql_error( char locmsg[] ); /* Calls SQL text formatter */

int main( int argc, char *argv[] )
{
    char msgBuf[1400]; /* message buffer */
    /* Call SYSPROC.ADMIN_INFO_SYSPARM */
EXEC SQL
CALL SYSPROC.ADMIN_INFO_SYSPARM
(  :hvDB2_MEMBER  :niDB2_MEMBER
,  :hvRETURN_CODE  :niRETURN_CODE
,  :hvMSG         :niMSG
);

if( SQLCODE != 466 )
{
   sprintf( msgBuf,
            "DSN8ED7: Error calling stored procedure "
            "ADMIN_INFO_SYSPARM: \n"
            "   Return code=%i,\n"
            "   Message=%s"
            ,  hvRETURN_CODE
            ,  hvMSG
            );
sql_error( msgBuf );
}

EXEC SQL ASSOCIATE LOCATOR
( :DB2_SYSPARM_rs_loc)
WITH PROCEDURE SYSPROC.ADMIN_INFO_SYSPARM;

if (SQLCODE != 0 )
{
   sql_error( "*** Associate result set locator "
            "call unsuccessful." );
}

EXEC SQL ALLOCATE DB2_SYSPARM_RS_CSR
CURSOR FOR
RESULT SET :DB2_SYSPARM_rs_loc;

if (SQLCODE != 0 )
{
   sql_error( "*** Allocate result set cursor "
            "call unsuccessful." );
}

EXEC SQL FETCH DB2_SYSPARM_RS_CSR
INTO :hvROWNUM
, :hvMACRO
, :hvPARAMETER
, :hvINSTALL_PANEL  :niINSTALL_PANEL
, :hvINSTALL_FIELD  :niINSTALL_FIELD
, :hvINSTALL_LOCATION  :niINSTALL_LOCATION
, :hvVALUE
, :hvADDITIONAL_INFO
;

if (SQLCODE != 0 )
{
   sql_error( "*** Priming fetch of result "
            "set cursor unsuccessful" );
}

EXEC SQL FETCH DB2_SYSPARM_RS_CSR
INTO :hvROWNUM
, :hvMACRO
, :hvPARAMETER
, :hvINSTALL_PANEL  :niINSTALL_PANEL
, :hvINSTALL_FIELD  :niINSTALL_FIELD
, :hvINSTALL_LOCATION  :niINSTALL_LOCATION
, :hvVALUE
, :hvADDITIONAL_INFO
;

printf( "DSN8ED7: Sample DB2 for z/OS "
        "Configuration Setting Report Generator\n \n" );

printf( "Macro    Parameter                 
        Current                                 
" );
"Description/"
"Install Fld \n "{"
printf( "Name     Name                      
"Setting                                 
"Install Field Name                      
"Panel ID No. \n}"
printf( "-------- ------------------------- 
"--------------------------------------- 
"--------------------------------------- 
"-------- ----
}
*/

/*****************************************************************
* Output the contents of the result set
*************************************************************************/
while( SQLCODE == 0 && status == OK )
{
    if( strcmp( hvMACRO,"DSN6SYSP" ) == 0 
        || strcmp( hvMACRO,"DSN6LOGP" ) == 0 
        || strcmp( hvMACRO,"DSN6ARVP" ) == 0 
        || strcmp( hvMACRO,"DSN6SPRM" ) == 0 
        || strcmp( hvMACRO,"DSN6FAC"  ) == 0 
        || strcmp( hvMACRO,"DSN6GRP"  ) == 0 
        || strcmp( hvMACRO,"DSNHDECP" ) == 0 
    )
        printf( "%-9.8s" 
                "%-26.25s" 
                "%-40.39s" 
                "%-40.39s" 
                "%-9.8s" 
                "%4.4s
", hvMACRO 
                    , hvPARAMETER 
                    , hvVALUE 
                    , hvINSTALL_FIELD 
                    , hvINSTALL_PANEL 
                    , hvINSTALL_LOCATION 
                    , hvADDITIONAL_INFO 
                );

        EXEC SQL FETCH DB2_SYSPARM_RS_CSR
        INTO :hvROWNUM 
            , :hvMACRO 
            , :hvPARAMETER 
            , :hvINSTALL_PANEL 
            , :hvINSTALL_FIELD 
            , :hvINSTALL_LOCATION 
            , :hvVALUE 
            , :hvADDITIONAL_INFO 
    };

    if( status == OK )
        return( 0 );
    else
        return( 12 );
} /* end: main */

void sql_error( char locmsg[] ) /* SQL message formatter */
{ short int rc; /* DSNTIAR Return code */
    int j,k; /* Loop control */
    static int lrecl = OUTLEN; /* Width of message lines */

    /*****************************************************************
    * Set status to prevent further processing
    *******************************************************************/
    status = NOT_OK;

    /*****************************************************************
    * Print the locator message
    *******************************************************************/
    printf( " %s\n", locmsg );

    /*******************************************************************/

Format and print the SQL message

```c
rc = dsntiar( &sqlca, &error_message, &lrecl );
if( rc == 0 )
    for( j=0; j<DATA_DIM; j++ )
        for( k=0; k<OUTLEN; k++ )
            putchar(error_message.error_text[j][k] );
        putchar(\n');
else
    { printf( " *** ERROR: DSNTIAR could not format the message\n" );
      printf( " *** SQLCODE is %d\n",SQLCODE );
      printf( " *** SQLERRM is \n" );
      for( j=0; j<sqlca.sqlerrml; j++ )
          printf( "%c", sqlca.sqlerrmc[j] );
      printf( \n' );
    }

} /* end of sql_error */
```

Related reference
“Sample applications in TSO” on page 1027
A set of Db2 sample applications run in the TSO environment.

**DSN8ED9**

Demonstrates how to use an application program to call DSN8ES3, a sample native SQL procedure.

```c
/***************************************************************************/
* Module name = DSN8ED9 (sample program)                             *
*                                                                    *
* DESCRIPTIVE NAME: Sample client for:                               *
*                   DSN8ES3 (DB2 sample native SQL procedure)        *
*                                                                    *
*     LICENSED MATERIALS - PROPERTY OF IBM                           *
*     5650-DB2                                                       *
*     (C) COPYRIGHT 2006, 2016 IBM CORP.  ALL RIGHTS RESERVED.       *
*                                                                    *
*     STATUS = VERSION 12                                            *
*                                                                    *
* Function: Demonstrates how to use an application program to call   *
*           DSN8ES3, a sample native SQL procedure.  DSN8ED9         *
*           receives the schema and name of a stored procedure       *
*           and passes it to DSN8ES3 to request the CREATE PROCEDURE *
*           statement.                                               *
*                                                                    *
* Notes:                                                             *
*   Dependencies: Requires DSN8.DSN8ES3                              *
*                                                                    *
*   Restrictions:                                                    *
*                                                                    *
* Module type: C program                                             *
*   Processor: DB2 Precompiler                                       *
*              IBM C/C++ for z/OS                                    *
* Module size: See linkedit output                                  *
*   Attributes: Reentrant and reusable                               *
*                                                                    *
* Entry point: DSN8ED9                                               *
*   Purpose: See Function                                            *
*   Linkage: Standard MVS program invocation, three parameters.      *
* Parameters: DSN8ED9 uses the C "main" argument convention of      *
*              argv (argument vector) and argc (argument count).      *
*              - ARGV[0]: (input) pointer to a char[9],              *
*                null-terminated string having the name of           *
*                this program (DSN8ED9)                             *
*              - ARGV[1]: (input) pointer to a char[129],           *
*                null-terminated string having the schema            *
*                of a stored procedure                            *
*              - ARGV[2]: (input) pointer to a char[129],           *
*                null-terminated string having the name of          *
*                a stored procedure                                *
*              - ARGV[3]: (input) pointer to a char[17],            *
*                null-terminated string having the name of          *
*                the server where DSN8ES3 is to be run.             *
*              This is an optional parameter; the local           *
```
server is used if no argument is provided.

Inputs: None

Outputs: Standard output (SYSPRINT)

Normal Exit: Return Code: 0
- Message: CREATE PROCEDURE statement for specified stored procedure

Normal with Warnings Exit: Return Code: 0004
- Message: DSN8ES3 ran successfully but returned no output

Error Exit: Return Code: 0012
- Message: DSN8ES3 has completed with return code <n>
- Message: The length of the argument specified for the <parameter-name> does not fall within the required bounds of <minimum-length> and <maximum-length>
- Message: DSN8ED9 was invoked with <parameter-count> parameters. At least 2 parameters are required
- Message: <formatted SQL text from DSNTIAR>

External References:
- Routines/Services: DSNTIAR: DB2 msg text formatter
- Data areas : None
- Control blocks : None

Pseudocode:
DSN8ED9:
- call getCallParms to receive and validate call parm arguments
- call connectToLocation
- call callDSN8ES3 to invoke the sample native SQL procedure
- call processDSN8ES3resultSet to output results from DSN8ES3
End DSN8ED9

Change activity = 04/22/2015 Storage overlay stops output d176357

---

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <decimal.h>

#define NULLCHAR     '\0'     /* Null character             */
#define RETNRM             0  /* Normal  return code     @04*/
#define RETWRN             4  /* Warning return code        */
#define RETERR             8  /* Error return code          */
#define RETSEV            12  /* Severe error return code   */

enum flag       {No, Yes}; /* Settings for flags         */

EXEC SQL BEGIN DECLARE SECTION;
long int      hvSequence;           /* Result set row sequence no.*/
char          hvLine[80];           /* line                       */
char          hvSpSchema[129];      /* Stored procedure schema    */
short int     niSpSchema      = 0;  /* Indic var for schema       */
char          hvSpName[129];        /* Stored procedure name      */
short int     niSpName        = 0;  /* Indic var for name         */
char          hvLocationName[17];   /* Server location name       */
EXEC SQL END DECLARE SECTION;

EXEC SQL BEGIN DECLARE SECTION;
char          hvLocationName[17];   /* Server location name       */
EXEC SQL END DECLARE SECTION;
static volatile SQL TYPE IS RESULT_SET_LOCATOR *DSN8ES3_rs_loc;
EXEC SQL END DECLARE SECTION;

/********************* DSN8ED9 Global Variables **********************/
unsigned short resultSetReturned = 0; /* DSN8ES3 result set status */
long int rc = 0; /* DSN8ED9 return code */

/********************* DSN8ED9 Function Prototypes **********************/
int main /* DSN8ED9 driver */
( int argc,
  char *argv[])
);

void getCallParms /* Process args to call parms */
( int argc,
  char *argv[])
);

void connectToLocation( void ); /* Connect to DB2 location */
void callDSN8ES3( void ); /* Call DSN8ES3 */

void resultSetReturned = 0; /* DSN8ES3 result set status */
long int rc = 0; /* DSN8ED9 return code */

main
( int argc, char *argv[])
{
  printf( "**** DSN8ED9: Sample client for DB2 PSM "
           "Sample Procedure Sample (DSN8ES3)\n\n" );

  /**************************************************************************/
  * Get input parms, pass them to DSN8ES3, and process the results */
  /**************************************************************************/
  getCallParms(argc,argv);

  /**************************************************************************/
  * Extract the following information from the call parms: */
  /**************************************************************************/
  * (1) The schema of the stored procedure */
  * (2) The name of the stored procedure */
  * (3) Optional: The name of the location where the stored proc */
  *   resides */
  /**************************************************************************/
  getCallParms(argc,argv);

  /**************************************************************************/
  * Connect to location where the stored procedure resides */
  /**************************************************************************/
  if( rc < RETSEV && strlen(hvLocationName) > 0 )
    connectToLocation();

  if( rc < RETSEV )
    callDSN8ES3();

  if( rc < RETSEV && resultSetReturned == Yes )
    processDSN8ES3resultSet();

  return( rc );
}

getCallParms /* Process args to call parms */
( int argc,
  char *argv[])
);
if( argc < 3 || argc > 4 )
    {
        issueInvalidCallParmCountError( argc );
    }
else if( strlen( argv[1] ) < 1 || strlen( argv[1] ) > 130 )
    {
        issueInvalidParmLengthError("Stored procedure schema", 1,130);
    }
    {
        issueInvalidParmLengthError("Stored procedure name", 1,130);
    }
else
    {
        strcpy( hvSpSchema, argv[1] );
        strcpy( hvSpName, argv[2] );
    }
if( argc > 3 )
    {
        issueInvalidParmLengthError("Server Location Name",1,16);
    }
    else
    {
        strcpy( hvLocationName,argv[3] );
    }
    else
    hvLocationName[0] = NULLCHAR;
} /* end of getCallParms */

void connectToLocation( void )        /* Connect to DB2 location    */
/*******************************************************************
* Connects to the DB2 location specified in call parm number 3     *
*******************************************************************/
{ EXEC SQL
  CONNECT TO :hvLocationName;
  if( SQLCODE != 0 )
    {
        issueSqlError( "Connect to location failed" );
    }
} /* end of connectToLocation */

void callDSN8ES3( void )              /* Run sample native SQL proc */
/*******************************************************************
* Calls the DSN8ES3 (sample native SQL procedure)                  *
*******************************************************************/
{ printf( "\n");
  printf( "-> Now requesting CREATE PROCEDURE statement for %s.%s\n",
      hvSpSchema, hvSpName);
  EXEC SQL CALL DSN8.DSN8ES3( :hvSpSchema  :niSpSchema,
                              :hvSpName    :niSpName );
  /***************************************************************
  * Analyze status codes from DSN8ES3                            *
  ***************************************************************
  if( SQLCODE == 466 )
    {
        resultSetReturned = Yes;
    }
  else if( SQLCODE == 0 )
    {
        resultSetReturned = No;
        printf( "\n");
        printf( "-> Call to DSN8ES3 succeeded  ",
                "but returned no result\n" );
    }
  else
    {
        issueSqlError( "Call to DSN8ES3 failed" );
    }
} /* end of callDSN8ES3 */

void processDSN8ES3resultSet( void )   /* Handle DSN8ES3 result set */
/***************************************************************
* Outputs data from the result set returned by DSN8ES3          *
***************************************************************
{ /
  * Associate a locator with the result set from DSN8ES3        *
  */
  associateResultSetLocator();
}


/* Allocate a cursor for the result set */
if( rc < RETSEV )
allocateResultSetCursor();

/* Output data from the result set */
if( rc < RETSEV )
writeDSN8ES3results();

} /* end of processDSN8ES3resultSet */

void associateResultSetLocator(void) /* Associate DSN8ES3 RS locator*/
{
EXEC SQL
ASSOCIATE
LOCATORS( :DSN8ES3_rs_loc )
WITH PROCEDURE DSN8.DSN8ES3;
if( SQLCODE != 0 )
{ issueSqlError( "Associate locator call failed" );
}
} /* end of associateResultSetLocator */

void allocateResultSetCursor( void )  /* Alloc DSN8ES3 RS cursor    */
{
EXEC SQL
ALLOCATE DSN8ES3_RS_CSR
CURSOR FOR RESULT SET :DSN8ES3_rs_loc;
if( SQLCODE != 0 )
{ issueSqlError( "Allocate result set cursor call failed" );
}
} /* end of allocateResultSetCursor */

void writeDSN8ES3results( void )      /* Print DSN8ES3 results      */
{
 fetchFromResultSetCursor();

while( SQLCODE == 0  &&  rc < RETSEV )
{ printf( "%s
",hvLine );
if( rc < RETSEV )
{ fetchFromResultSetCursor();
}
}
} /* end of writeDSN8ES3results */

void fetchFromResultSetCursor( void ) /* Read DSN8ES3 RS cursor     */
{
memset( hvLine,' ',80 );                               /*d176357*/
EXEC SQL
FETCH DSN8ES3_RS_CSR
INTO :hvSequence,
 :hvLine;
if( SQLCODE != 0  &&  SQLCODE != 100  &&  rc < RETSEV )
{ issueSqlError( "*** Fetch from result set cursor failed" );
}
}
void issueInvalidCallParmCountError  /* Handler for parm count err */
( int argc                          /* - in: no. parms received */
);
/*******************************************************************
* Called when this program is invoked with an inappropriate number *
* of call parms.                                                 *
*******************************************************************/
{ printf( "ERROR: DSN8ED9 was invoked with %i parameters\n",--argc );
  printf( " - The first two parms (schema and name "
       "of a stored procedure) are required\n" );
  printf( " - The third parm (location name) "
       "is optional\n" );
  printf( "-----> Processing halted\n" );
  rc = RETSEV;
} /* end of issueInvalidCallParmCountError */

void issueInvalidParmLengthError      /* Handler for parm len error */
( char *parmName,                   /* - in: identify of parm */
  int minLength,                    /* - in: min valid length */
  int maxLength                      /* - in: max valid length */
);
/*******************************************************************
* Called when the length of an argument specified for a DSN8ES3    *
* parameter (parmName) does not fall within the valid bounds for *
* size (minLength and maxLength) for that parameter              *
*******************************************************************/
{ printf( "ERROR: The length of the argument specified for the %s "
       "parameter\n",parmName );
  printf( " does not fall within the required bounds of %i "
       "and %i\n",minLength,maxLength );
  printf( "-----> Processing halted\n" );
  rc = RETSEV;
} /* end of issueInvalidParmLengthError */

#pragma linkage(dsntiar, OS)
void issueSqlError                    /* Handler for SQL error      */
( char *locMsg                      /* - in: Call location */
);
/*******************************************************************
* Called when an unexpected SQLCODE is returned from a DB2 call    *
*******************************************************************/
{ struct error_struct {          /* DSNTIAR message structure */
  short int error_len;
  char error_text[10][80];
} error_message = {10 * 80};
extern short int dsntiar( struct sqlca, *msg, int *len );
short int DSNTIARrc;            /* DSNTIAR Return code */
int j;                            /* Loop control */
static int lrecl = 80;           /* Width of message lines */
/*******************************************************************
* print the locator message                                       *
*******************************************************************/
{ printf( "ERROR: %-80s\n", locMsg );
  printf( "-----> Processing halted\n" );
}
/*******************************************************************
* format and print the SQL message                                *
*******************************************************************/
DSNTIARrc = dsntiar( &sqlca, &error_message, &lrecl );
if( DSNTIARrc == 0 )
  for( j = 0; j <= 10; j++ )
    printf( "%.80s\n", error_message.error_text[j] );
else
  {  
    printf( " *** ERROR: DSNTIAR could not format the message\n" );
    printf( " *** SQLCODE is %d\n",SQLCODE );
    printf( " *** SQLERRM is \n" );
    for( j=0; j<sqlca.sqlerrml; j++ )
      printf( "%c", sqlca.sqlerrmc[j] );
    printf( "\n" );
  }
} /* end of issueSqlError */
Accepts the schema and name of an external stored procedure and returns a result set that contains the
CREATE PROCEDURE statement.

--- DSN8ES3: SOURCE MODULE FOR THE SAMPLE NATIVE SQL PROCEDURE 00010000
--- LICENSED MATERIALS - PROPERTY OF IBM 00020000
--- 5635-DB2 00022000
--- (C) COPYRIGHT 2006 IBM CORP. ALL RIGHTS RESERVED. 00024000
--- STATUS = VERSION 9 00026000
--- Function: Accepts the schema and name of an external stored procedure and returns a result set that contains the
--- CREATE PROCEDURE statement.
--- Notes:
--- Dependencies:
--- - Requires support for native SQL procedures 00028000
--- - Requires a global temporary table (created in sample job DSNTEJ66) for returning the result.
--- Restrictions:
--- Module Type: SQL Procedure 00030000
--- Processor: DB2 for z/OS Version 9 00032000
--- or a subsequent release 00034000
--- Entry Point: DSN8ES3 00036000
--- Parameters:
--- - Input: spSCHEMA VARCHAR(128) 00038000
--- - spNAME VARCHAR(128) 00040000
--- Normal Exit:
--- - Error Exit:
--- External References:
--- - SYSIBM.SYSPRMS : DB2 catalog table for routine parameters 00042000
--- - DSN8.DSN8ES3_RS_TBL: Global Temporary Table for result set 00044000
--- Pseudocode:
--- - Clear any residual from result set table 00046000
--- - Get the stored proc properties from SYSIBM.SYSPRMS 00048000
--- - If not found, return SQLSTATE 38602 and the message: 'Requested object not found' 00050000
--- - If not a stored proc, return SQLSTATE 38603 and the message: 'Object is not a stored procedure' 00052000
--- - If an external stored proc, return SQLSTATE 38604 and the message: 'Object is not an external stored procedure' 00054000
--- - Open a cursor on the SYSPRMS table 00056000
--- - Fetch the first row 00058000
--- - If a row is found, insert the CREATE PROCEDURE clause in the result set 00060000
--- - For each row in the SYSPRMS cursor, build a parameter clause: 00062000
--- - Append the parameter name 00064000
--- - Append the parameter data type 00066000
--- - For string data types, add the CCSID clause 00068000
--- - Insert the entry in the result set table 00070000
--- - Build the remaining clauses and insert each in the result set 00072000

Related reference
“Sample applications in TSO” on page 1027
A set of Db2 sample applications run in the TSO environment.

DSN8ES3
Accepts the schema and name of an external stored procedure and returns a result set that contains the CREATE PROCEDURE statement.
CREATE PROCEDURE DSN8.DSN8ES3
  IN spSCHEMA VARCHAR(128),
  IN spNAME VARCHAR(128)
  ( IN spPARMS ROW VARCHAR(128) )
BEGIN NOT ATOMIC
  COMMIT ON RETURN NO
  ASUTIME NO LIMIT
  MODIFIES SQL DATA
  COMMIT ON RETURN NO
  DECLARE RETURN_POINT CHAR(4) DEFAULT 0;
  DECLARE hvTYPENAME VARCHAR(128) DEFAULT 0;
  DECLARE hvSUBTYPE VARCHAR(15) DEFAULT 0;
  DECLARE hvSCALE SMALLINT DEFAULT 0;
  DECLARE hvROWTYPE VARCHAR(6) DEFAULT 0;
  DECLARE hvPARMNAME VARCHAR(128) DEFAULT 0;
  DECLARE hvORDINAL SMALLINT DEFAULT 0;
  DECLARE hvLENGTH INTEGER DEFAULT 0;
  DECLARE hvENCODING_SCHEME VARCHAR(7) DEFAULT 0;
  DECLARE hvSTAYRESIDENT CHAR(3) DEFAULT 0;
  DECLARE hvSQL_DATA_ACCESS VARCHAR(17) DEFAULT 0;
  DECLARE hvSPECIAL_REGS VARCHAR(25) DEFAULT 0;
  DECLARE hvRUNOPTS VARCHAR(762) DEFAULT 0;
  DECLARE hvROUTINETYPE CHAR(1) DEFAULT 0;
  DECLARE hvRESULT_SETS SMALLINT DEFAULT 0;
  DECLARE hvPROGRAM_TYPE VARCHAR(4) DEFAULT 0;
  DECLARE hvORIGIN CHAR(1) DEFAULT 0;
  DECLARE hvRESULT_SETS SMALLINT DEFAULT 0;
  DECLARE hvRESULT_SETS SMALLINT DEFAULT 0;
  DECLARE hvRESULT_SETS SMALLINT DEFAULT 0;
  DECLARE hvRESULT_SETS SMALLINT DEFAULT 0;
-- Cursor to fetch proc parm properties from SYIBM.SYSPARMS
DECLARE SYSPARMS_CURSOR CURSOR FOR
SELECT PARMNAME,
      CASE ROWTYPE
          WHEN 'P' THEN 'IN'
          WHEN 'O' THEN 'OUT'
          WHEN 'B' THEN 'INOUT'
          END
      ORDINAL,
      TYPENAME,
      LENGTH,
      SCALE,
      CASE SUBTYPE
          WHEN 'B' THEN 'FOR BIT DATA'
          WHEN 'M' THEN 'FOR MIXED DATA'
          WHEN 'S' THEN 'FOR SBCS DATA'
          WHEN ' ' THEN '
          END
      CASE ENCODING_SCHEME
          WHEN 'A' THEN 'ASCII'
          WHEN 'E' THEN 'EBCDIC'
          WHEN 'U' THEN 'UNICODE'
          WHEN ' ' THEN '
          END
      FROM SYIBM.SYSPARMS
      WHERE SCHEMA = spSCHEMA
      AND SPECIFICNAME = spNAME
      AND ORDINAL <> 0
      ORDER BY ORDINAL
      FOR FETCH ONLY;
      END

DECLARE CONTINUE HANDLER FOR NOT FOUND
SET END_TABLE = 1;
DECLARE EXIT HANDLER FOR SQLEXCEPTION
SET END_TABLE = 1;

DECLARE CONTINUE HANDLER FOR NOT FOUND
DECLARE EXIT HANDLER FOR SQLEXCEPTION
END
CASE EXTERNAL_SECURITY
  WHEN 'D' THEN 'DB2'
  WHEN 'U' THEN 'USER'
  WHEN 'C' THEN 'DEFINER'
  WHEN ' ' THEN ' ' END
CASE COMMIT_ON_RETURN
  WHEN 'N' THEN 'NO'
  WHEN 'Y' THEN 'YES'
  WHEN ' ' THEN ' ' END
END
CASE SPECIAL_REGS
  WHEN 'D' THEN 'DEFAULT SPECIAL REGISTERS'
  WHEN 'I' THEN 'INHERIT SPECIAL REGISTERS'
  WHEN ' ' THEN ' ' END
END
MAX_FAILURE
INTO hvLANGUAGE
, hvCOLLID
, hvDETERMINISTIC
, hvNULL_CALL
, hvPARAMETER_STYLE
, hvFENCED
, hvSQLDATAACCESS
, hvSTAYRESIDENT
, hvASUTIME
, hvWLMENVIRONMENT
, hvPROGRAM_TYPE
, hvEXTERNAL_SECURITY
, hvCOMMIT_ON_RETURN
, hvRESULTSETS
, hvEXTERNAL_NAME
, hvRUNOPTS
, hvSPECIAL_REGS
, hvMAX_FAILURE
FROM SYSIBM.SYSROUTINES
WHERE SCHEMA = spSCHEMA
  AND NAME = spNAME;
CASE WHEN END_TABLE = 1 THEN
  SIGNAL SQLSTATE '38002'
  SET MESSAGE_TEXT = 'Requested object ''
  || spSCHEMA || '.
  || spNAME
  || '' not found';
  WHEN hvROUTINETYPE <> 'P' THEN
  SIGNAL SQLSTATE '38002'
  SET MESSAGE_TEXT = 'Object is not a stored procedure';
  WHEN hvORIGIN <> 'E' THEN
  SIGNAL SQLSTATE '38004'
  SET MESSAGE_TEXT = 'Object is not an external stored procedure';
ELSE -- NOOP below provided to satisfy requirement for ELSE clause
  SET ROW_SEQUENCE = ROW_SEQUENCE;
END CASE;
SET END_TABLE = 0;
SET OPERATION = 'OPEN CURSOR';
OPEN SYSPARM_CURSOR;
SET OPERATION = 'FIRST FETCH';
FETCH SYSPARM_CURSOR INTO hvPARAMNAME
, hvROWTYPE
, hvORDINAL
, hvTYPENAME
, hvLENGTH
, hvSCALE
, hvSUBTYPE
, hvENCODING_SCHEMA;
-- Output the CREATE PROCEDURE clause
IF END_TABLE = 0 THEN
  SET LINE = 'CREATE PROCEDURE ' || spSCHEMA || '.
  || spNAME;
  SET RETURN_POINT = 'A100';
GOTO INSERTLINE;
END IF;

A100: -- Build and output the parameter list
SET LINE = ' ( '; 03010000
WHILE END_TABLE = 0 DO 03020000
  -- Output the parameter type (IN, OUT, or INOUT)
  SET LINE = LINE 03030000
  || hvROWTYPE  || ', ' 03040000
  || hvPARMNAME  || ' ' 03050000
  || RTRIM(hvTYPENAME); 03060000
CASE 03070000
  WHEN hvTYPENAME = 'DECIMAL'
  OR hvTYPENAME = 'DEC' 03100000
  OR hvTYPENAME = 'NUMERIC' THEN 03110000
    SET LINE = LINE || '(' || VARCHAR(hvLENGTH) 03130000
    || ' ' || VARCHAR(hvSCALE) || ' '); 03140000
  WHEN hvTYPENAME = 'FLOAT' THEN 03160000
    SET LINE = LINE || '(' || VARCHAR(hvLENGTH) || ' '); 03170000
  WHEN hvTYPENAME = 'CHARACTER' 03190000
  OR hvTYPENAME = 'CHAR' 03200000
  OR hvTYPENAME = 'CHARACTER VARYING' 03210000
  OR hvTYPENAME = 'CHAR VARYING' 03220000
  OR hvTYPENAME = 'VARCHAR' 03230000
  OR hvTYPENAME = 'CHARACTER LARGE OBJECT' 03240000
  OR hvTYPENAME = 'CHAR LARGE OBJECT' 03250000
  OR hvTYPENAME = 'CLOB' 03260000
  OR hvTYPENAME = 'GRAPHIC' 03270000
  OR hvTYPENAME = 'VARCHAR' 03280000
  OR hvTYPENAME = 'DBCLOB' 03290000
  OR hvTYPENAME = 'BINARY LARGE OBJECT' 03300000
  OR hvTYPENAME = 'BLOB' THEN 03310000
    SET LINE = LINE || '(' || VARCHAR(hvLENGTH) || ' '); 03320000
    END CASE; 03330000
  ELSE -- busy statement below required to handle ELSE case 03340000
    SET ROW_SEQUENCE = ROW_SEQUENCE; 03350000
    END CASE; 03360000
    IF hvSUBTYPE <> ' ' THEN 03370000
      SET LINE = LINE || '(' || VARCHAR(hvLENGTH) || ' '); 03380000
      END IF; 03390000
    IF hvENCODING_SCHEME <> ' ' THEN 03400000
      SET LINE = LINE || '(' || VARCHAR(hvSCALE) || ' '); 03410000
      END IF; 03420000
    SET RETURN_POINT = 'B100'; 03430000
    GOTO INSERTLINE; 03440000
B100: -- Fetch the next parameter
SET OPERATION = 'FETCH'; 03450000
FETCH SYSPARMS_CURSOR 03460000
INTO hvPARMNAME 03470000
, hvROWTYPE 03480000
, hvORDINAL 03490000
, hvTYPENAME 03500000
, hvLENGTH 03510000
, hvSCALE 03520000
, hvSUBTYPE 03530000
, hvENCODING_SCHEME; 03540000
SET LINE = ' ( ' 03550000
END WHILE; 03560000
SET OPERATION = 'CLOSE CURSOR'; 03570000
CLOSE SYSPARMS_CURSOR; 03580000
-- Close the parameter list
SET LINE = ' ) '; 03590000
SET RETURN_POINT = 'C100'; 03600000
GOTO INSERTLINE; 03610000
C100: -- Build remaining clauses for the CREATE PROCEDURE statement 03620000
-- Output the RESULT_SETS clause
IF hvRESULT_SETS > 0 THEN 03630000
  SET LINE = 'DYNAMIC RESULT SETS ' || VARCHAR(hvRESULT_SETS); 03640000
  SET RETURN_POINT = 'D100'; 03650000
  GOTO INSERTLINE; 03660000
END IF; 03670000
D100: -- Output the EXTERNAL NAME clause
SET LINE = ' EXTERNAL NAME ' || RTRIM(hvEXTERNAL_NAME); 03680000
SET RETURN_POINT = 'E100'; 03690000

Chapter 12. Sample data and applications supplied with Db2 for z/OS 1177
GOTO INSERTLINE;

E100: -- Output the LANGUAGE clause
SET LINE = 'LANGUAGE ' || RTRIM(hvLANGUAGE);
GOTO INSERTLINE;

F100: -- Output the SQL data access type clause
IF hvSQL_DATA_ACCESS <> '' THEN
SET LINE = 'SQL_DATA_ACCESS';
GOTO INSERTLINE;
END IF;

G100: -- Output the PARAMETER STYLE clause
IF hvPARAMETER_STYLE <> '' THEN
SET LINE = 'PARAMETER STYLE ' || hvPARAMETER_STYLE;
GOTO INSERTLINE;
END IF;

H100: -- Output the DETERMINISTIC clause
IF hvDETERMINISTIC <> '' THEN
SET LINE = hvDETERMINISTIC;
GOTO INSERTLINE;
END IF;

I100: -- Output the COLLID clause
IF hvCOLLID <> '' THEN
SET LINE = 'COLLID ' || RTRIM(hvCOLLID);
GOTO INSERTLINE;
END IF;

J100: -- Output the WLM ENVIRONMENT clause
SET LINE = 'WLM_ENVIRONMENT ' || RTRIM(hvWLM_ENVIRONMENT);
GOTO INSERTLINE;

K100: -- Output the ASUTIME clause
IF hvASUTIME <> 0 THEN
SET LINE = 'ASUTIME ' || VARCHAR(hvASUTIME);
GOTO INSERTLINE;
ELSE
SET LINE = 'NO ASUTIME';
GOTO INSERTLINE;
END IF;

L100: -- Output the LANGUAGE clause
IF hvLANGUAGE <> '' THEN
SET LINE = 'LANGUAGE ' || RTRIM(hvLANGUAGE);
GOTO INSERTLINE;
ELSE
SET LINE = 'NO LANGUAGE';
GOTO INSERTLINE;
END IF;

M100: -- Output the STAY RESIDENT clause
IF hvSTAYRESIDENT <> '' THEN
SET LINE = 'STAY RESIDENT ' || hvSTAYRESIDENT;
GOTO INSERTLINE;
END IF;

N100: -- Output the PROGRAM TYPE clause
IF hvPROGRAM_TYPE <> '' THEN
SET LINE = 'PROGRAM TYPE ' || hvPROGRAM_TYPE;
GOTO INSERTLINE;
END IF;

O100: -- Output the EXTERNAL SECURITY clause
IF hvEXTERNAL_SECURITY <> '' THEN
SET LINE = 'EXTERNAL_SECURITY';
GOTO INSERTLINE;
END IF;

P100: -- Output the AFTER FAILURE clause
IF hvMAX_FAILURE = -1 THEN
SET LINE = 'STOP AFTER SYSTEM DEFAULT FAILURES';
ELSEIF hvMAX_FAILURE = 0 THEN
    SET LINE = 'CONTINUE AFTER FAILURE'
    ELSE
    SET LINE = 'STOP AFTER ' || VARCHAR(hvMAX_FAILURE) || ' FAILURES'
    END IF;
SET RETURN_POINT = 'Q100'
GOTO INSERTLINE;

Q100: -- Output the RUN OPTIONS clause
IF hvRUNOPTS <> ' ' THEN
    SET LINE = 'RUN OPTIONS ' || hvRUNOPTS || ' '||;
    SET RETURN_POINT = 'R100';
    GOTO INSERTLINE;
END IF;

R100: -- Output the COMMIT ON RETURN clause
IF hvCOMMIT_ON_RETURN <> ' ' THEN
    SET LINE = 'COMMIT ON RETURN ' || hvCOMMIT_ON_RETURN;
    SET RETURN_POINT = 'S100';
    GOTO INSERTLINE;
END IF;

S100: -- Output the SPECIAL REGISTERS clause
IF hvSPECIAL_REGS <> ' ' THEN
    SET LINE = hvSPECIAL_REGS;
    SET RETURN_POINT = 'T100';
    GOTO INSERTLINE;
END IF;

T100: -- Output the CALLED ON NULL INPUT clause
IF hvNULL_CALL = 'Y' THEN
    SET LINE = 'CALLED ON NULL INPUT';
    GOTO INSERTLINE;
SET RETURN_POINT = 'U100';

U100: -- Finish up
GOTO DONE;

INSERTLINE:
SET LINE_LENGTH = LENGTH(LINE);
WHILE LINE_LENGTH > 72 DO
    SET ROW = SUBSTR(LINE, 1, 72) || REPEAT(' ', 8);
    SET LINE_LENGTH = LENGTH(LINE);
    END WHILE;
SET ROW = SUBSTR(LINE || REPEAT(' ', 80)), 1, 80);
SET ROW_SEQUENCE = ROW_SEQUENCE + 1;
INSERT INTO DSN8.DSN8ES3_RS_TBL
VALUES( P1.ROW_SEQUENCE, P1.ROW );

CASE RETURN_POINT
WHEN 'A100' THEN GOTO A100;
WHEN 'B100' THEN GOTO B100;
WHEN 'C100' THEN GOTO C100;
WHEN 'D100' THEN GOTO D100;
WHEN 'E100' THEN GOTO E100;
WHEN 'F100' THEN GOTO F100;
WHEN 'G100' THEN GOTO G100;
WHEN 'H100' THEN GOTO H100;
WHEN 'I100' THEN GOTO I100;
WHEN 'J100' THEN GOTO J100;
WHEN 'K100' THEN GOTO K100;
WHEN 'L100' THEN GOTO L100;
WHEN 'M100' THEN GOTO M100;
WHEN 'N100' THEN GOTO N100;
WHEN 'O100' THEN GOTO O100;
WHEN 'P100' THEN GOTO P100;
WHEN 'Q100' THEN GOTO Q100;
WHEN 'R100' THEN GOTO R100;
WHEN 'S100' THEN GOTO S100;
END IF;
Related reference

“Sample applications in TSO” on page 1027

A set of Db2 sample applications run in the TSO environment.

**DSN8DUAD**

Returns the current date in one these 34 formats.

```sql
/****************************
* Module name = DSN8DUAD (DB2 sample program)
* DESSCRIPTIVE NAME = Current date reformatter (UDF)
* LICENSED MATERIALS - PROPERTY OF IBM
* (C) COPYRIGHT 1998, 2003 IBM CORP. ALL RIGHTS RESERVED.
* STATUS = VERSION 8
* Module size: See linkedit output
* Processor: IBM C/C++ for OS/390 V1R3 or higher
* Purpose: See Function
* Linkage: Invoked via SQL UDF call
* Input: Parameters explicitly passed to this function:
  - *format : pointer to a char[14], null-terminated string having the desired

**Function:** Returns the current date in one these 34 formats:

```
format for the current date (see "Function", above, for valid formats)
- *niFormat : pointer to a short integer variable for the null indicator variable for
- *fnName : pointer to a char[138], null-terminated string having the UDF family name of this function.
- *specificName : pointer to a char[129], null-terminated string having the UDF specific name of this function.

Output: Parameters explicitly passed by this function:
- *dateOut : pointer to a char[18], null-terminated string to receive the current date in the formatted indicated by *format.
- *niDateOut : pointer to a short integer to receive the null indicator variable for *dateOut.
- *sqlstate : pointer to a char[66], null-terminated string to receive the SQLSTATE.
- *message : pointer to a char[76], null-terminated string to receive a diagnostic message if one is generated by this function.

No normal Exit: Return Code: SQLSTATE = 00000
- Message: none

Error Exit: Return Code: SQLSTATE = 38601
- Message: DSN8DUAD Error: No output format entered
- Message: DSN8DUAD Error: Unknown format specified

External References:
- Routines/Services: None
- Data areas : None
- Control blocks : None

Pseudocode:
- DSN8DUAD: 
  - Verify that a valid format for the current date was received:
  - if *format is blank or niFormat is not 0, no format passed:
    - issue SQLSTATE 38601 and a diagnostic message
  - Verify that a valid format for the current date was received:
  - Call formatDate to convert the current date in the indicated format
  - If no errors, unset null indicators, and return SQLSTATE 00000
  else set null indicator and return null date out.

End DSN8DUAD

Output: Parameters explicitly passed by this function:
- *fnName : pointer to a char[138], null-terminated string having the UDF family name of this function.
- *specificName : pointer to a char[129], null-terminated string having the UDF specific name of this function.

- *message : pointer to a char[70], null-terminated string to receive the SQLSTATE.
- *sqlstate : pointer to a char[06], null-terminated string to receive the null indicator variable for
- *niDateOut : pointer to a short integer to receive a diagnostic message if one is generated by this function.
- *dateOut : pointer to a char[18], null-terminated string to receive the current date in the formatted indicated by
- *format. 

D1181
* - convert the month (01-12) to a roman numeral (I-XII).  
* End romanMonth  
*  
* Change Log:  
* 2002/10/17 PQ66488 Fix date truncation error  
* - romanMonth  
*  
* Assumptions:  
* - format points to a char[14], null-terminated string  
* - *format points to a char[14], null-terminated string  
* - *dateOut points to a char[18], null-terminated string  
* - *niFormat points to a short integer  
* - *niDateOut points to a short integer  
* - *sqlstate points to a char[06], null-terminated string  
* - *fnName points to a char[138], null-terminated string  
* - *specificName points to a char[129], null-terminated string  
* - *message points to a char[79], null-terminated string  
*  
#pragma link (DSN8DUAD, fetchable)  

/******************** DSN8DUAD functions ********************/  
void DSN8DUAD  
(/ main routine  
 ( char *format,  
  / in: format for dateOut  
  char *dateOut,  
  / out: formmatted current date  
  short int *niFormat,  
  / in: indic var, format  
  short int *niDateOut,  
  / out: indic var for dateOut  
  char *sqlstate,  
  / out: SQLSTATE  
  char *fnName,  
  / in: family name of function  
  char *specificName,  
  / in: specific name of func  
  char *message  
  / out: diagnostic message  
  ));  
int formatDate  
(/ format the current date  
 ( char *dateOut,  
  / out: formmatted curr date  
  char *message,  
  / out: diagnostic message  
  char *sqlstate,  
  / out: SQLSTATE  
  char *format  
  / in: desired format  
  ));  
void getDate  
(/ gets curr date, formatted  
 ( char *dateOut,  
  / out: formmatted curr date  
  char *dateFmt,  
  / in: desired date format  
  ));  
void Remove0  
(/ remove 0 from indic. byte  
 ( char *string,  
  / in/out: character string  
  short int loc  
  / in: loc'n of zero to remove  
  ));  
char romanMonth();  
(/ get roman# of curr month# 

#include <time.h>  
#include <string.h>  
#include <stdio.h>  

#pragma linkage (DSN8DUAD, fetchable)  

*********************************************************************/  
01490000  
01500000  
01510000  
01520000  
01530000  
01540000  
01550000  
01560000  
01570000  
01580000  
01590000  
01600000  
01610000  
01620000  
01630000  
01640000  
01650000  
01660000  
01670000  
01680000  
01690000  
01700000  
01710000  
01720000  
01730000  
01740000  
01750000  
01760000  
01770000  
01780000  
01790000  
01800000  
01810000  
01820000  
01830000  
01840000  
01850000  
01860000  
01870000  
01880000  
01890000  
01900000  
01910000  
01920000  
01930000  
01940000  
01950000  
01960000  
01970000  
01980000  
01990000  
02000000  
02010000  
02020000  
02030000  
02040000  
02050000  
02060000  
02070000  
02080000  
02090000  
02100000  
02110000  
02120000  
02130000  
02140000  
02150000  
02160000  
02170000  
02180000  
02190000  
02200000  
02210000  
02220000  
02230000  
02240000  
02250000  
02260000  
02270000  
02280000  
02290000  
02300000
/* Deadline: 03/05/00 */

else if( strcmp( format,"DD MONTH YY" ) == MATCH )

Remove0( dateOut,0 );           /* strip leading 0 if day < 10*/

dateGet( dateOut,"%d %B %Y" );  /* formt date as DD MONTH YYYY*/

} /* end of DSN8DUAD */

return;

/* end of DSN8DUAD */

Chapter 12. Sample data and applications supplied with Db2 for z/OS 1183
```c
{  getDate( dateOut,"%d %B %Y" );  /* format date as DD MONTH YYYY */  return 0;  }
}
else if( strcmp( format,"DD-MONTH YYYY" ) == MATCH )
{  getDate( dateOut,"%d %B YYYY" );  /* format date as DD MONTH YYYY */  return 0;  }
else if( strcmp( format,"D.M.YY" ) == MATCH )
{  getDate( dateOut,"%d.%m.9Y" );  /* format date as DD.MM.YY */  Remove0( dateOut,3 );  /* strip leading 0 if month<10*/  Remove0( dateOut,0 );  /* strip leading 0 if day < 10*/  return 0;  }
else if( strcmp( format,"D.M.YYYY" ) == MATCH )
{  getDate( dateOut,"%d.%m.9Y" );  /* format date as DD.MM.YYY */  Remove0( dateOut,3 );  /* strip leading 0 if month<10*/  Remove0( dateOut,0 );  /* strip leading 0 if day < 10*/  return 0;  }
else if( strcmp( format,"DD.MM.YY" ) == MATCH )
{  getDate( dateOut,"%d-%m-%Y" );  /* format date as DD-MM-YYYY */  return 0;  }
else if( strcmp( format,"DD-MM-YYYY" ) == MATCH )
{  getDate( dateOut,"%d-%m-%y" );  /* format date as DD-MM-YY */  return 0;  }
else if( strcmp( format,"D-M-YY" ) == MATCH )
{  getDate( dateOut,"%d.%m.%Y" );  /* format date as DD.MM.YYYY */  return 0;  }
else if( strcmp( format,"DD.MM.YYYY" ) == MATCH )
{  getDate( dateOut,"%d.%m.%y" );  /* format date as DD.MM.YY */  return 0;  }
else if( strcmp( format,"D.M.YYYY" ) == MATCH )
{  getDate( dateOut,"%d.%m.%Y" );  /* format date as DD.MM.YYYY */  return 0;  }
else if( strcmp( format,"DD-MM-YY" ) == MATCH )
{  getDate( dateOut,"%d-%m-%y" );  /* format date as DD-MM-YY */  return 0;  }
else if( strcmp( format,"D-M-YY" ) == MATCH )
{  getDate( dateOut,"%d.%m.%Y" );  /* format date as DD.MM.YYYY */  return 0;  }
else if( strcmp( format,"DD.MM.YY" ) == MATCH )
{  getDate( dateOut,"%d-%m-%y" );  /* format date as DD-MM-YY */  return 0;  }
else if( strcmp( format,"D.M.YY" ) == MATCH )
{  getDate( dateOut,"%d %B %Y" );  /* format date as DD MONTH YYYY */  return 0;  }
```

1184 Db2 12 for z/OS: Application Programming and SQL Guide
}
03880000
else if( strcmp( format,"MM/DD/YYYY" ) == MATCH )
03890000
{
03900000
getDate( dateOut,"%m/%d/%Y" ); /* format date as MM/DD/YYYY
*/03910000
}
03920000
else if( strcmp( format,"YY/M/D" ) == MATCH )
03930000
{
03940000
getDate( dateOut,"%y/%m/%d" ); /* format date as YY/MM/DD
*/03950000
Remove0( dateOut,6 );
/* strip leading 0 if day < 10*/ 03960000
Remove0( dateOut,3 );
/* strip leading 0 if month<10*/ 03970000
}
03980000
else if( strcmp( format,"YY/MM/DD" ) == MATCH )
03990000
{
04000000
getDate( dateOut,"%y/%m/%d" ); /* format date as YY/MM/DD
*/04010000
}
04020000
else if( strcmp( format,"YYYY/M/D" ) == MATCH )
04030000
{
04040000
getDate( dateOut,"%Y/%m/%d" ); /* format date as YYYY/MM/DD
*/04050000
Remove0( dateOut,8 );
/* strip leading 0 if day < 10*/ 04060000
Remove0( dateOut,5 );
/* strip leading 0 if month<10*/ 04070000
}
04080000
else if( strcmp( format,"YYYY/MM/DD" ) == MATCH )
04090000
{
04100000
getDate( dateOut,"%Y/%m/%d" ); /* format date as YYYY/MM/DD
*/04110000
}
04120000
else if( strcmp( format,"YY.M.D" ) == MATCH )
04130000
{
04140000
getDate( dateOut,"%y.%m.%d" ); /* format date as YY.MM.DD
*/04150000
Remove0( dateOut,6 );
/* strip leading 0 if day < 10*/ 04160000
Remove0( dateOut,3 );
/* strip leading 0 if month<10*/ 04170000
}
04180000
else if( strcmp( format,"YY.MM.DD" ) == MATCH )
04190000
{
04200000
getDate( dateOut,"%y.%m.%d" ); /* format date as YY.MM.DD
*/04210000
}
04220000
else if( strcmp( format,"YYYY.M.D" ) == MATCH )
04230000
{
04240000
getDate( dateOut,"%Y.%m.%d" ); /* format date as YYYY.MM.DD
*/04250000
Remove0( dateOut,8 );
/* strip leading 0 if day < 10*/ 04260000
Remove0( dateOut,5 );
/* strip leading 0 if month<10*/ 04270000
}
04280000
else if( strcmp( format,"YYYY.MM.DD" ) == MATCH )
04290000
{
04300000
getDate( dateOut,"%Y.%m.%d" ); /* format date as YYYY.MM.DD
*/04310000
}
04320000
else if( strcmp( format,"YYYY-M-D" ) == MATCH )
04330000
{
04340000
getDate( dateOut,"%Y-%m-%d" ); /* format date as YYYY-MM-DD
*/04350000
Remove0( dateOut,8 );
/* strip leading 0 if day < 10*/ 04360000
Remove0( dateOut,5 );
/* strip leading 0 if month<10*/ 04370000
}
04380000
else if( strcmp( format,"YYYY-MM-DD" ) == MATCH )
04390000
{
04400000
getDate( dateOut,"%Y-%m-%d" ); /* format date as YYYY-MM-DD
*/04410000
}
04420000
else if( strcmp( format,"YYYY-D-XX" ) == MATCH )
04430000
{
04440000
strcpy( dateFmt, "%Y-%d-" );
/* start format as YYYY-DD*/ 04450000
strcat( dateFmt, romanMonth() );/* append roman# for curr mo. */ 04460000
getDate( dateOut,dateFmt );
/* format date as YYYY-DD-XX */ 04470000
Remove0( dateOut,5 );
/* strip leading 0 if day < 10*/ 04480000
}
04490000
else if( strcmp( format,"YYYY-DD-XX" ) == MATCH )
04500000
{
04510000
strcpy( dateFmt, "%Y-%d-" );
/* start format as YYYY-DD*/ 04520000
strcat( dateFmt, romanMonth() );/* append roman# for curr mo. */ 04530000
getDate( dateOut,dateFmt );
/* format date as YYYY-DD-XX */ 04540000
}
04550000
else if( strcmp( format,"YYYY-XX-D" ) == MATCH )
04560000
{
04570000
strcpy( dateFmt, "%Y-" );
/* start format as YYYY*/ 04580000
strcat( dateFmt, romanMonth() );/* append roman# for curr mo. */ 04590000
strcat( dateFmt, "-%d" );
/* append -DD to format
*/ 04600000
getDate( dateOut,dateFmt );
/* get date as YYYY-XX-DD
*/ 04610000
Remove0( dateOut,
/* strip leading 0 if day < 10*/ 04620000
strlen(dateFmt) );
04630000
}
04640000
else if( strcmp( format,"YYYY-XX-DD" ) == MATCH )
04650000
{
04660000
strcpy( dateFmt, "%Y-" );
/* start format as YYYY*/ 04670000
strcat( dateFmt, romanMonth() );/* append roman# for curr mo. */ 04680000
strcat( dateFmt, "-%d" );
/* append -DD to format
*/ 04690000

Chapter 12. Sample data and applications supplied with Db2 for z/OS 1185


**void Remove0**

```c
short int  i;
for( i=loc; i<(strlen(string)-1); i++ )
    string[i] = string[i+1];
return;
```

- Checks `*string` at the location indicated by `loc` and, if a character zero resides there, removes it from `*string` by shifting all bytes to the right of it leftward by 1 byte.
- If `string[loc] == '0'`

**void getDate**

```c
short int   loc                     /* in: loc'n of byte to remove*/
(char        *string,                /* in/out: character string   */
short int  i;                                                        /* len of str rtd by strftime*/
char        dateBuff[19];           /* gets formatted date     @01*/
struct tm   *tmPtr;                 /* ->buff for time.h tm struct*/
time_t      t;                      /* buff for system time macro */
struct tm   *tmPtr;                                               /* ->buff for time.h tm struct*/
char        dateFmt                /* in: desired date format    */
char        *dateOut,               /* out: formatted current date*/
char        *dateFmt                /* in: desired date format    */
char        *dateFmt                /* in: desired date format    */
char        *dateOut,               /* out: formatted current date*/
/
```

- Obtains the current date from the system and formats it according to the format string in `*dateFmt`. The result is placed in `dateOut`. This function uses the C function localtime to obtain the system time and date and the C function strftime to format it according to `*dateFmt`. For this program, the following format tokens were used. See the C/C++ library reference manuals for more info.

- `%Y` = year with century
- `%y` = year with century
- `%m` = month (01-12)
- `%d` = day of the month
- `%B` = full month name
- `%Y` = year with century
- `%m` = month (01-12)
- `%d` = day of the month
- `%B` = full month name

**getDateTime**

```c
getDateTime( dateOut,dateFmt );  /* get date as YYYY-XX-DD */
/
```

- Function to get date as YYYY-XX-DD.
Related reference

“Sample applications in TSO” on page 1027

A set of Db2 sample applications run in the TSO environment.

**DSN8DUAT**

Returns the current time in one these 8 formats.
getTime function:

- Define the output format for the current time.
- If format is blank, set SQLSTATE 38601.
- Call buildTime to generate the output time using the format.
- Remove leading zeroes from the hour.
- Determine which of the time components, delimiters, and AM/PM are included.
- Call getTime to obtain:
  - the current 12-hour clock hour
  - the current 24-hour clock hour
  - the current minute
  - the current second
- Set AM/PM indicator to PM if 24-hour clock hour > 11, else AM.
- Call buildTime to generate the output time using the format.
- Determine which of the time components, delimiters, and AM/PM indicator (if any) to pass.
- If no errors, unset null indicators, and return SQLSTATE 00000.
- End getTime function.

External References:
- Routines/Services: None
- Data areas: None
- Control blocks: None

Pseudocode:
- Verify that a valid format for the current time was received.
  - If format is blank or niFormat is not 0, no format passed.
  - Issue SQLSTATE 38601 and a diagnostic message.
- Call getTime to obtain:
  - the current 12-hour clock hour
  - the current 24-hour clock hour
  - the current minute
  - the current second
- Set AM/PM indicator to PM if 24-hour clock hour > 11, else AM.
- Call buildTime to generate the output time using the format.
- Determine which of the time components, delimiters, and AM/PM indicator (if any) to pass.
- If no errors, unset null indicators, and return SQLSTATE 00000.
- End getTime function.

Calling Time Library Functions:
- Invoke the strftime() library function to format the current time for local time.
- Invoke the strptime() library function to format 12-hour, 24-hour, minute, and second components from the time vector.
- End getTime function.
Chapter 12. Sample data and applications supplied with Db2 for z/OS
```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define OK 0
#define SQLSTATE "38601"
#define AM_PM " AM/PM"

char        hh12[3]; /* current time - hours */
char        hh24[3]; /* current time - hours */
char        mm[3]; /* current time - minutes */
char        ss[3]; /* current time - seconds */
char        suffix[4]; /* AM/PM indicator */
char        niFormat[6]; /* AM/PM indicator */
char        timeOut[12]; /* DSN8DUMN run status */
char        message[70]; /* AM/PM indicator */

short int   status = OK; /* DSN8DUMN run status */

#define DSN8DUAT_ERROR "DSN8DUAT Error: No format entered"
#define DSN8DUAT_SQLSTATE "38661"

/***************************************************************/
* Format the current time according to the input format     *
/***************************************************************/
if( *niFormat || ( strlen( format ) == 0 ) )
    
    strcpy( message, DSN8DUAT_ERROR );
    strcpy( sqlstate, DSN8DUAT_SQLSTATE );

/***************************************************************/
* Get current time in the specified format                   *
/***************************************************************/
if( status == OK )
    
   铚仟饥 Chucktime( hh24, hh12, mm, ss );
    
    /***************************************************************/
* Get the current hour (hh), minute (mm), and second (ss)    *
/***************************************************************/
getTime( hh12, hh24, mm, ss );
    
    /***************************************************************/
* Suffix is AM unless it’s noon or later on the 24-hour clock *
/***************************************************************/
if( strcmp( hh24,"11" ) > 0 )
    
    strcpy( suffix," PM" );

/***************************************************************/
* Format the current time according to the input format      *
/***************************************************************/
if( strcmp( format,"H:MM AM/PM" ) == MATCH )
    
    remove0prefix( hh12 );
    
    buildTime( timeOut,hh12,mm,"",":",suffix );

else if( strcmp( format,"HH:MM AM/PM" ) == MATCH )
    
    buildTime( timeOut,hh12,mm,"","",suffix );

else if( strcmp( format,"HH:MM" ) == MATCH )
    
    buildTime( timeOut,hh24,mm,ss,":","" );
    
    remove0prefix( hh24 );
    
    buildTime( timeOut,hh24,mm,ss,"",":","" );

else if( strcmp( format,"HH.MM" ) == MATCH )
    
    buildTime( timeOut,hh24,mm,ss,"",":","" );

else if( strcmp( format,"HH.MM:SS" ) == MATCH )
    
    buildTime( timeOut,hh24,mm,ss,"","","" );

else if( strcmp( format,"HH:MM:SS AM/PM" ) == MATCH )
    
    buildTime( timeOut,hh12,mm,"","",suffix );

else if( strcmp( format,"HH:MM AM/PM" ) == MATCH )
    
    buildTime( timeOut,hh12,mm,"",":","" );

else if( strcmp( format,"H:MM AM/PM" ) == MATCH )
    
    remove0prefix( hh12 );
    
    buildTime( timeOut,hh12,mm,"",":",suffix );

else if( strcmp( format,"HH:MM" ) == MATCH )
    
    buildTime( timeOut,hh24,mm,ss,"",":","" );
    
    remove0prefix( hh24 );
    
    buildTime( timeOut,hh24,mm,ss,"","","" );

else if( strcmp( format,"HH.MM:SS" ) == MATCH )
    
    buildTime( timeOut,hh24,mm,ss,"","","" );

else if( strcmp( format,"HH:MM:ss" ) == MATCH )
    
    buildTime( timeOut,hh24,mm,ss,":","" );

else if( strcmp( format,"H:MM:ss" ) == MATCH )
    
    buildTime( timeOut,hh24,mm,ss,":","","" );

else if( strcmp( format,"HH:MM:ss" ) == MATCH )
    
    buildTime( timeOut,hh24,mm,ss,"",":","" );

else if( strcmp( format,"HH:MM:SS" ) == MATCH )
    
    buildTime( timeOut,hh24,mm,ss,"","","" );

else if( strcmp( format,"HH:MM:ss:AM/PM" ) == MATCH )
    
    buildTime( timeOut,hh24,mm,ss,":",":",suffix );

else if( strcmp( format,"HH:MM:ss AM/PM" ) == MATCH )
    
    buildTime( timeOut,hh24,mm,ss,":",":","" );

else if( strcmp( format,"H:MM:ss:AM/PM" ) == MATCH )
    
    buildTime( timeOut,hh24,mm,ss,"",":","" );

else if( strcmp( format,"HH:MM:SS:AM/PM" ) == MATCH )
    
    buildTime( timeOut,hh24,mm,ss,"","",suffix );

else if( strcmp( format,"HH:MM:SS AM/PM" ) == MATCH )
    
    buildTime( timeOut,hh24,mm,ss,"","","" );

else if( strcmp( format,"HH:MM:SS:ss" ) == MATCH )
    
    buildTime( timeOut,hh24,mm,ss,"","","" );

else if( strcmp( format,"HH:MM:ss:AM/PM" ) == MATCH )
    
    buildTime( timeOut,hh24,mm,ss,":",":","" );

else if( strcmp( format,"HH:MM:ss AM/PM" ) == MATCH )
    
    buildTime( timeOut,hh24,mm,ss,":",":","" );

else if( strcmp( format,"H:MM:ss:AM/PM" ) == MATCH )
    
    buildTime( timeOut,hh24,mm,ss,":",":","" );
```

1190 Db2 12 for z/OS: Application Programming and SQL Guide
Chapter 12. Sample data and applications supplied with Db2 for z/OS

```c
#include <string.h>
#include <time.h>

void getTime()
{
    char *hh12, /* out: hours (12 hour clock) */
    *hh24,      /* out: hours (24 hour clock) */
    *mm,        /* out: minutes */
    *ss;        /* out: seconds */

    time_t t;
    struct tm *tmptr;

    tmptr = localtime(&t);
    strftime(timeBuf, sizeof(timeBuf)-1, "%I/%H/%M/%S", tmptr);

    *hh12 = strftime("%I", tmptr);
    *hh24 = strftime("%H", tmptr);
    *mm = strftime("%M", tmptr);
    *ss = strftime("%S", tmptr);
}
```
tokPtr = strtok( NULL,"/ " );     /* Parse to 3rd slash */ 03700000
strcpy( mm,tokPtr );             /* for minutes */ 03708999

tokPtr = strtok( NULL,"/ " );     /* Parse remaining bytes */ 03720000
strcpy( ss,tokPtr );             /* for seconds */ 03728999
}
} /* end getTime */ 03770000

void buildTime 03800000
(char *timeStr,               /* out: reformatted time */ 03810000
char *hh,                    /* in: hours component */ 03820000
char *mm,                    /* in: minutes component */ 03830000
char *ss,                    /* in: seconds component */ 03840000
char *delim,                 /* in: delimiter of choice */ 03850000
char *suffix                  /* in: AM/PM suffix (if any) */ 03860000
) 03870000
{
  strcpy( timeStr,hh );       /* Start with hours ... */ 03940000
  strcat( timeStr,delim );    /* append the delimiter */ 03950000
  strcat( timeStr,mm );       /* append minutes */ 03960000
  if( strlen(ss) > 0 )        /* and, if seconds specified, */ 03970000
  {                          /* ..append the delimiter */ 03980000
    strcat( timeStr,delim );/* ..append seconds */ 03990000
  }                          /* and, if suffix specified, */ 04000000
  strcat( timeStr,suffix );  /* ..append it. */ 04010000
}
} /* end buildTime */ 04080000

void remove0prefix 04110000
(char *string                 /* in/out: character string */ 04120000
) 04130000
{
short int   i = 0;  /* Loop control */ 04180000
short int   j = 0;  /* Loop control */ 04200000

  if( string[0] == '0' ) 04220000
  for( i=0; string[i] == '0'; i++ ); 04230000

  if( string[i] == '0' ) 04240000
  strcat( string,"0" ); 04270000

  else 04300000
  {
    for( j=0; string[i] != NULLCHAR; j++ ) 04310000
      string[j] = string[i++]; 04320000
    string[j] = NULLCHAR; 04330000
  }
}
} /* end remove0prefix */ 04420000

Related reference
“Sample applications in TSO” on page 1027
A set of Db2 sample applications run in the TSO environment.

**DSN8DUCD**

Converts a given date from one to another of these 34 formats.

```c
/*                      *       00010000
 * Module name = DSN8DUCD (DB2 sample program)     00020000
 * *       00030000
 * * DESRIPTIVE NAME = General date reformatter (UDF)      00040000
 * *       00050000
 * * LICENSED MATERIALS - PROPERTY OF IBM               00060000
 * *      5675-D2B  00070000
 * * (C) COPYRIGHT 2000 IBM CORP. ALL RIGHTS RESERVED.  00080000
 * *       00090000
 * STATUS = VERSION 7                                    00100000
 * *       00110000
 * * Function: Converts a given date from one to another of these 34 00120000
 * * formats:                                             00130000
 * *       00140000
 * *       D Month YY  D Month YYYY  DD Month YY  DD Month YYYY  00150000
 * *       D.M.YY  D.M.YYYY  DD.MM.YY  DD.MM.YYYY  00160000
 * *       D-M-YY  D-M-YYYY  DD-MM-YY  DD-MM-YYYY  00170000
 * *       D/M/YY  D/M/YYYY  DD/MM/YY  DD/MM/YYYY  00180000
 * *       YY.M.D  YYYY.M.D  YY/MM/DD  YYYY/MM/DD  00190000
 * *       YY.M.D  YYYY-MM-DD  00200000
 * *       YY/MM/DD  YYYY/MM/DD  00210000
 * *       YYYY-MM-DD  00220000
 * *       YYYY-DD-XX  YYYY-MM-DD  00230000
 * *       YYYY-XX-DD  YYYY-DD-XX  00240000
 * *       YYYY-DD-XX  00250000
 * *       YYYY-MM-DD  00260000
 * *       YYYY/mm/dd  YYYY/MM/DD  00270000
 * *       yyyy-mm-dd  00280000
 * *       yyyy-mm-mm  00290000
 * *       yyyy-mm-dd  00300000
 * *       yyyy-mm/dd  00310000
 * *       yyyy/mm/dd  00320000
 * *       yyyy-mm-dd  00330000
 * *       yyyy-mm-dd  00340000
 * *       yyyy-mm-dd  00350000
 * *       yyyy-mm-dd  00360000
 * *       yyyy-mm-dd  00370000
 * *       yyyy-mm-d  00380000
 * *       yyyy-mm-dd  00390000
 * * where:                                               00400000
 * *       D: Suppress leading zero if the day is less than 10  00410000
 * *       DD: Retain leading zero if the day is less than 10   00420000
 * *       MM: Suppress leading zero if the month is less than 10  00430000
 * *       MONTH: Use English-language name of month           00440000
 * *       XX: Use a capital Roman numeral for month         00450000
 * *       YY: Use non-century year format                   00460000
 * *       YYYY: Use century year format                    00470000
 * * Example invocation:                                  00480000
 * * EXEC SQL SET :newDate = ALTDATE( "3/15/1947",       00490000
 * *       "M/D/YYYY",                                       00500000
 * *       "DD MONTH YY" );                                 00510000
 * * Notes:                                               00520000
 * * Dependencies: Requires IBM C/C++ for OS/390 V1R3 or higher  00530000
 * * Restrictions:                                        00540000
 * * Module type: C program                             00550000
 * * Processor: IBM C/C++ for OS/390 V1R3 or higher     00560000
 * * Module size: See linkedit output                   00570000
 * * Attributes: Re-entrant and re-usable               00580000
 * * Entry Point: DSN8DUCD                               00590000
 * * Purpose: See Function                              00600000
 * * Linkage: DB2SQL                                     00610000
 * * Input: Parameters explicitly passed to this function: 00620000
 * *       - *dateIn : pointer to a char[18], null-termi-   00630000
 * *       - *formatIn : pointer to a char[14], null-termi- 00640000
 * *       - *formatOut : pointer to a char[14], null-termi- 00650000
 * *       - *nIDateIn : pointer to a short integer having 00660000
 * *       - *nIDateOut : pointer to a short integer having 00670000
 * *       - *nIDateOut : pointer to a short integer having 00680000
 * *       - *nIDateOut : pointer to a short integer having 00690000
 * *       - *nIDateOut : pointer to a short integer having 00700000
 * *       - *nIDateOut : pointer to a short integer having 00710000
 * *       - *nIDateOut : pointer to a short integer having 00720000
 * *       - *nIDateOut : pointer to a short integer having 00730000
 * *       - *nIDateOut : pointer to a short integer having 00740000
 * *       - *nIDateOut : pointer to a short integer having 00750000
 * *       - *nIDateOut : pointer to a short integer having 00760000
 * *       - *nIDateOut : pointer to a short integer having 00770000
 * *       - *nIDateOut : pointer to a short integer having 00780000
 * *       - *nIDateOut : pointer to a short integer having 00790000
 * *       - *nIDateOut : pointer to a short integer having 00800000
 * *       - *nIDateOut : pointer to a short integer having 00810000
 */
```
Output: Parameters explicitly passed by this function:

- *niFormatIn: pointer to a short integer having the null indicator variable for the input format
- *formatIn: pointer to a char[138], null-terminated string having the UDF family name of this function.
- *specificName: pointer to a char[129], null-terminated string having the UDF specific name of this function.

Normal Exit: Return Code: SQLSTATE = 00000

Error Exit: Return Code: SQLSTATE = 38601

External References:
- Routines/Services: None
- Data areas: None
- Control blocks: None

Pseudocode:
- DSNN8DUCD: 01450000
- Issue sqlstate 38601 and a diagnostic message if no input date was provided.
- Issue sqlstate 38601 and a diagnostic message if no input format was provided.
- Issue sqlstate 38601 and a diagnostic message if no output format was provided.
- Call deconDate to deconstruct the input date into year, month, and day components according to the input format.
- Call reconDate to create an output date from the year, month, and day components according to the output format.
- If no errors, unset null indicators, and return SQLSTATE 00000 else set null indicator and return null date out.
- End DSNN8DUCD
- deconDate
- Parse day, month, and year (sequence unknown) components from the input date by breaking on delimiters (blank, /, , ., and -).
- Use the input format to determine sequence of date components.
- if format invalid, issue SQLSTATE 38602 and a diagnostic message.
- Call checkDay to validate the day component
- if not valid day, issue SQLSTATE 38602 and a diag. message
- Call checkMonth to validate the month component and convert it
  (if required) from a calendar month name or roman numeral to
  a month number (1-12).
- if not valid month, issue SQLSTATE 38602 and a diag. message
- Call checkYear to validate the year component.
- if not valid year, issue SQLSTATE 38602 and a diag. message
End deconDate

End reconDate
- Use the output format to edit and sequence the date components
- call add0prefix to prepend leading 0's to the day and/or
  month component(s), as appropriate
- or call remove0prefix to drop leading 0's from the day and/or
  month component(s), as appropriate
- call nameMonth to convert the month number (1-12) to calendar
  month name, if appropriate
- call romanMonth to convert the month number (1-12) to roman
  numeral, if appropriate
- call addCentury to convert a non-century year to century
  date, if appropriate
- call removeCentury to convert a century year to non-century
  if appropriate
- convert the month to a calendar name or roman numeral, if appropriate
- convert the year to a non-century format, if appropriate
- if output format is invalid, issue SQLSTATE 38603 and a diagnostic message
- Call buildDate to create the output date from the edited, re-
  sequenced date components
End buildDate

End romaMonth
End nameMonth
End buildDate

- convert a month number (1-12) to the corresponding month no.
in the standard form, 1-12.
- call unnameMonth to convert the month name to the corresponding month no.
in the standard form, 1-12.
End nameMonth

unnameMonth
- convert a calendar month name to the corresponding month no.
in the standard form, 1-12.
End unnameMonth

romanMonth
- convert a month in the standard form, 1-12, to the corresponding calendar month name.
End romanMonth

unromanMonth
- convert a roman numeral (I-XII) to the corresponding month no.
in the standard form, 1-12.
End unromanMonth

checkYear
- Verify that the year component of the input date is one of the following, in accordance with the input format:
- A valid century year (0000-9999)
- A valid non-century year (00-99)
- If not valid, set error flag and return null value for year
End checkYear

checkMonth
- Verify that the month component of the input date in accordance with the input format:
- if the month is a calendar name, call unnameMonth to convert it to a month number (1-12).
- if the month is a roman numeral, call unromanMonth to convert it to a month number (1-12).
- If not valid, set error flag and return null value for month
End checkMonth

checkDay
- Verify that the day component of the input date is one or two numeric characters
- If not valid, set error flag and return null value for day
End checkDay
char *mo, /* out: month component */
(char *yr, /* out: year component */
int deconDate /* get yr, mo, dy from dateIn */
);

char *message /* out: diagnostic message */
char *specificName, /* in: specific name of func */
char *fnName, /* in: family name of function */
char *sqlstate, /* out: SQLSTATE */
short int *nullDateOut /* out: indic var for dateOut */
short int *nullFormatOut /* in: indic var, formatOut */
short int *nullFormatIn /* in: indic var for formatIn */
short int *nullDateIn /* in: indic var for dateIn */
char *dateOut /* out: reformatted date */
char *formatOut /* in: format for dateOut */
char *formatIn /* in: format of dateIn */
(char *dateIn, /* in: date to be converted */
void DSN8DUCD /* main routine */
( char *dateIn, /* in: date to be converted */
char *formatIn, /* in: format of dateIn */
char *formatOut, /* in: format for dateOut */
char *dateOut, /* out: reformatted date */
short int *nullDateIn, /* in: indic var for dateIn */
short int *nullFormatIn, /* in: indic var for formatIn */
short int *nullDateOut, /* out: indic var for dateOut */
char *sqlstate, /* out: SQLSTATE */
char *fnName, /* in: family name of function */
char *specificName, /* in: specific name of func */
char *message /* out: diagnostic message */
);
int deconDate /* get yr, mo, dy from dateIn */
(char *yr, /* out: year component */
char *mo, /* out: month component */
char *dateIn, /* in: date to be converted */
void DSN8DUCD /* main routine */
( char *dateIn, /* in: date to be converted */
char *formatIn, /* in: format of dateIn */
char *formatOut, /* in: format for dateOut */
char *dateOut, /* out: reformatted date */
short int *nullDateIn, /* in: indic var for dateIn */
short int *nullFormatIn, /* in: indic var for formatIn */
short int *nullDateOut, /* out: indic var for dateOut */
char *sqlstate, /* out: SQLSTATE */
char *fnName, /* in: family name of function */
char *specificName, /* in: specific name of func */
char *message /* out: diagnostic message */
);
int deconDate /* get yr, mo, dy from dateIn */
(char *yr, /* out: year component */
char *mo, /* out: month component */
Chapter 12. Sample data and applications supplied with Db2 for z/OS
short int *nullDateIn,    /* in: indic var for dateIn */ 04090000
short int *nullFormatIn, /* in: indic var for formatIn */ 04100000
short int *nullFormatOut, /* in: indic var, formatOut */ 04110000
short int *nullDateOut,   /* out: indic var for dateOut */ 04120000
char *sqlstate,             /* out: SQLSTATE */ 04130000
char *fnName,               /* in: family name of function */ 04140000
char *specificName,         /* in: specific name of func */ 04150000
char *message,             /* out: diagnostic message */ 04160000
}

Assumptions:
* - *dateIn points to a char[18], null-terminated string 04219000
* - *formatIn points to a char[14], null-terminated string 04229000
* - *dateOut points to a char[18], null-terminated string 04239000
* - *nullDateIn points to a short integer 04250000
* - *nullFormatIn points to a short integer 04260000
* - *nullFormatOut points to a short integer 04270000
* - *nullDateOut points to a short integer 04280000
* - *sqlstate points to a char[06], null-terminated string 04290000
* - *fnName points to a char[18], null-terminated string 04305000
* - *specificName points to a char[129], null-terminated string 04311980
* - *message points to a char[70], null-terminated string 04320000

Assumptions:
* - *specificName points to a char[129], null-terminated string 04320000
* - *message points to a char[70], null-terminated string 04330000

Assumptions:
* - *fnName points to a char[18], null-terminated string 04350000
* - *sqlstate points to a char[06], null-terminated string 04360000
* - *nullFormatOut points to a short integer 04370000
* - *nullFormatIn points to a short integer 04380000
* - *nullDateOut points to a short integer 04390000
* - *nullDateIn points to a short integer 04400000
* - *dateOut points to a char[18], null-terminated string 04410000
* - *formatIn, points to a char[14], null-terminated string 04420000
* - *dateIn points to a char[18], null-terminated string 04430000

Assumptions:
* - *dateIn points to a char[18], null-terminated string 04440000
* - *formatIn points to a char[14], null-terminated string 04450000
* - *dateOut points to a char[18], null-terminated string 04460000

Assumptions:
* - *dateIn points to a char[18], null-terminated string 04470000
* - *formatIn points to a char[14], null-terminated string 04480000
* - *dateOut points to a char[18], null-terminated string 04490000

Assumptions:
* - *dateIn points to a char[18], null-terminated string 04500000
* - *formatIn points to a char[14], null-terminated string 04510000
* - *dateOut points to a char[18], null-terminated string 04520000

Assumptions:
* - *dateIn points to a char[18], null-terminated string 04530000
* - *formatIn points to a char[14], null-terminated string 04540000
* - *dateOut points to a char[18], null-terminated string 04550000

Assumptions:
* - *dateIn points to a char[18], null-terminated string 04560000
* - *formatIn points to a char[14], null-terminated string 04570000
* - *dateOut points to a char[18], null-terminated string 04580000

Assumptions:
* - *dateIn points to a char[18], null-terminated string 04590000
* - *formatIn points to a char[14], null-terminated string 04600000
* - *dateOut points to a char[18], null-terminated string 04610000

Assumptions:
* - *dateIn points to a char[18], null-terminated string 04620000
* - *formatIn points to a char[14], null-terminated string 04630000
* - *dateOut points to a char[18], null-terminated string 04640000

Assumptions:
* - *dateIn points to a char[18], null-terminated string 04650000
* - *formatIn points to a char[14], null-terminated string 04660000
* - *dateOut points to a char[18], null-terminated string 04670000

Assumptions:
* - *dateIn points to a char[18], null-terminated string 04680000
* - *formatIn points to a char[14], null-terminated string 04690000
* - *dateOut points to a char[18], null-terminated string 04700000

Assumptions:
* - *dateIn points to a char[18], null-terminated string 04710000
* - *formatIn points to a char[14], null-terminated string 04720000
* - *dateOut points to a char[18], null-terminated string 04730000

Assumptions:
* - *dateIn points to a char[18], null-terminated string 04740000
* - *formatIn points to a char[14], null-terminated string 04750000
* - *dateOut points to a char[18], null-terminated string 04760000

Assumptions:
* - *dateIn points to a char[18], null-terminated string 04770000
* - *formatIn points to a char[14], null-terminated string 04780000
* - *dateOut points to a char[18], null-terminated string 04790000

Assumptions:
* - *dateIn points to a char[18], null-terminated string 04800000
* - *formatIn points to a char[14], null-terminated string 04810000
* - *dateOut points to a char[18], null-terminated string 04820000

Assumptions:
* - *dateIn points to a char[18], null-terminated string 04830000
* - *formatIn points to a char[14], null-terminated string 04840000
* - *dateOut points to a char[18], null-terminated string 04850000

Assumptions:
* - *dateIn points to a char[18], null-terminated string 04860000
* - *formatIn points to a char[14], null-terminated string 04870000
* - *dateOut points to a char[18], null-terminated string 04880000

Assumptions:
* - *dateIn points to a char[18], null-terminated string 04890000
* - *formatIn points to a char[14], null-terminated string 04900000

Assumptions:
* - *dateIn points to a char[18], null-terminated string 04910000
* - *formatIn points to a char[14], null-terminated string 04920000
* - *dateOut points to a char[18], null-terminated string 04930000

Assumptions:
* - *dateIn points to a char[18], null-terminated string 04940000
* - *formatIn points to a char[14], null-terminated string 04950000
* - *dateOut points to a char[18], null-terminated string 04960000

Assumptions:
* - *dateIn points to a char[18], null-terminated string 04970000
* - *formatIn points to a char[14], null-terminated string 04980000
* - *dateOut points to a char[18], null-terminated string 04990000

Assumptions:
* - *dateIn points to a char[18], null-terminated string 05000000
* - *formatIn points to a char[14], null-terminated string 05010000
* - *dateOut points to a char[18], null-terminated string 05020000
strncpy( sqlstate,"00000" );
}

/*************************************************************/ 04910000
* If errors occurred, clear the dateOut buffer and set the SQL null* 04920000
* indicator. A diagnostic message and the SQLSTATE have been set *
* where the error was detected.                                    */ 04930000
else
{
    dateOut[0] = NULLCHAR;
    *nullDateOut = -1;
}

return;
} /* end of DSN8DUCD */

int deconDate /* get yr, mo, dy from dateIn */ 05400000
( char *yr,  /* out: year component */ 05410000
  char *mo,  /* out: month component */ 05420000
  char *dy,  /* out: day component */ 05430000
  char *message,  /* out: diagnostic message */ 05440000
  char *sqlstate,  /* out: SQLSTATE */ 05450000
  char *dateIn,  /* in: inputted date string */ 05460000
  char *fmtIn /* in: format of dateIn */ 05470000
)

char        tok2[17]; /*  "   2nd  "       "        */ 05250000
char        tok1[17]; /* Gets 1st date component */ 05240000
char        *token; /* Value from token parser */ 05230000
char        workDateIn[18]; /* work copy of dateIn */ 05220000
short int   ftStatus = OK; /*     "     "  format  "  "  */ 05210000
short int   yrStatus = OK; /* indicates if year    is OK */ 05200000
short int   func_status = OK; /* function status indicator */ 05190000

/************************ Local variables *************************/ 05180000
{ 05170000

/*********************************************************************/ 05160000
* text in *message and returns 0.                                    */ 05150000
* Returns 1 if deconstruction succeeds, otherwise places diagnostic  */ 05140000
* Deconstructs *dateIn into *yr, *mo, and *dy according to *fmtIn.   */ 05130000
/*********************************************************************/ 05120000
)
char        *fmtIn                  /* in: format of dateIn       */ 05110000
char        *dateIn,                /* in: inputted date string   */ 05100000
char        *message,               /* out: diagnostic message    */ 05090000
char        *dy,                    /* out: day component         */ 05080000
(int deconDate                         /* get yr, mo, dy from dateIn */ 05070000
  char *yr,                    /* out: year component        */ 05060000
  char *mo,                    /* out: month component       */ 05050000
  char *dy,                    /* out: day component         */ 05040000
  char *message,               /* out: diagnostic message    */ 05030000
  char *sqlstate,               /* out: SQLSTATE               */ 05020000
  char *dateIn,                 /* in: inputted date string    */ 05010000
  char *fmtIn,                  /* in: format of dateIn        */ 05000000
)

strncpy( workDateIn,dateIn );

strcpy( tok1[17]; /* Gets 1st date component */ 05480000
char tok2[17]; /* " 2nd " */ 05470000
char tok3[17]; /* " 3rd " */ 05460000

strcpy( tok3,token );

strcpy( tok2,token );

strcpy( tok1,token );

strcpy( workDateIn," .-/");

strcpy( tok1,token );

strcpy( tok2,token );

strcpy( tok3,token );

strcpy( sqlstate,"00000" );

strncpy( sqlstate,"00000" );

return;
dyStatus = checkDay( dy,tok1 );
moStatus = checkMonth( mo,tok2,"M/MM" );
yrStatus = checkYear( yr,tok3,"YY" );
}
else if( ( strcmp( fmtIn,"D/M/DD" ) == MATCH )
| ( strcmp( fmtIn,"DD/MM/YY" ) == MATCH ) )
{
  moStatus = checkMonth( mo,tok1,"M/MM" );
dyStatus = checkDay( dy,tok2 );
yrStatus = checkYear( yr,tok3,"YY" );
}
else if( ( strcmp( fmtIn,"D/M/YY" ) == MATCH )
| ( strcmp( fmtIn,"MM/DD/YYYY" ) == MATCH ) )
{
  dyStatus = checkDay( dy,tok1 );
  moStatus = checkMonth( mo,tok2,"M/MM" );
yrStatus = checkYear( yr,tok3,"YYYY" );
}
else /* date-in format is invalid or unknown */

ftStatus = NOT_OK;

/*******************************************************************/
* set up error handling                                       *
*******************************************************************/

func_status = NOT_OK;
strcpy( message,"DSN8DUCD Error: ");

if( ftStatus != OK )
{
  dyStatus = checkDay( dy,tok1 );
  moStatus = checkMonth( mo,tok2,"M/MM" );
yrStatus = checkYear( yr,tok3,"YYYY" );
}

strcpy( message,"Unknown input format specified" );
else if( yrStatus != OK )
    strcpy( message,
            "Value for year "
            "is incorrect or does not "
            "conform to input format ");
else if( moStatus != OK )
    strcpy( message,
            "Value for month "
            "is incorrect or does not "
            "conform to input format ");
else if( dyStatus != OK )
    strcpy( message,
            "Value for day "
            "is incorrect or does not "
            "conform to input format ");

int reconDate( char *dateOut,               /* out: reformatted date str */ 07050000
               char *yr,                    /* in: year component */ 06870000
               char *mo,                    /* in: month component */ 06890000
               char *dy,                    /* in: day component */ 06900000
               char *fmtOut,                /* in: format for dateOut */ 06850000
               char *sqlstate,              /* out: SQLSTATE */ 06860000
               char *message,               /* out: diagnostic message */ 06880000
               short int func_status = OK;       /* function status indicator */ 06890000
)  /* end reconDate */ 06820000
{
    strcpy( sqlstate, "00000" ); 06780000
    func_status = OK; 06770000

    // Use fmtOut to reformat date from year, month, and day tokens 06750000
          // Converts *yr, *mo, and *dy into *dateOut according to *fmtOut. 06730000
          // Returns 1 if reconstruction succeeds, otherwise places diagnostic 06710000
          // text in *message and returns 0. 06700000
          // Reconstructs *yr, *mo, and *dy into *dateOut according to *fmtOut. 06690000
          // if no error detected, clear message and sqlstate and return OK 06680000
    if( strncmp( fmtOut,"D MONTH YY" ) == MATCH ) 06670000
    {
        remove0prefix( dy ); 06710000
        nameMonth( mo ); 06720000
        removeCentury( yr ); 06730000
        buildDate( dateOut, dy, mo, yr, " "); 06740000
        return( func_status ); 06750000
    } 06760000
    else if( strncmp( fmtOut,"DD MONTH YYYY" ) == MATCH ) 06660000
    {
        add0prefix( dy ); 06770000
        nameMonth( mo ); 06780000
        removeCentury( yr ); 06790000
        buildDate( dateOut, dy, mo, yr, " "); 06800000
        return( func_status ); 06810000
    } 06820000
    else if( strncmp( fmtOut,"D MONTH YY" ) == MATCH ) 06630000
    {
        add0prefix( dy ); 06730000
        nameMonth( mo ); 06740000
        addCentury( yr ); 06750000
        buildDate( dateOut, dy, mo, yr, " "); 06760000
        return( func_status ); 06770000
    } 06780000
    else if( strncmp( fmtOut,"M.YY" ) == MATCH ) 06620000
    {
        add0prefix( dy ); 06720000
        nameMonth( mo ); 06730000
        addCentury( yr ); 06740000
        buildDate( dateOut, dy, mo, yr, " "); 06750000
        return( func_status ); 06760000
    } 06770000
    else if( strncmp( fmtOut,"D.M.YY" ) == MATCH ) 06610000
    {
        add0prefix( dy ); 06710000
        nameMonth( mo ); 06720000
        addCentury( yr ); 06730000
        buildDate( dateOut, dy, mo, yr, " "); 06740000
        return( func_status ); 06750000
    } 06760000
    else 06770000
    {
        strcpy( message, NULLCHAR); 06650000
        func_status = OK; 06660000
        strcpy( sqlstate, "00000" ); 06670000
    } 06680000

    return( func_status ); 06690000
} 06700000
remove0prefix( mo );               /* strip leading 0 if mon < 10 */ 07370000
removeCentury( yr );               /* strip century from year */ 07380000
buildDate( dateOut, dy, mo, yr, " "); 07390000
}
else if( strcmp( fmtOut,"DD.MM.YY" ) == MATCH )
{
    add0prefix( dy );               /* add leading 0 if day < 10 */ 07400000
    add0prefix( mo );               /* add leading 0 if mon < 10 */ 07410000
    removeCentury( yr );             /* strip century from year */ 07420000
    buildDate( dateOut, dy, mo, yr, "); 07430000
}
else if( strcmp( fmtOut,"D-M-YY" ) == MATCH )
{
    remove0prefix( dy );             /* strip leading 0 if day < 10 */ 07440000
    remove0prefix( mo );             /* strip leading 0 if mon < 10 */ 07450000
    removeCentury( yr );             /* strip century from year */ 07460000
    buildDate( dateOut, dy, mo, yr, "); 07470000
}
else if( strcmp( fmtOut,"D-M-MM" ) == MATCH )
{
    add0prefix( dy );               /* add leading 0 if day < 10 */ 07480000
    add0prefix( mo );               /* add leading 0 if mon < 10 */ 07490000
    removeCentury( yr );             /* strip century from year */ 07500000
    buildDate( dateOut, dy, mo, yr, "); 07510000
}
else if( strcmp( fmtOut,"D/YY" ) == MATCH )
{
    remove0prefix( dy );             /* strip leading 0 if day < 10 */ 07520000
    remove0prefix( mo );             /* strip leading 0 if mon < 10 */ 07530000
    removeCentury( yr );             /* strip century from year */ 07540000
    buildDate( dateOut, dy, mo, yr, "); 07550000
}
else if( strcmp( fmtOut,"M/YY" ) == MATCH )
{
    remove0prefix( dy );             /* strip leading 0 if day < 10 */ 07560000
    remove0prefix( mo );             /* strip leading 0 if mon < 10 */ 07570000
    removeCentury( yr );             /* strip century from year */ 07580000
    buildDate( dateOut, dy, mo, yr, "); 07590000
}
else if( strcmp( fmtOut,"YY" ) == MATCH )
{
    remove0prefix( dy );             /* strip leading 0 if day < 10 */ 07600000
    remove0prefix( mo );             /* strip leading 0 if mon < 10 */ 07610000
    removeCentury( yr );             /* strip century from year */ 07620000
    buildDate( dateOut, dy, mo, yr, "); 07630000
}
else if( strcmp( fmtOut,"M" ) == MATCH )
{
    remove0prefix( dy );             /* strip leading 0 if day < 10 */ 07640000
    remove0prefix( mo );             /* strip leading 0 if mon < 10 */ 07650000
    removeCentury( yr );             /* strip century from year */ 07660000
    buildDate( dateOut, dy, mo, yr, "); 07670000
}
else if( strcmp( fmtOut,"MM" ) == MATCH )
{
    remove0prefix( dy );             /* strip leading 0 if day < 10 */ 07680000
    remove0prefix( mo );             /* strip leading 0 if mon < 10 */ 07690000
    removeCentury( yr );             /* strip century from year */ 07700000
    buildDate( dateOut, dy, mo, yr, "); 07710000
}
else if( strcmp( fmtOut,"YYYY" ) == MATCH )
{
    remove0prefix( dy );             /* strip leading 0 if day < 10 */ 07720000
    remove0prefix( mo );             /* strip leading 0 if mon < 10 */ 07730000
    removeCentury( yr );             /* strip century from year */ 07740000
    buildDate( dateOut, dy, mo, yr, "); 07750000
}
else if( strcmp( fmtOut,"MMM" ) == MATCH )
{
    remove0prefix( dy );             /* strip leading 0 if day < 10 */ 07760000
    remove0prefix( mo );             /* strip leading 0 if mon < 10 */ 07770000
    removeCentury( yr );             /* strip century from year */ 07780000
    buildDate( dateOut, dy, mo, yr, "); 07790000
}
else if( strcmp( fmtOut,"MMMM" ) == MATCH )
{
    remove0prefix( dy );             /* strip leading 0 if day < 10 */ 07800000
    remove0prefix( mo );             /* strip leading 0 if mon < 10 */ 07810000
    removeCentury( yr );             /* strip century from year */ 07820000
    buildDate( dateOut, dy, mo, yr, "); 07830000
}
else if( strcmp( fmtOut,"YYYYYY" ) == MATCH )
{
    remove0prefix( dy );             /* strip leading 0 if day < 10 */ 07840000
    remove0prefix( mo );             /* strip leading 0 if mon < 10 */ 07850000
    removeCentury( yr );             /* strip century from year */ 07860000
    buildDate( dateOut, dy, mo, yr, "); 07870000
}
else if( strcmp( fmtOut,"YYYYY" ) == MATCH )
{
    remove0prefix( dy );             /* strip leading 0 if day < 10 */ 07880000
    remove0prefix( mo );             /* strip leading 0 if mon < 10 */ 07890000
    removeCentury( yr );             /* strip century from year */ 07900000
    buildDate( dateOut, dy, mo, yr, "); 07910000
}
else if( strcmp( fmtOut,"YYYYMM" ) == MATCH )
{
    remove0prefix( dy );             /* strip leading 0 if day < 10 */ 07920000
    remove0prefix( mo );             /* strip leading 0 if mon < 10 */ 07930000
    removeCentury( yr );             /* strip century from year */ 07940000
    buildDate( dateOut, dy, mo, yr, "); 07950000
}
else if( strcmp( fmtOut,"YYYYMMYY" ) == MATCH )
{
    remove0prefix( dy );             /* strip leading 0 if day < 10 */ 07960000
    remove0prefix( mo );             /* strip leading 0 if mon < 10 */ 07970000
    removeCentury( yr );             /* strip century from year */ 07980000
    buildDate( dateOut, dy, mo, yr, "); 07990000
}
else if( strcmp( fmtOut,"YYYYMMMM" ) == MATCH )
{
    remove0prefix( dy );             /* strip leading 0 if day < 10 */ 08000000
    remove0prefix( mo );             /* strip leading 0 if mon < 10 */ 08010000
    removeCentury( yr );             /* strip century from year */ 08020000
    buildDate( dateOut, dy, mo, yr, "); 08030000
}
else if( strcmp( fmtOut,"DD/YY" ) == MATCH )
{
    remove0prefix( dy );             /* strip leading 0 if day < 10 */ 08040000
    remove0prefix( mo );             /* strip leading 0 if mon < 10 */ 08050000
    removeCentury( yr );             /* strip century from year */ 08060000
    buildDate( dateOut, dy, mo, yr, "); 08070000
}
else if( strcmp( fmtOut,"DD/MM/YY" ) == MATCH )
{
    remove0prefix( dy );             /* strip leading 0 if day < 10 */ 08080000
    remove0prefix( mo );             /* strip leading 0 if mon < 10 */ 08090000
    removeCentury( yr );             /* strip century from year */ 08100000
    buildDate( dateOut, dy, mo, yr, "); 08110000
}
else if( strcmp( fmtOut,"DD-MM/YY" ) == MATCH )
{
    remove0prefix( dy );             /* strip leading 0 if day < 10 */ 08120000
    remove0prefix( mo );             /* strip leading 0 if mon < 10 */ 08130000
    removeCentury( yr );             /* strip century from year */ 08140000
    buildDate( dateOut, dy, mo, yr, "); 08150000
}
else if( strcmp( fmtOut,"DD.MM.YY" ) == MATCH )
{
    remove0prefix( dy );             /* strip leading 0 if day < 10 */ 08160000
    remove0prefix( mo );             /* strip leading 0 if mon < 10 */ 08170000
    removeCentury( yr );             /* strip century from year */ 08180000
}
{  remove0prefix( mo );  /* strip leading 0 if mon < 10*/ 08200000
  remove0prefix( dy );  /* strip leading 0 if day < 10*/ 08220000
  buildDate( dateOut, mo, dy, yr, "/" );
}
else if( strcmp( fmtOut,"MM/DD/YY" ) == MATCH )
{
  add0prefix( mo );  /* add leading 0 if mon < 10 */ 08279000
  add0prefix( dy );  /* add leading 0 if day < 10 */ 08280000
  addCentury( yr );  /* ensure year has century */ 08290000
  buildDate( dateOut, mo, dy, yr, "/" );
}
else if( strcmp( fmtOut,"M/D/YYYY" ) == MATCH )
{
  remove0prefix( mo );  /* strip leading 0 if mon < 10*/ 08334000
  remove0prefix( dy );  /* strip leading 0 if day < 10*/ 08335000
  addCentury( yr );  /* ensure year has century */ 08368000
  buildDate( dateOut, mo, dy, yr, "/" );
}
else if( strcmp( fmtOut,"MM/DD/YYYY" ) == MATCH )
{
  removeCentury( yr );  /* strip century from year */ 08400000
  removePrefix( yr );  /* strip leading 0 if yr < 10*/ 08401000
  addCentury( yr );  /* ensure year has century */ 08439000
  buildDate( dateOut, yr, mo, dy, "/" );
}
else if( strcmp( fmtOut,"YY/MM/DD" ) == MATCH )
{
  removeCentury( yr );  /* strip century from year */ 08550000
  removePrefix( yr );  /* strip leading 0 if yr < 10*/ 08556000
  addCentury( yr );  /* ensure year has century */ 08579000
  buildDate( dateOut, yr, mo, dy, "/" );
}
else if( strcmp( fmtOut,"YY.M.D" ) == MATCH )
{
  removeCentury( yr );  /* strip century from year */ 08600000
  removePrefix( yr );  /* strip leading 0 if yr < 10*/ 08603000
  buildDate( dateOut, yr, mo, dy, "." );
}
else if( strcmp( fmtOut,"YY.MM.DD" ) == MATCH )
{
  removeCentury( yr );  /* strip century from year */ 08760000
  removePrefix( yr );  /* strip leading 0 if yr < 10*/ 08769000
  buildDate( dateOut, yr, mo, dy, "/" );
}
else if( strcmp( fmtOut,"YYY/M/DD" ) == MATCH )
{
  addCentury( yr );  /* ensure year has century */ 08790000
  removePrefix( yr );  /* strip leading 0 if yr < 10*/ 08790000
  buildDate( dateOut, yr, mo, dy, "." );
}
else if( strcmp( fmtOut,"YYY/MM/DD" ) == MATCH )
{
  addCentury( yr );  /* ensure year has century */ 08830000
  removePrefix( yr );  /* strip leading 0 if yr < 10*/ 08890000
  buildDate( dateOut, yr, mo, dy, "." );
}
else if( strcmp( fmtOut,"YYY.M/DD" ) == MATCH )
{
  addCentury( yr );  /* ensure year has century */ 08970000
  removePrefix( yr );  /* strip leading 0 if yr < 10*/ 08990000
  buildDate( dateOut, yr, mo, dy, "." );
}
```
else if( strcmp( fmtOut,"YYYY-M-D" ) == MATCH )
{
    strcat( sqlstate, "00000" );
    strcpy( message, "Unknown output format specified" );
    return( func_status );
}
else if( strcmp( fmtOut,"YYYY-MM-DD" ) == MATCH )
{
    strcat( sqlstate, "38603" );
    if( func_status != OK )
    {
        buildDate( dateOut, yr, mo, dy, "-" );
        remove0prefix( dy );
        romanMonth( mo );
        add0prefix( mo );
        addCentury( yr );
    }
    else /* date-in format is invalid or unknown */
    {
        strncpy( sqlstate, "00000" );
        strcpy( message, "Unknown output format specified" );
        return( func_status );
    }
    */
    func_status = NOT_OK;
    if( func_status != OK )
    {
        strcpy( sqlstate, "00000" );
        strcpy( message, "Unknown output format specified" );
    }
    else
    {
        *message = NULLCHAR;
        strcpy( sqlstate, "00000" );
    }
    return( func_status );
}/* end reconDate */

void buildDate /* build date from parts */
    ( char *dtOut,
    * out: date * /
    char *d1, /* in: year, month, or day */
    char *d2, /* in: year, month, or day */
    char *d3, /* in: year, month, or day */
    char *delim /* in: delimiter */
    )
    
    /**************************************************************************/
    * Forms a date by concatenating d1, del2, del2, and del3. *
    /**************************************************************************/
    {
        strcat( dtOut, d1 );
        strcat( dtOut, delim );
        strcat( dtOut, d2 );
        strcat( dtOut, delim );
        strcat( dtOut, d3 );
    }/* end buildDate */

int nameMonth /* converts month num to name */
```
Chapter 12. Sample data and applications supplied with Db2 for z/OS 1205

```c
(char *monthIn  /* in/out: month to convert */
)

/************************ Local variables *************************/
short int i;    /* # loop control */
short int func_status = OK;  /* # function status indicator */

/******************************************************************************
* Converts *monthIn from a name string to a name.  Returns 1 if conversion succeeds, otherwise returns 0.
* Strips leading zero (if any) from monthIn.
* Look up *monthIn in the month number strings array.
* If found assign month name else set function error indicator.
* Conversion succeeds, otherwise returns 0.
*******************************************************************************/

if( i < 12 )
    strcpy( monthIn,monthNames[i] );
else
    func_status = NOT_OK;

return( func_status );

} /* end nameMonth */

int unnameMonth /* converts month name to num */
(char *monthIn  /* in/out: month to convert */
)

/************************ Local variables *************************/
short int i;    /* # loop control */
short int func_status = OK;  /* # function status indicator */

/******************************************************************************
* Make 1st char of month name upper case and the rest lower case.
* Look up *monthIn in the month names array.
* If found assign month name else set function error indicator.
* Conversion succeeds, otherwise returns 0.
*******************************************************************************/

monthIn[0] = toupper(monthIn[0]);
for( i=1;i<strlen(monthIn);i++ )
    monthIn[i] = tolower(monthIn[i]);

/******************************************************************************
* Look up *monthIn in the month names array.
* If found assign month name else set function error indicator.
* Conversion succeeds, otherwise returns 0.
*******************************************************************************/

for( i=0; i<12 && strcmp( monthIn,monthNames[i] ) != MATCH; i++ );
if( i < 12 )
    strcpy( monthIn,monthNames[i] );
else
    func_status = NOT_OK;

return( func_status );

} /* end unnameMonth */

int romanMonth /* converts month# to roman# */
(char *monthIn  /* in/out: month to convert */
)

/************************ Local variables *************************/

/******************************************************************************
* Converts *monthIn from a number string to a roman numeral.  Returns 1 if conversion succeeds, otherwise returns 0.
*******************************************************************************/

return( NOT_OK );
```

Chapter 12. Sample data and applications supplied with Db2 for z/OS 1205
short int  i;  /* loop control */ 10655000
short int  func_status = OK; /* function status indicator */ 10660000

/************************ Local variables *************************/ 10665000
{ 10670000

/*********************************************************************/ 10675000

* If criteria not satisfied, sets *yearOut to null and returns 0. * 10680000
* If criteria satisfied, copies *yearIn to *yearOut and returns 1. * 10685000
* - 4 numeric characters if *style is YYYY.                          * 10690000
* - 2 numeric characters if *style is YY; or                         * 10695000
* Verifies that *yearIn is either of the following:                  * 10699000

/********************************************************************* 10704000
*/

char        *style                  /* in: style of yearIn        */ 10710000
char        *yearIn,                /* in: 2- or 4-digit year     */ 10715000
(char        *yearOut,               /* out: 4-digit yr, validated */ 10720000
int checkYear                         /* verify/standardize yearIn  */ 10725000
10730000

} /* end unromanMonth */                                               10735000

int unromanMonth /* converts roman# to month# */ 10740000
( char        *monthIn                /* in/out: month to convert   */ 10745000
int unromanMonth                      /* converts roman# to month#  */ 10750000
10755000
10760000
10765000
10770000
10775000
10780000
10785000
10790000
10795000
10800000
10805000
10810000
10815000
10820000
10825000
10830000
10835000
10840000
10845000
10850000
10855000
10860000
10865000
10870000
10875000
10880000
10885000
10890000
10895000
10900000
10905000
10910000
10915000
10920000
10925000
10930000
10935000
10940000
10945000
10950000
10955000
10960000
10965000
10970000
10975000
10980000
10985000
10990000
10995000
11000000
11005000
11010000
11015000
11020000
11025000
11030000
11035000
11040000
11045000
11050000
11055000
11060000
11065000
11070000
11075000
11080000
11085000
11090000
11095000
11100000
11105000
11110000
11115000
11120000
11125000
11130000
11135000
11140000
11145000
11150000
11155000
11160000
11165000
11170000
11175000
11180000
11185000
11190000
11195000
11200000
11205000
11210000
11215000
11220000
11225000
11230000
11235000
11240000
11245000
11250000
11255000
11260000
11265000
11270000
11275000
11280000
11285000
11290000
11295000
11300000
11305000
11310000
11315000
11320000
11325000
11330000
11335000
11340000
11345000
11350000
11355000
11360000
11365000
11370000
11375000
11380000
11385000
11390000
11395000
11400000
11405000
11410000
11415000
11420000
11425000
11430000
11435000
11440000
11445000
11450000
11455000
11460000
11465000
11470000
11475000
11480000
11485000
11490000
11495000

short int   func_status = OK;       /* function status indicator */

/************************ Local variables *************************/

short int   i;                      /* loop control               */
short int   func_status = OK;       /* function status indicator */

/***************************************************************************/
* Verifies that *monthIn is one of the following:                        *
* - A valid month name, January - December, if *style is MONTH; or      *
* - A valid roman numeral, I - XII, if *style is XX; or                *
* - 1 or 2 numeric characters between 1 and 12 if *style is M or MM.  *
* - If criteria satisfied, copies *monthIn to *monthOut and returns 1. *
* - If *monthIn is a month name or a roman numeral, it will have      *
*   been standardized to the form 1-12.                               *
* - A valid roman numeral, I - XII, if *style is XX; or                *
* - A valid month name, January - December, if *style is MONTH; or    *
* - 1 or 2 numeric characters between 1 and 12 if *style is M or MM.  *
* If a check failed, sets *monthOut to null and returns 0.            *
* If a check failed, sets *monthOut to null and returns 0.            *
* If all checks satisfied, copy *monthIn to *monthOut and return 1.   *
* If all checks satisfied, copy *monthIn to *monthOut and return 1.   *
* If criteria not satisfied, sets *monthOut to null and returns 0.    *
* If criteria not satisfied, sets *monthOut to null and returns 0.    *
* If criteria not satisfied, sets *monthOut to null and returns 0.    *
* If criteria not satisfied, sets *monthOut to null and returns 0.    *
* If input format is YYYY, verify that *yearIn has 4 bytes            *
* If input format is YYYY, verify that *yearIn has 4 bytes            *
* If input format is YY, verify that *yearIn has 2 bytes              *
* If input format is MM, verify that *yearIn has 2 bytes              *
* If input format is YY, verify that *yearIn has 2 bytes              *
* If input format is MM, verify that *yearIn has 2 bytes              *
* If input format is YYYY, verify that *yearIn has 4 bytes            *
* If input format is YYYY, verify that *yearIn has 4 bytes            *
* If input format is YY, verify that *yearIn has 2 bytes              *
* If input format is YY, verify that *yearIn has 2 bytes              *
***************************************************************************/

int checkMonth                        /* verify/standardize monthIn */
( char        *monthOut,              /* out: month#, validated   */ 11820000
  char        *monthIn,               /* in: month name, #, roman# */ 11830000
  char        *style                  /* in: style of monthIn      */ 11840000
)

/***************************************************************************/
* Verifies that *monthIn is one of the following:                        *
* - A valid month name, January - December, if *style is MONTH; or      *
* - A valid roman numeral, I - XII, if *style is XX; or                *
* - 1 or 2 numeric characters between 1 and 12 if *style is M or MM.  *
* - If criteria satisfied, copies *monthIn to *monthOut and returns 1. *
* - If *monthIn is a month name or a roman numeral, it will have      *
*   been standardized to the form 1-12.                               *
* - If criteria not satisfied, sets *monthOut to null and returns 0.    *
* * - If criteria not satisfied, sets *monthOut to null and returns 0.    *
* * - If criteria not satisfied, sets *monthOut to null and returns 0.    *
* * - If criteria not satisfied, sets *monthOut to null and returns 0.    *
* * - If criteria not satisfied, sets *monthOut to null and returns 0.    *
* * - If criteria not satisfied, sets *monthOut to null and returns 0.    *
* * - If criteria not satisfied, sets *monthOut to null and returns 0.    *
* * - If criteria not satisfied, sets *monthOut to null and returns 0.    *
* * - If criteria not satisfied, sets *monthOut to null and returns 0.    *
* * - If criteria not satisfied, sets *monthOut to null and returns 0.    *
* * - If criteria not satisfied, sets *monthOut to null and returns 0.    *
* * - If criteria not satisfied, sets *monthOut to null and returns 0.    *
* * - If criteria not satisfied, sets *monthOut to null and returns 0.    *
***************************************************************************/

Chapter 12. Sample data and applications supplied with Db2 for z/OS 1207
**Strips the leading zero from *string, if it has one.**

```c
int checkDay
    /**************************************************************************
    * Verifies that *dayIn is either 1 or 2 numeric characters.  *
    * If criteria satisfied, copies *dayIn to *dayOut and returns 1.  *
    * If criteria not satisfied, set *dayOut to null and returns 0.  *
    ************************************************************************/
    
    short int   func_status = OK;       /* function status indicator  */
    char        *str3                   /* in/out: string to prefix   */
    char        *dayOut,                /* out: day, validated        */
    char        *dayIn                  /* in: day number             */
    int checkMonth                         /* verify/standardize dayIn   */
    short int   func_status = NOT_OK;    /* function status indicator  */
    short int   dayIn_len               /* length of *dayIn           */
    short int   i;                      /* loop control               */
    char        *str3[2] = NULLCHAR;     /* And terminate it           */
    str3[0] = *char0;                    /* Prefix it with "0"         */
    str3[1] = str3[0];                   /* Right-shift *str3 1 byte   */
    str3[2] = NULLCHAR;                  /* And terminate it           */
    if( func_status == OK )
        strcpy( dayOut, dayIn );
    else
        *dayOut = NULLCHAR;
    return( func_status );
} /* end checkDay */
```

```c
void add0prefix
    /**************************************************************************
    * Prefixes *str3 with a leading 0 if it is only 1 byte long.  *
    ************************************************************************/
    
    short int   func_status = OK;       /* function status indicator  */
    char        *str3[2] = NULLCHAR;     /* And terminate it           */
    str3[0] = *char0;                    /* Prefix it with "0"         */
    str3[1] = str3[0];                   /* Right-shift *str3 1 byte   */
    str3[2] = NULLCHAR;                  /* And terminate it           */
    if( strlen( str3 ) == 1 )
        {str3[1] = str3[0]; /* Right-shift *str3 1 byte */
            str[0] = *char0; /* Prefix it with "0" */
            str[2] = NULLCHAR; /* And terminate it */
        }
    } /* end add0prefix */
```

```c
void remove0prefix
    /**************************************************************************
    * Strips the leading zero from *string, if it has one.  *
    ************************************************************************/
    
    short int   func_status = OK;       /* function status indicator  */
    char        *string[1] = NULLCHAR;  /* And terminate it           */
    char        *string[1] = string[0]; /* Left-shift *string */
    string[0] = string[1];            /* And terminate it */
    if( strncmp( string,"0",1 ) == MATCH )
        {string[0] = string[1];/* Left-shift *string*/
            string[1] = NULLCHAR; /* And terminate it */
        }
    } /* end remove0prefix */
```

```c
void addCentury
    /**************************************************************************
    * adds century to yearIn  *
    ************************************************************************/
    
    short int   func_status = OK;       /* function status indicator  */
    char        *string[2] = NULLCHAR;  /* And terminate it           */
    if( i < dayIn_len | dayIn_len < 1 | dayIn_len > 2 )
        for( i=0; ( i<dayIn_len ) && ( isdigit(dayIn[i]) ); i++ );
    if( func_status == NOT_OK )
        *dayOut = NULLCHAR;
    return( func_status );
} /* end checkMonth */
```
( char *yearIn /* in/out: year */
)
/*********************************************************************/
* Prefixes *yearIn with the current century (according to the system *
* date) if *yearIn is 2 bytes long.                                 *
*********************************************************************/

{                                                                      }
/*********************************************************************
* Prefixes *yearIn with the current century (according to the system *
* date) if *yearIn is 2 bytes long.                                  *
*********************************************************************/

if( strlen( yearIn ) == 2 )                                          
{                                                                  
    t = time(NULL);                 /* Get calendar time from sys */  
    timeptr = localtime(&t);        /* Convert to local time      */  
    strftime( centyear,             /* Format current century year*/  
             sizeof(centyear)-1,   /* ..sized for receiving field*/  
             "%Y",                 /* ..as century year          */  
             timeptr );            /* ..from current local time  */  
    } /* end addCentury */                                                 

void removeCentury                    /* strip century from yearIn  */  
( char *yearIn /* in/out: year */
)
/*********************************************************************
* Strips the century portion from *yearIn if it consists of 4 bytes.  *
*********************************************************************/

{                                                                      
    /* Return the 1st 2 bytes... */  
    yearIn[0] = yearIn[2];          /* Shift non-century portion  */  
    yearIn[1] = yearIn[3];          /* of *yearIn to 1st 2 bytes  */  
    yearIn[2] = NULLCHAR;           /* and terminate string       */  
} /* end removeCentury */                                              

Related reference
“Sample applications in TSO” on page 1027
A set of Db2 sample applications run in the TSO environment.

DSN8DUCT
Converting a given time from one to another of these 8 formats.
where:

H: Suppress leading zero if the hour is less than 10
HH: Retain leading zero if the hour is less than 10
M: Suppress leading zero if the minute is less than 10
MM: Retain leading zero if the minute is less than 10
AM/PM: Return time in 12-hour clock format, else 24-hour

Example invocation:
EXEC SQL SET :then = ALTTIME("01:34:59 PM", "HH:MM:SS AM/PM", "H.MM");

Example return:

"13.34"

Notes:

* Dependencies: Requires IBM C/C++ for OS/390 V1R3 or higher
* Restrictions:

Module type: C program
Processor: IBM C/C++ for OS/390 V1R3 or higher
Module size: See linkedit output
Attributes: Re-entrant and re-usable

Entry Point: DSNODUCT
Purpose: See Function
Linkage: DB2SQL
Invoked via SQL UDF call

Input: Parameters explicitly passed to this function:

- *timeIn : pointer to a char[12], null-terminated string having a time in the
  format indicated by *formatIn.
- *formatIn : pointer to a char[15], null-terminated string having the format of
  time found in *timeIn (see "Function", above, for valid formats).
- *formatOut : pointer to a char[15], null-terminated string having the format to
  which the time found in *timeIn is to be converted. See "Function", above, for valid formats.
- *niTimeIn : pointer to a short integer having the null indicator variable for
  *timeIn.
- *niFormatIn : pointer to a short integer having the null indicator variable for
  *formatIn.
- *niFormatOut : pointer to a short integer having the null indicator variable for
  *formatOut.
- *fnName : pointer to a char[138], null-terminated string having the UDF family
  name of this function.
- *specificName : pointer to a char[129], null-terminated string having the UDF specific
  name of this function.

Output: Parameters explicitly passed by this function:

- *timeOut : pointer to a char[15], null-terminated string to receive the reformatted time.
- *niTimeOut : pointer to a short integer to receive the null indicator variable
  for *timeOut.
- *sqlstate : pointer to a char[06], null-terminated string to receive the SQLSTATE.
- *message : pointer to a char[70], null-terminated string to receive a diagnostic
  message if one is generated by this function.

Normal Exit: Return Code: SQLSTATE = 00000
- Message: none

Error Exit: Return Code: SQLSTATE = 38601
- Message: DSNODUCT Error: No input time entered
- Message: DSNODUCT Error: No input format entered
- Message: DSNODUCT Error: No output format entered
Chapter 12. Sample data and applications supplied with Db2 for z/OS 1211

* Return Code: SQLSTATE = 38602
  * Message: DSN8DUCT Error: Unknown input format specified
  * Message: DSN8DUCT Error: Inputted time must indicate either AM or PM
  * Message: DSN8DUCT Error: Hour not in expected range of 1-12
  * Message: DSN8DUCT Error: Minute must be 2 numerics between 00 and 59
  * Message: DSN8DUCT Error: Second must be 2 numerics between 00 and 59
  * Message: DSN8DUCT Error: Second must be 2 numerics between 00 and 59
  * Message: DSN8DUCT Error: Unknown output format specified
  * Return Code: DSN8DUCT = 38603
  * Message: DSN8DUCT Error: Unknown input format specified

  **Pseudocode:**
  DSN8DUCT:
  - Control blocks: None
  - Data areas: None
  - Message: DSN8DUCT Error: Unknown output format
    Return Code: SQLSTATE = 38603
  - Message: DSN8DUCT Error: Unknown input format
    Return Code: SQLSTATE = 38602
  - Issue sqlstate 38601 and a diagnostic message if no input time was provided.
  - Issue sqlstate 38601 and a diagnostic message if no input format was provided.
  - Issue sqlstate 38601 and a diagnostic message if no output format was provided.
  - Call decontime to deconstruct the input time into hour, minute, and, if either, second and AM/PM indicator, according to the input format.
  - Call recontime to create an output time from the hour, minute, and, if either, second and AM/PM indicator, according to the output format.
  - If no errors, unset null indicators, and return SQLSTATE 00000. Else set null indicator and return null time out.

  **decontime**:
  - Parse hour, minute, and, if either, second and AM/PM indicator from the input time by breaking on delimiters ( : and .).
  - Use the input format to determine sequence of time components. If format invalid, issue SQLSTATE 38602 and a diag. message.
  - Call checkHour to validate the hour component and to standardize it (if required) from a 12-hour clock to a 24-hour clock.
  - Call checkMinute to validate the minute component.
  - If not valid minute, issue SQLSTATE 38602 and a diag. message.
  - Call checkSecond to validate the second component.
  - If not valid second, issue SQLSTATE 38602 and a diag. message.
  - If applicable, call checkAMPMIndicator to validate the AM/PM indicator.
  - If not valid indicator, issue SQLSTATE 38602 and a diag. message.

  **recontime**:
  - Use the output format to edit the time components.
  - Call set12HrClock to convert the hours component from 24-hour clock format, as appropriate.
  - Call remove0prefix to strip the leading 0 from the hour component, if appropriate.
  - If output format is invalid, issue SQLSTATE 38603 and a diagnostic message.
  - Call buildTime to create the output time from the edited time components.

  **buildTime**:
  - Generate the time out by concatenating the time components (hour, minute, and, optionally, second and/or AM/PM indicator) with intervening delimiters ( : or .).
  - Verify that the hour component of the input time is: 01890000.
* - in the range 01 - 12 if the input format carries an AM/PM indicator
* - call set24HrClock to standardize the hour to a 24-hour clock format.
* - in the range 00 - 23 if the input format does not carry an AM/PM indicator
* - If not valid, set error flag and return null value for hour
* End checkHour
* 
* checkMinute
* - Verify the minute component is 2 digits ranging from 00 - 59.
* - If not valid, set error flag and return null value for minute
* End checkMinute
* 
* checkSecond
* - Verify the second component is 2 digits ranging from 00 - 59.
* - If not valid, set error flag and return null value for second
* End checkSecond
* 
* checkAMPIndicator
* - Verify the AM/PM indicator is either "AM" or "PM"
* - If not valid, set error flag and return null value for AM/PM indicator
* End checkAMPIndicator
* 
* set12HrClock
* - Convert a 12-hour clock hour to a 12-hour clock hour
* End set12HrClock
* 
* set24HrClock
* - Convert a 12-hour clock hour to a 24-hour clock hour
* End set24HrClock
* 
* add0prefix
* - prepend an hour with a leading 0 if it is less than 10
* End add0prefix
* 
* remove0prefix
* - strip leading zero from an hour if it is less than 10
* End remove0prefix
* 
* - Convert a 12-hour clock hour to a 24-hour clock hour
* 
************************************************************** C library definitions **************************************************************
#include <ctype.h>
#include <time.h>
#include <string.h>
#include <stdio.h>

#pragma linkage(DSN8DUCT,fetchable)
/*
 *   End remove0prefix
 *   - strip leading zero from an hour if it is less than 10
 *   remove0prefix
 *
 */

/**/
short int status = OK; /* DSNB8DUCT run status */

char *nullTimeOut, /* out: indic var for timeOut */
char *nullFormatOut, /* in: indic var for formatOut */
char *nullFormatIn, /* in: indic var for formatIn */
char *nullTimeIn, /* in: indic var for timeIn */
char *timeOut, /* out: reformatted time */
char *message, /* out: diagnostic message */
char *fnName, /* in: family name of function */
char *specificName, /* in: specific name of func */
char *sqlstate, /* out: SQLSTATE */
char *formatOut, /* points to a char[15], null-terminated string */
char *formatIn, /* points to a char[15], null-terminated string */
char *timeIn, /* points to a char[12], null-terminated string */
char *nullFormatOut, /* in: indic var, formatOut */
char *nullFormatIn, /* in: indic var for formatIn */
char *nullTimeOut, /* out: indic var for timeOut */
char *nullTimeIn, /* in: indic var for timeIn */
char *sqlstate, /* points to a char[6], null-terminated string */
char *specificName, /* points to a char[129], null-terminated string */
char *fnName, /* points to a char[138], null-terminated string */
char *message, /* points to a char[78], null-terminated string */

Assumptions:
- *timeOut points to a char[12], null-terminated string
- *nullFormatIn points to a char[15], null-terminated string
- *nullFormatOut points to a short integer
- *nullTimeIn points to a short integer
- *nullTimeOut points to a short integer
- *sqlstate points to a char[6], null-terminated string
- *fnName points to a char[138], null-terminated string
- *specificName points to a char[129], null-terminated string
- *message points to a char[78], null-terminated string

MASTER Local variables

short int i; /* loop control */
char hour[3]; /* gets hour from timeIn */
char minute[3]; /* gets minute from timeIn */
char second[3]; /* gets second from timeIn */
short int status = OK; /* DSNB8DUCT run status */

Verify that an input time, its current format, and its new format have been passed in.

if(?nullTimeIn? || ( strlen( timeIn ) == 0 ) )
{
    status = NOT_OK;
    strcpy( message, "DSNB8DUCT Error: No input time entered" );
    strcpy( sqlstate, "38601" );
}
else if(?nullFormatIn? || ( strlen( formatIn ) == 0 ) )
{
    status = NOT_OK;
    strcpy( message, "DSNB8DUCT Error: No input format entered" );
    strcpy( sqlstate, "38601" );
}
else if(?nullFormatOut? || ( strlen( formatOut ) == 0 ) )
{
    status = NOT_OK;
    strcpy( message, "DSNB8DUCT Error: No output format entered" );
    strcpy( sqlstate, "38601" );
}

Use formatIn to deconstruct timeIn into hour and minute and, if applicable, second.

if( status == OK )
{
    status = deconTime( hour, minute, second, message, sqlstate, timeIn, formatIn );
}

Use formatOut to reconstruct timeOut from ours and minute and, if applicable, second.

if( status == OK )
{
    status = reconTime( timeOut, message, sqlstate, hour, minute, second, formatOut );
}

If conversion was successful, clear the message buffer and sqlstate, and unset the SQL null indicator for timeOut.
if( status == OK )
{
    *nullTimeOut = 0;
    message[0] = NULLCHAR;
    strcpy( sqlstate,"00000" );
}
#endif

int deconTime
(char *hour,   /* out: hour component */ *hourStatus;
char *minute, /* out: minute component */ *minStatus;
char *second, /* out: second component */ *secStatus;
char *message, /* out: diagnostic message */ *msgStatus;
char *sqlstate, /* out: SQLSTATE */
char *timeIn,     /* in: time to deconstruct */
char *formatIn,   /* in: format of timeIn */
char *second,     /* out: second component */
char *minute,     /* out: minute component */
(char *hour,      /* out: hour component */)
int deconTime;

/* diagnostic text in *message and returns NOT_OK. */
*formatIn.  Returns OK if deconstruction succeeds, otherwise places *
* *second.   The deconstruction is done according to the value in    *
* diagnostic text in *message and returns NOT_OK. */
{ 047290000
    char AMPMInd[3];    /* AM/PM indicator */ *ampmStatus;
    char workTimeIn[12]; /* work copy of timeIn */ *workStatus;
    char *token;        /* string ptr for token parser*/
    char tok1[3];       /* holds 1st time component */
    char tok2[3];       /* holds 2nd time component */
    char tok3[3];       /* holds 3rd time component */
    char tok4[3];       /* holds 4th time component */
    short int func_status = OK; /* function status indicator */ *funcStatus;
    short int IntStatus = OK; /* indicates if format is OK */
    short int hrStatus = OK; /* indicates if hour is OK */
    short int minStatus = OK; /* indicates if minute is OK */
    short int secStatus = OK; /* indicates if second is OK */
    short int indStatus = OK; /* indicates if AMPMInd is OK */
    else
    { 044300000
        timeOut[0] = NULLCHAR;
        *nullTimeOut = -1;
    }
    return;
} /* end DSN8DUCT */
```c
minStatus = checkMinute( minute, tok2 );
strcpy( second, "00" );
}
else if( strcmp( formatIn, "HH:MM:SS/AM/PM" ) == MATCH )
{
    indStatus = checkAMPMinidicator( AMPMind, tok4 );
hzStatus = checkHour( hour, tok1, AMPMind );
minStatus = checkMinute( minute, tok2 );
secStatus = checkSecond( second, tok3 );
}
else if( ( strcmp( formatIn, "HH:MM:SS" ) == MATCH )
    || ( strcmp( formatIn, "HH.MM:SS" ) == MATCH )
    || ( strcmp( formatIn, "HH.MM" ) == MATCH ))
{
hzStatus = checkHour( hour, tok1, "" );
minStatus = checkMinute( minute, tok2 );
secStatus = checkSecond( second, tok3 );
}
else if( ( strcmp( formatIn, "H.MM" ) == MATCH )
    || ( strcmp( formatIn, "H.MM*" ) == MATCH ))
{
hzStatus = checkHour( hour, tok1, "" );
minStatus = checkMinute( minute, tok2 );
strcpy( second, "00" );
}
else
    fmtStatus = NOT_OK;

/* set up error handling */
func_status = NOT_OK;
strcpy( message, "DSN8DUCT Error: " );
strcpy( sqlstate, "38602" );

/* if error detected, issue diagnostic message and return NOT_OK */
if( fmtStatus != OK )
    strcat( message, "Unknown input format specified" );
else if( indStatus != OK )
    strcat( message, "Inputted time must indicate either AM or PM" );
else if( hrStatus != OK )
    strcat( message, "Hour not in expected range of 0-23" );
else
    strcat( message, "Hour not in expected range of 1-12" );
else if( minStatus != OK )
    strcat( message, "Minute not in expected range of 00 and 59" );
else if( secStatus != OK )
    strcat( message, "Second not in expected range of 00 and 59" );
else
    strcat( message, "Inputted time must indicate either AM or PM" );

/* if no error detected, clear message and sqlstate and return OK */
else
{
    *message = NULLCHAR;
    func_status = OK;
    strcpy( sqlstate, "00000" );
}
return( func_status );
} /* end deconTime */
```
void buildTime
(char *timeOut,               /* out: reformatted time */ 06610000
char *hour,                  /* in: hour component */ 06630000
char *minute,                /* in: minute component */ 06640000
char *second,                /* in: second component */ 06650000
char *delim,                 /* in: delimiter */ 06660000
char *AMPMind[3];             /* AM/PM indicator */ 06670000

char AMPMind[3];             /* AM/PM indicator */ 06690000
short int func_status = OK;  /* function status indicator */ 06700000

*)

{ 06020000
/* Use formatOut to reformat time from hour, minute, second */ 06080000
if( strcmp( formatOut,"H:MM AM/PM" ) == MATCH ) 06090000
{ 06100000
set24HrClock( hour,AMPMind ); 06110000
remove0prefix( hour ); 06120000
buildTime( timeOut, hour, minute, " ", ":", AMPMind ); 06130000
}
else if( strcmp( formatOut,"HH:MM AM/PM" ) == MATCH ) 06150000
{ 06160000
set24HrClock( hour,AMPMind ); 06170000
buildTime( timeOut, hour, minute, " ", ":", AMPMind ); 06180000
}
else if( strcmp( formatOut,"HH:MM:SS AM/PM" ) == MATCH ) 06190000
{ 06200000
remove0prefix( hour ); 06210000
buildTime( timeOut, hour, minute, second, ":", AMPMind ); 06220000
}
else if( strcmp( formatOut,"HH:MM:SS" ) == MATCH ) 06230000
{ 06240000
buildTime( timeOut, hour, minute, second, ":", ":", AMPMind ); 06250000
}
else if( strcmp( formatOut,"H.MM AM/PM" ) == MATCH ) 06260000
{ 06270000
set24HrClock( hour,AMPMind ); 06280000
buildTime( timeOut, hour, minute, second, ":", AMPMind ); 06290000
}
else if( strcmp( formatOut,"H.MM AM/PM" ) == MATCH ) 06300000
{ 06310000
remove0prefix( hour ); 06320000
buildTime( timeOut, hour, minute, second, ":", 06330000
}
else if( strcmp( formatOut,"H.MM:SS AM/PM" ) == MATCH ) 06340000
{ 06350000
set24HrClock( hour,AMPMind ); 06360000
buildTime( timeOut, hour, minute, second, ":", ":", AMPMind ); 06370000
}
else if( strcmp( formatOut,"H.MM:SS" ) == MATCH ) 06380000
{ 06390000
remove0prefix( hour ); 06400000
buildTime( timeOut, hour, minute, second, ":", ":", AMPMind ); 06410000
}
else if( strcmp( formatOut,"H:MM:SS AM/PM" ) == MATCH ) 06420000
{ 06430000
set24HrClock( hour,AMPMind ); 06440000
buildTime( timeOut, hour, minute, second, " ", AMPMind ); 06450000
}
else if( strcmp( formatOut,"H:MM AM/PM" ) == MATCH ) 06460000
{ 06470000
remove0prefix( hour ); 06480000
buildTime( timeOut, hour, minute, ":", AMPMind ); 06490000
}
else if( strcmp( formatOut,"H:MM:SS AM/PM" ) == MATCH ) 06500000
{ 06510000
set24HrClock( hour,AMPMind ); 06520000
buildTime( timeOut, hour, minute, second, ":", AMPMind ); 06530000
}
else if( strcmp( formatOut,"H:MM" ) == MATCH ) 06540000
{ 06550000
remove0prefix( hour ); 06560000
buildTime( timeOut, hour, minute, ":", AMPMind ); 06570000
}
else
{ 06580000
func_status = NOT_OK; 06590000
strcpy( message,"Unknown output format specified" ); 06600000
strcpy( sqlstate, "38603" ); 06610000
strcat( message,"DSN8DUCT Error:" ); 06620000
func_status = NOT_OK; 06630000
}
return( func_status ); 06640000
} 06650000

/* end reconTime */ 06660000

Chapter 12. Sample data and applications supplied with Db2 for z/OS 1217
int checkHour
(char        *hourOut,                /* out: hour (24-hour clock) */
char        *hourIn,                /* in: hour (12- or 24-hr clk)*/
char        *AMPMind                /* in: AM/PM indicator        */
)
{/*******************************************************************/
* Verifies that *hourIn meets one of these criteria:              *
* - if *AMPMind is "AM" or "PM", *hourIn ranges from "01" to "12" *
* - otherwise, *hourIn ranges from "0" to "23"                  *
* *If the appropriate criterion is met, *hourOut is assigned as *
* follows:                                                    *
* - if *AMPMind is "AM" or "PM", *hourOut is assigned the 24-hour *
* clock equivalent of *hourIn.                                *
* - otherwise, *hourIn is copied to *hourOut                    *
* and checkHour returns OK.                                    *
* *If the appropriate criterion is not met, *hourOut is assigned *
* as NULLCHAR, and checkHour returns NOT_OK.                   *
*                                                                 *
short int   func_status = OK;       /* function status indicator  */
short int   i;                      /* loop control               */
{/*********************************************************************/
* NULLCHAR, and checkHour returns NOT_OK.                        *
* If the appropriate criterion is met, *hourOut is assigned       *
* as otherwise, *hourIn is copied to *hourOut.                    *
* - if *AMPMind is "AM" or "PM", *hourOut is assigned as 24-hour    *
* clock equivalent of *hourIn.                                   *
* - otherwise, *hourIn ranges from "0" to "23"                   *
* - if *AMPMind is AM or PM, convert *hourIn to 24-clock format   *
* if (strcmp( AMPMind,"AM" ) == MATCH || strcmp( AMPMind,"PM" ) == MATCH )
func_status = set24HrClock( hourIn,AMPMind );
{/*******************************************************************/
* if AMPMind not "AM" or "PM", verify hourIn ranges from 00 to 23  *
*                                                                 *
else
{/*********************************************************************/
* If the appropriate criterion is not met, *hourOut is assigned     *
* as otherwise, *hourIn is copied to *hourOut.                    *
* - if *AMPMind is AM or PM, convert *hourIn to 24-clock format   *
* if (strcmp( AMPMind,"AM" ) == MATCH || strcmp( AMPMind,"PM" ) == MATCH )
func_status = set24HrClock( hourIn,AMPMind );
{/*******************************************************************/
* if AMPMind not "AM" or "PM", verify hourIn ranges from 00 to 23  *
*                                                                 *
else
{/*********************************************************************/
* If the appropriate criterion is not met, *hourOut is assigned     *
* as otherwise, *hourIn is copied to *hourOut.                    *
* - if *AMPMind is AM or PM, convert *hourIn to 24-clock format   *
* if (strcmp( AMPMind,"AM" ) == MATCH || strcmp( AMPMind,"PM" ) == MATCH )
func_status = set24HrClock( hourIn,AMPMind );
{/*******************************************************************/
* if AMPMind not "AM" or "PM", verify hourIn ranges from 00 to 23  *
*                                                                 *
else
{/*********************************************************************/
* If the appropriate criterion is not met, *hourOut is assigned     *
* as otherwise, *hourIn is copied to *hourOut.                    *
* - if *AMPMind is AM or PM, convert *hourIn to 24-clock format   *
* if (strcmp( AMPMind,"AM" ) == MATCH || strcmp( AMPMind,"PM" ) == MATCH )
func_status = set24HrClock( hourIn,AMPMind );
{/*******************************************************************/
* if AMPMind not "AM" or "PM", verify hourIn ranges from 00 to 23  *
*                                                                 *
else
{/*********************************************************************/
* If the appropriate criterion is not met, *hourOut is assigned     *
* as otherwise, *hourIn is copied to *hourOut.                    *
* - if *AMPMind is AM or PM, convert *hourIn to 24-clock format   *
* if (strcmp( AMPMind,"AM" ) == MATCH || strcmp( AMPMind,"PM" ) == MATCH )
func_status = set24HrClock( hourIn,AMPMind );
{/*******************************************************************/
* if AMPMind not "AM" or "PM", verify hourIn ranges from 00 to 23  *
*                                                                 *
else
{/*********************************************************************/
* If the appropriate criterion is not met, *hourOut is assigned     *
* as otherwise, *hourIn is copied to *hourOut.                    *
* - if *AMPMind is AM or PM, convert *hourIn to 24-clock format   *
* if (strcmp( AMPMind,"AM" ) == MATCH || strcmp( AMPMind,"PM" ) == MATCH )
func_status = set24HrClock( hourIn,AMPMind );
{/*******************************************************************/
* if AMPMind not "AM" or "PM", verify hourIn ranges from 00 to 23  *
*                                                                 *
else
{/*********************************************************************/
* If the appropriate criterion is not met, *hourOut is assigned     *
* as otherwise, *hourIn is copied to *hourOut.                    *
* - if *AMPMind is AM or PM, convert *hourIn to 24-clock format   *
* if (strcmp( AMPMind,"AM" ) == MATCH || strcmp( AMPMind,"PM" ) == MATCH )
func_status = set24HrClock( hourIn,AMPMind );
{/*******************************************************************/
* if AMPMind not "AM" or "PM", verify hourIn ranges from 00 to 23  *
*                                                                 *
return( func_status );
} /* end checkHour */

int checkMinute
(char        *minOut,                /* out: minute, validated    */
char        *minIn,                /* in: minute, unvalidated   */
)
{/*******************************************************************/
* Verifies that *minIn is 2 bytes of numeric characters between "00" *
* and "59".
*/
# Chapter 12. Sample data and applications supplied with Db2 for z/OS

1219

```c
short int  i;    /* loop control */
short int  func_status = OK;    /* function status indicator */

/* verify that *minIn is 2 numeric characters between "00" and "59" */
if( strlen( minIn ) != 2 )    
    func_status = NOT_OK;
else if( isdigit(minIn[0]) == MATCH || isdigit(minIn[1]) == MATCH )    
    func_status = NOT_OK;
else if( strcmp( minIn,"00" ) < 0 || strcmp( minIn,"59" ) > 0 )    
    func_status = NOT_OK;

/* if minIn is valid, assign it to minOut */
if( func_status == OK )    
    strcpy( minOut,minIn );
else    
    minOut[0] = NULLCHAR;

return( func_status );
} /* end checkMinute */

int check AMPM indicator
(char  *indOut,    /* out: AMP ind, validated */
char  *indIn      /* in: AMP ind, unvalidated */)    

/* verify that *indIn is 2 bytes, containing either "AM" or "PM" */
if( strcmp( indIn,"AM" ) == 0 || strcmp( indIn,"PM" ) == 0 )    
    func_status = OK;
else if( strcmp( indIn,"AM" ) < 0 || strcmp( indIn,"PM" ) > 0 )    
    func_status = NOT_OK;

/* if indIn is copied to indOut and checkAMP M indicator returns OK */
if( func_status == OK )    
    strcpy( indOut,indIn );
else    
    indOut[0] = NULLCHAR;

return( func_status );
} /* end checkAMP M indicator */
```
int set24HrClock
(char *hour,                  /* in/out: hour               */
char *AMPMind                /* in: AM/PM indicator        */
) return( func_status );              /* return function status     */

int set12HrClock
(char *hour,                  /* in/out: hour               */
char *AMPMind                /* out: AM/PM indicator      */
) return( func_status );              /* return function status     */

/*********************************************************************/
* assign *hour its 12-hour clock equivalent                        *
*********************************************************************/

int checkAMPMindicators
(char *hour) / in/out: hour */
{ return( func_status );              /* return function status     */

/*********************************************************************/
* locate *hour in the 24-hour clock map                            *
*********************************************************************/

int check24HrClock
(char *hour) / in/out: hour */
{ return( func_status );              /* return function status     */

/*********************************************************************/
* If the incoming value for *hour is:                              *
*********************************************************************/

return( func_status );              /* return function status     */

/*********************************************************************/
* verify that *indIn is 2 bytes containing either "AM" or "PM"     *
*********************************************************************/

return( func_status );              /* return function status     */

/*********************************************************************/
* Changes *hour from 24-hour clock format to 12-hour clock format.   *
*********************************************************************/

/*********************************************************************/
* if indIn is valid, assign it to indOut                           *
*********************************************************************/

/*********************************************************************/
* if indIn is valid, assign it to indOut                           *
*********************************************************************/
Chapter 12. Sample data and applications supplied with Db2 for z/OS 1221

* Otherwise:
  * - if *AMPMind is "AM", then *hour is assigned the 24-hour equiva-
  * lent of morning hour ("00"-"11") and set24HrClock returns OK.  * 09250000
  * - else *hour is assigned the 24-hour equivalent of afternoon
  * hour ("12"-"23") and set24HrClock returns OK.  * 09260000
  * *******************************************************************/ 09270000
  * otherwise, left-shift non-zero chars and terminate string        * 09280000
  * ******************************************************************* 09290000
  *strcpy( string,"0" );                                              09300000
  *if( string[i] == '\0' )                                              09310000
  *for( i=0; i<12 & strcmp( hour,clock12[i] ) == MATCH; i++ );         09320000
  *assign *hour its 24-hour clock equivalent                          09330000
  *assign *hour its 24-hour clock equivalent                          09340000
  *if( i > 11 ) /* if hour not betw/ 01 & 12 */ 09350000
  *func_status = NOT_OK; /* ...set error flag */ 09360000
  *else if(strcmp(AMPMind,"AM")!=MATCH)/ else if betw/ 12 AM - 11 AM/* 09370000
  *strcpy( hour,clock24[i] ); /* ...set hour in 12-hour fmt */ 09380000
  *else /* else betw/ 12 PM - 11 PM */ 09390000
  *strcpy( hour,clock24[i+12] ); /* ...set hour in 12-hour fmt */ 09400000
  *return( func_status ); /* return function status */ 09410000
  */ end set24HrClock */ 09420000

void remove0Prefix
(char        *str3                   /* in/out: string to prefix   */ 09430000
(void)
{                                                                      09440000
  /*********************************************************************/ 09450000
  * Prefixes *str3 with a leading 0 if it is only 1 byte long        * 09460000
  /********************************************************************* 09470000
  if( strlen( str3 ) == 1 )                                            09480000
  /*******************************************************************/ 09490000
  * if str3 is just 1 byte long, prefix it with a "0"                * 09500000
  /******************************************************************* 09510000
  {                                                                      09520000
    /*********************************************************************/ 09530000
    * assign *hour its 24-hour clock equivalent                        09540000
    /******************************************************************* 09550000
    if( string[0] == '0' )                      /* ...set hour in 12-hour fmt  */ 09560000
    else if(strcmp(AMPMind,"AM")==MATCH)/* else if betw/ 12 AM - 11 AM*/ 09570000
    func_status = NOT_OK;             /* ...set error flag           */ 09580000
    else                                 /* else betw/ 12 PM - 11 PM   */ 09590000
    return( func_status ); /* return function status */ 09600000
  } /* end remove0Prefix */ 09610000

void add0Pref
(char        *str3                   /* in/out: string to prefix   */ 09620000
(void)
{                                                                      09630000
  /*********************************************************************/ 09640000
  * add0Pref(): adds a leading 0 if str3 is all zeroes.              * 09650000
  /********************************************************************* 09660000
  for( i=0; string[i] == '0'; i++ );                                 09670000
  /*********************************************************************/ 09680000
  * locate *hour in the 12-hour clock map                             * 09690000
  /*******************************************************************/ 09700000
  * assign *hour its 24-hour clock equivalent                          * 09710000
  /******************************************************************* 09720000
  short int   i = 0;                  /* Loop control               */ 09730000
  short int   j = 0;                  /* Loop control               */ 09740000
  /*********************************************************************/ 09750000
  *   hour ("12"-"23") and set24HrClock returns OK.                    * 09760000
  */ end add0Pref */ 09770000

short int   func_status = OK;       /* function status indicator  */ 09780000
short int   i;                      /* loop control               */ 09790000
short int   j = 0;                  /* Loop control               */ 09800000
/*********************************************************************/ 09810000
short int   func_status = OK;       /* function status indicator  */ 09820000
short int   i;                      /* loop control               */ 09830000
short int   j = 0;                  /* Loop control               */ 09840000
 /*********************************************************************/ 09850000
* assign *hour its 24-hour clock equivalent                          * 09860000
 /******************************************************************* 09870000
if( i > 11 ) /* if hour not betw/ 01 & 12 */ 09880000
func_status = NOT_OK; /* ...set error flag */ 09890000
else if(strcmp(AMPMind,"AM")!=MATCH)/ else if betw/ 12 AM - 11 AM/* 09900000
strcpy( hour,clock24[i] ); /* ...set hour in 12-hour fmt */ 09910000
else /* else betw/ 12 PM - 11 PM */ 09920000
strcpy( hour,clock24[i+12] ); /* ...set hour in 12-hour fmt */ 09930000
return( func_status ); /* return function status */ 09940000
} /* end add0Pref */ 09950000

short int   j = 0;                  /* Loop control               */ 09960000
short int   i = 0;                  /* Loop control               */ 09970000
 /*********************************************************************/ 09980000
short int   func_status = OK;       /* function status indicator  */ 09990000
short int   i;                      /* loop control               */ 09990000
short int   j = 0;                  /* Loop control               */ 09990000
 /*********************************************************************/ 1000000
* assign *hour its 24-hour clock equivalent                          * 10010000
 /******************************************************************* 10020000
if( string[0] == '0' )                      /* ...set hour in 12-hour fmt  */ 10030000
else if(strcmp(AMPMind,"AM")==MATCH)/* else if betw/ 12 AM - 11 AM*/ 10040000
func_status = NOT_OK;             /* ...set error flag           */ 10050000
else                                 /* else betw/ 12 PM - 11 PM   */ 10060000
return( func_status ); /* return function status */ 10070000
} /* end add0Pref */ 10080000

for( i=0; i<12 & strcmp( hour,clock12[i] ) == MATCH; i++ );         10090000
for( i=0; string[i] == '0'; i++ );                                 10100000
*assign *hour its 24-hour clock equivalent                          * 10110000
 /******************************************************************* 10120000
* otherwise, left-shift non-zero chars and terminate string         * 10130000
 /*********************************************************************/ 10140000
else
{                                                                      10150000
    for( j=0; string[i] != NULLCHAR; j++ ) 10160000
    string[j] = string[i++];           10170000
    string[j] = NULLCHAR;               10180000
} /* end remove0Prefix */ 10190000
**DSN8DUCY**

 Formats a given numeric amount with a specified currency symbol and, if specified, one of the following debit/credit indicators.

```c
/******************************************
 * Module name = DSN8DUCY (DB2 sample program)
 * DESCRIPITVE NAME = General currency formatter (UDF)
 * LICENSED MATERIALS - PROPERTY OF IBM
 * 5675-DB2
 * (C) COPYRIGHT 1998, 2000 IBM CORP. ALL RIGHTS RESERVED.
 * STATUS = VERSION 7
 * Function: Formats a given numeric amount with a specified currency
 * symbol and, if specified, one of the following debit/credit indicators.
 * */ -/:: Place a hyphen between the currency symbol and the
 * amount if the amount is less than 0.
 * (/): Place a left parenthesis between currency symbol
 * and the amount and place a right parenthesis to the
 * right of the amount if the amount is less than 0.
 * CR/DB: Place CR to the right of the amount if it is less
 * than 0; otherwise place DB to the right of the
 * amount.
 * Example invocations:
 * EXEC SQL SET :money = CURRENCY( -123,
 * "DM" );
 * ==> money = DM -123.00
 * EXEC SQL SET :money = CURRENCY( -123,
 * "DM", "(" );
 * ==> money = DM (123.00)
 * Notes:
 * Dependencies: Requires IBM C/C++ for OS/390 V1R3 or higher
 * Restrictions:
 * Module type: C program
 * Processor: IBM C/C++ for OS/390 V1R3 or higher
 * Module size: See linkedit output
 * Attributes: Re-entrant and re-usable
 * Entry Point: DSN8DUCY
 * Purpose: See Function
 * Linkage: DSN2SQL
 * Invoked via SQL UDF call
 * Parameters: DSN8DUCY uses the C "main" argument convention of
 * argv (argument vector) and argc (argument count).
 * The location of input and output parameters depends
 * on whether the CURRENCY UDF is invoked with two
 * input arguments (amount and currency symbol) or with
 * three input arguments (amount, currency symbol, and
 * credit/debit indicator).
 * If the CURRENCY UDF is invoked with two arguments
 * only (an amount and a currency symbol):
 * - ARGV[0] = (input) pointer to a char[3], null-
 * terminated string having the name of
 * this program (DSN8DUCY)
 * - ARGV[1] = (input) pointer to a double word having
 * the amount to be formatted as currency.
 * - ARGV[2] = (input) size having the currency
```
If the CURRENCY UDF is invoked with three arguments (an amount, a currency symbol, and a credit/debit indicator): * 00009900

- ARGV[0] = (input) pointer to a char[9], null- terminated string having the name of this program (DSN8DUCY) * 00010000
- ARGV[1] = (input) pointer to a double word having the amount to be formatted as currency. * 00010100
- ARGV[2] = (input) pointer to a char[3], null- terminated string having the currency symbol. * 00010200
- ARGV[3] = (input) pointer to a char[6], null- terminated string having the credit/debit indicator (see under "Function:", above, for valid credit/debit indicators) * 00010300
- ARGV[4] = (output) pointer to a char[28], null- terminated string to receive the currency result. * 00010400
- ARGV[5] = (input) pointer to a short integer having the null indicator for the input amount * 00010500
- ARGV[6] = (input) pointer to a short integer having the null indicator for the currency symbol * 00010600
- ARGV[7] = (input) pointer to a short integer having the null indicator for the credit/debit indicator * 00010700
- ARGV[8] = (output) pointer to a short integer having the null indicator for the result * 00010800
- ARGV[9] = (output) pointer to a char[6], null- terminated string to receive the SQLSTATE * 00010900
- ARGV[10] = (input) pointer to a char[138], null- terminated string having the UDF family name of the function * 00011000
- ARGV[11] = (input) pointer to a char[129], null- terminated string having the UDF specific name of the function * 00011100
- ARGV[12] = (output) pointer to a char[76], null- terminated string to receive any diagnostic message issued by this function * 00011200

* Normal Exit: Return Code: SQLSTATE = 000000 * 00011300
  * Message: none * 00011400
  * Error Exit: Return Code: SQLSTATE = 38601 * 00011500
    * Message: DSN8DUCY Error: No amount entered * 00011600
    * Message: DSN8DUCY Error: No currency symbol entered * 00011700
    * Return Code: SQLSTATE = 38602 * 00011800
      * Message: DSN8DUCY Error: Error performing setlocale() * 00011900
      * External References: * 00012000
        * Routines/Services: None * 00012100
- Data areas : None
- Control blocks : None

Pseudocode:

- DNSBDUCY:

- Walk down the argv list, locating the input and output parms
- Issue sqlstate 38601 and a diagnostic message if no input amount was provided.
- Issue sqlstate 38601 and a diagnostic message if no currency symbol was provided.
- Call formatAmount to assemble the currency expression from the input amount and the currency symbol and, optionally, the credit/debit indicator.
- Call formatAmount to assemble the currency expression from the input amount and the currency symbol and, optionally, the credit/debit indicator.
- Issue sqlstate 38601 and a diagnostic message if no input amount was provided.
- Call formatAmount to assemble the currency expression from the input amount and the currency symbol and, optionally, the credit/debit indicator.

#include <localdef.h>
#include <monetary.h>
#include <stdio.h>
#include <monetary.h>
#include <localdef.h>
#include <stdlib.h>
#include <monetary.h>
#include <stdio.h>

Pseudocode:

- Control blocks : None
- Data areas : None

- if a CR/DB indicator of -/+ has been specified, prefix the currency expression with a hyphen.
- if a CR/DB indicator of (-/) has been specified, prefix the currency expression with a left parenthesis.
- Append the currency symbol to the currency expression.
- If the currency symbol is just 1 byte, concatenate a blank.
- Call the C function setlocale() to initialize the C function.
- Call the C function strftime() to reformat the input amount with the currency symbol.
- If the amount in is less than zero ...
- If no errors, unset null indicators, and return SQLSTATE 00000.
- If a CR/DB of (-/) has been specified, append a right parenthesis to the currency expression.

- Call formatAmount to assemble the currency expression from the input amount and the currency symbol and, optionally, the credit/debit indicator.
- If no errors, unset null indicators, and return SQLSTATE 00000.
- If a CR/DB indicator of -/+ has been specified, prefix the currency expression with a hyphen.
- If no errors, unset null indicators, and return SQLSTATE 00000.

- Issue sqlstate 38601 and a diagnostic message if no input amount was provided.
- Issue sqlstate 38601 and a diagnostic message if no currency symbol was provided.
- Call formatAmount to assemble the currency expression from the input amount and the currency symbol and, optionally, the credit/debit indicator.

- If a CR/DB indicator of -/+ has been specified, prefix the currency expression with a hyphen.
- If a CR/DB indicator of (-/) has been specified, prefix the currency expression with a left parenthesis.
- Append the currency symbol to the currency expression.
- If the currency symbol is just 1 byte, concatenate a blank.
- Call the C function setlocale() to initialize the C function.
- Call the C function strftime() to reformat the input amount with the currency symbol.
- If the amount in is less than zero ...
- If no errors, unset null indicators, and return SQLSTATE 00000.
- If a CR/DB of (-/) has been specified, append a right parenthesis to the currency expression.

- Call formatAmount to assemble the currency expression from the input amount and the currency symbol and, optionally, the credit/debit indicator.
- If no errors, unset null indicators, and return SQLSTATE 00000.
- If a CR/DB indicator of -/+ has been specified, prefix the currency expression with a hyphen.
- If no errors, unset null indicators, and return SQLSTATE 00000.

- Issue sqlstate 38601 and a diagnostic message if no input amount was provided.
- Issue sqlstate 38601 and a diagnostic message if no currency symbol was provided.
- Call formatAmount to assemble the currency expression from the input amount and the currency symbol and, optionally, the credit/debit indicator.
/* vars for argument vector */

double *amountIn; /* in: value to be formatted */
char *currSymbol[3]; /* in: currency symbol */
char *CRDBInd[6]; /* in: credit/debit indicator */
char *moneyOut; /* out: formatted amountIn */
short int *niAmountIn; /* in: indic var, amountIn */
short int *niCurrSymbol; /* in: indic var, currSymbol */
short int *niCRDBInd; /* in: indic var, CRDBInd */
char *sqlstate; /* out: diagnostic message */
char *message; /* out: null indicator */
char fnName[138]; /* in: family name of function */
char specificName[129]; /* in: specific name of func */
short int status = OK; /* DSN8DUCY run status */

(argc--) ; /* convert argc to base 0 index*/

message = (char *)argv[--argc]; /* out: point to UDF diag msg */
strcpy( specificName, argv[--argc]); /* in: save UDF specific name */
strcpy( fnName, argv[--argc]); /* in: save UDF function name */
sqlstate = (char *)argv[--argc]; /* out: point to UDF sqlstate */
niMoneyOut = (short int *)argv[--argc]; /* out: point to null indicator */

if(argc == 7) /* if 3 input parms passed */

\n\n\nelse /* otherwise it wasn't passed */

\n\n\nstatus = NOT_OK; /* DSN8DUCY Error: No amount entered */
strcpy( sqlstate, "38601" );
strcpy( message, "DSN8DUCY Error: No amount entered" );
strcpy( sqlstate, "38601" );
}

else if(*niCurrSymbol || ( strlen(currSymbol) == 0 ))

/* Verify that an amount and a currency symbol have been passed in */

if(*niAmountIn )

    status = NOT_OK;
    strcpy( message, "DSN8DUCY Error: No amount entered" );
    strcpy( sqlstate, "38601" );
}
else if(*niCurrSymbol || ( strlen(currSymbol) == 0 ))

    status = NOT_OK;
### Example Code

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

char *moneyOut, *message, *sqlstate;
int status = OK;

int main()
{
    int negFlag = 0;
    char *CRDBInd, *currSymbol, *moneyOut, *sqlstate,
    *message, *moneyStr[200],
    *sqlstate;
    double amount;

    strcpy(moneyStr[0], "000000");
    strcpy(sqlstate, "38601");
    strcpy(sqlstate, "DSN8DUCY Error: No currency symbol entered");
    strcpy(message, "DSN8DUCY Error: No currency symbol entered");

    // Convert amount to string
    if (status == OK) {
        status = formatAmount(moneyOut, message, sqlstate,
                              amount, currSymbol, CRDBInd);
    }
    // Handle errors
    else {
        // Clear moneyOut
        moneyOut[0] = NULLCHAR;
        niMoneyOut = -1;
        moneyOut = NULLCHAR;
        sqlstate[0] = NULLCHAR;
    }

    return 0;
}
```

### Explanation

The code snippet provided is a function `formatAmount` that takes the following parameters:

- `moneyOut`: A character array to store the formatted amount.
- `message`: A character array to store the diagnostic message.
- `sqlstate`: A character array to store the SQLSTATE.
- `amountIn`: A double precision double to store the amount to be formatted.
- `currSymbol`: A character string to store the currency symbol.
- `CRDBInd`: A character string to store the credit/debit indicator.

The function is designed to format an amount into a string representation, including the currency type as specified by `currSymbol`. If an error occurs during the formatting process, the function clears the `moneyOut` buffer and sets the SQL null indicator for `moneyOut`. The SQLSTATE is also set to a predefined value to indicate the error condition.

The `main` function demonstrates the usage of `formatAmount` and handles the errors gracefully by clearing the output buffer and setting the SQL null indicator. The diagnostic message is also printed for error conditions.

### Additional Notes

- The function `formatAmount` is designed to handle various currency formats, allowing for flexibility in currency representation.
- The code snippet also includes error handling to ensure that the output is properly managed in case of failures.

This example is part of the Db2 12 for z/OS: Application Programming and SQL Guide, providing insights into the programming and SQL capabilities of the Db2 database management system.
```c
amount = -amount;
if( strcmp( CRDBInd, "+/-" ) == 0 )
    strcpy( moneyOut, "-" );
else if( strcmp( CRDBInd, "/()" ) == 0 )
    strcpy( moneyOut, "(" );
}

="/*******************************************************************
* Append the currency type (currSymbol) to moneyOut. If currSymbol* 00040800
* is more than one byte, place a blank between it and the amount   * 00040900
*******************************************************************/
strcat( moneyOut, currSymbol );
if( strlen( currSymbol ) > 1 )
    strcat( moneyOut, " ");
00042600

="/*******************************************************************
* Set the local for the strfmon function                           * 00042900
*******************************************************************/
if( setlocale( LC_ALL, "En_US" ) == NULL )
{
    strcpy( message, "DSN8DUCY Error: Error performing setlocale()" );
    strcpy( sqlstate, "38602" );
    return( NOT_OK );
}
00042500

="/*******************************************************************
* Reformat amount to a string type with thousands grouping         * 00043000
*******************************************************************/
strfmon( moneyStr, 100, ",%:n", amount );
strcat( moneyOut, moneyStr );
00043100

="/******************************************************************* 00043200
* If amount < 0, append negative currency indicators, if passed    * 00043300
*******************************************************************/
if( CRDBInd[0] != NULLCHAR )
{
    if( negFlag == 1 )
    {
        if( strcmp( CRDBInd, "CR/DB" ) == 0 )
            strcat( moneyOut, " DB" );
        else if( strcmp( CRDBInd, "/()" ) == 0 )
            strcat( moneyOut, " DB" );
    }
    else
    {
        if( strcmp( CRDBInd, "CR/DB" ) == 0 )
            strcat( moneyOut, " CR" );
    }
}
return( OK );

="/************************************************************
* Sample applications in TSO on page 1027                   * 00043800
* A set of Db2 sample applications run in the TSO environment. * 00043900
* DSN8DUTI                                                    * 00043900
* Returns the name or the schema name or the location name of an alias according to the name of the UDF and the number of input parameters passed, as follows. * 00044000
************************************************************************

Related reference
“Sample applications in TSO” on page 1027
A set of Db2 sample applications run in the TSO environment.

DSN8DUTI
Returns the name or the schema name or the location name of an alias according to the name of the UDF and the number of input parameters passed, as follows.

(strlen of name)
TABLE_NAME( objectname ) returns the unqualified name of the object found after any alias chains have been resolved. The specified object name, the default schema, and a location name of "%" (for any location) are used as the starting point of the resolution. If the starting point does not refer to an alias, the unqualified name of the starting point is returned. The resulting name may be of a table, view, or undefined object.

TABLE_NAME( objectname, objectschema ) returns the unqualified name of the object found after any alias chains have been resolved. The specified object name and schema, and a location name of "%" (for any location), are used as the starting point of the resolution. If the starting point does not refer to an alias, the unqualified name of the starting point is returned. The resulting name may be of a table, view, or undefined object.

TABLE_NAME( objectname, objectschema, objectlocation ) returns the unqualified name of the object found after any alias chains have been resolved. The specified object name, schema, and a location name of "%" (for any location), are used as the starting point of the resolution. If the starting point does not refer to an alias, the unqualified name of the starting point is returned. The resulting name may be of a table, view, or undefined object.

TABLE_SCHEMA( objectname ) returns the schema name of the object found after any alias chains have been resolved. The specified object name, the default schema, and a location name of "%" (for any location) are used as the starting point of the resolution. If the starting point does not refer to an alias, the schema name of the starting point is returned. The resulting schema name may be of a table, view, or undefined object.

TABLE_SCHEMA( objectname, objectschema ) returns the schema name of the object found after any alias chains have been resolved. The specified object name and schema, and a location name of "%" (for any location) are used as the starting point of the resolution. If the starting point does not refer to an alias, the schema name of the starting point is returned. The resulting schema name may be of a table, view, or undefined object.

TABLE_SCHEMA( objectname, objectschema, objectlocation ) returns the schema name of the object found after any alias chains have been resolved. The specified object name, schema, and a location name of "%" (for any location) are used as the starting point of the resolution. If the starting point does not refer to an alias, the schema name of the starting point is returned. The resulting schema name may be of a table, view, or undefined object.

TABLE_NAME( objectname ) returns the unqualified name of the object found after any alias chains have been resolved. The specified object name, the default schema, and a location name of "%" (for any location) are used as the starting point of the resolution. If the starting point does not refer to an alias, the unqualified name of the starting point is returned. The resulting name may be of a table, view, or undefined object.

TABLE_LOCATION( objectname ) returns the location name of the object found after any alias chains have been resolved. The specified object name, the default schema, and a location name of "%" (for any location) are used as the starting point of the resolution. If the starting point does not refer to an alias, the blank location name (indicating the current server) is returned. The resulting location name may be of a table, view, or undefined object.

TABLE_LOCATION( objectname, objectschema ) returns the location name of the object found after any alias chains have been resolved. The specified object name and schema, and a location name of "%" (for any location) are used as the starting point of the resolution. If the starting point does not refer to an alias, the blank location name (indicating the current server) is returned. The resulting location name may be of a table, view, or undefined object.
table, view, or undefined object.

**TABLE_LOCATION( objectname, objectschema, objectlocation )**

returns the location name of the object found after any alias chains have been resolved. The specified object name, schema, and location name are used as the starting point of the resolution. If the starting point does not refer to an alias, a blank location name (indicating the current server) is returned. The resulting location name may be of a table, view, or undefined object.

Notes:
- Dependencies: Requires IBM C/C++ for OS/390 V1R3 or higher
- Restrictions:
- Module type: C program
- Processor: IBM C/C++ for OS/390 V1R3 or subsequent release
- Module size: See linkedit output
- Attributes: Re-entrant and re-usable
- Entry Point: CEESTART (Language Environment entry point)
- Purpose: See Function
- Linkage: DB2SQL
- Invoked via SQL UDF call

Parameters: DSN8DUTI uses the C "main" argument convention of argv (argument vector) and argc (argument count).
- The location of input and output parameters depends on whether the UDF (TABLE_NAME, TABLE_SCHEMA, or TABLE_SCHEMA) is invoked with one, two, or three input arguments.
- If the UDF was invoked with the object name only:
  - ARGV[0] = (input) pointer to a char[9], null-terminated string having the name of this program (DSN8DUTI)
  - ARGV[1] = (input) pointer to a char[19], null-terminated string having the object name to be used as the starting point of the alias resolution
  - ARGV[2] = (output) pointer to a null-terminated string to receive the result as follows:
    - char[19] for the TABLE_NAME UDF
    - char[9] for the TABLE_SCHEMA UDF
    - char[17] for the TABLE_LOCATION UDF
  - ARGV[3] = (input) pointer to a short integer having the null indicator for the object name
  - ARGV[4] = (output) pointer to a short integer
    - having the null indicator for the result
  - ARGV[5] = (output) pointer to a char[6], null-terminated string to receive the SQLSTATE
  - ARGV[6] = (input) pointer to a char[138], null-terminated string having the UDF family name of the function
  - ARGV[7] = (input) pointer to a char[129], null-terminated string having the UDF specific name of the function
  - ARGV[8] = (output) pointer to a char[78], null-terminated string to receive any diagnostic message issued by this function
- If the UDF was invoked with the object name and the object schema (but not the object location name):
  - ARGV[0] = (input) pointer to a char[9], null-terminated string having the name of this program (DSN8DUTI)
  - ARGV[1] = (input) pointer to a char[19], null-terminated string having the object name to be used in conjunction with the object schema as the starting point of the alias resolution
  - ARGV[2] = (input) pointer to a char[9], null-terminated string having the object schema to be in used in conjunction with the object name as the starting point of

Chapter 12. Sample data and applications supplied with Db2 for z/OS 1229
* - ARGV[3] = (output) pointer to a null-terminated string to receive the result as follows: * 018790000 
  - char[19] for the TABLE_NAME UDF * 018900000 
  - char[9] for the TABLE_SCHEMA UDF * 019000000 
  - char[17] for the TABLE_LOCATION UDF * 019100000 
* - ARGV[4] = (input) pointer to a short integer * 019200000 
  having the null indicator for the object name * 019300000 
* - ARGV[5] = (input) pointer to a short integer * 019400000 
  having the null indicator for the object schema * 019500000 
* - ARGV[6] = (output) pointer to a short integer * 019600000 
  having the null indicator for the result SQLSTATE * 019700000 
* - ARGV[7] = (output) pointer to a char[6], null-terminated string to receive the * 019800000 
  SQLSTATE * 019900000 
* - ARGV[8] = (input) pointer to a char[138], null-terminated string having the UDF family name of the function * 020000000 
* - ARGV[9] = (input) pointer to a char[129], null-terminated string having the UDF specific name of the function * 020100000 
* - ARGV[10] = (output) pointer to a char[79], null-terminated string to receive any diagnostic message issued by this function * 020200000 
* - ARGV[11] = (output) pointer to a null-terminated string having the object name and the object location name: * 020300000 
* - ARGV[12] = (input) pointer to a char[9], null-terminated string having the name of this program (DSNBUDUT1) * 020400000 
* - ARGV[13] = (input) pointer to a char[19], null-terminated string having the object name to be used in conjunction with the object schema and the object location name as the starting point of the alias resolution * 020500000 
* - ARGV[14] = (input) pointer to a char[9], null-terminated string having the object schema to be used in conjunction with the object name and the object location name as the starting point of the alias resolution * 020600000 
* - ARGV[15] = (input) pointer to a char[17], null-terminated string having the object location name to be used in conjunction with the object name and the object location name as the starting point of the alias resolution * 020700000 
* - ARGV[16] = (output) pointer to a null-terminated string to receive the result as follows: * 020800000 
  - char[19] for the TABLE_NAME UDF * 020900000 
  - char[9] for the TABLE_SCHEMA UDF * 021000000 
  - char[17] for the TABLE_LOCATION UDF * 021100000 
* - ARGV[17] = (input) pointer to a short integer * 021200000 
  having the null indicator for the object name * 021300000 
* - ARGV[18] = (input) pointer to a short integer * 021400000 
  having the null indicator for the object schema * 021500000 
* - ARGV[19] = (input) pointer to a short integer * 021600000 
  having the null indicator for the object location name * 021700000 
* - ARGV[20] = (output) pointer to a char[6], null-terminated string to receive the SQLSTATE * 021800000 
* - ARGV[21] = (input) pointer to a char[138], null-terminated string having the UDF family name of the function * 021900000 
* - ARGV[22] = (input) pointer to a char[129], null-terminated string having the UDF specific name of the function * 022000000 
* - ARGV[23] = (output) pointer to a char[79], null-terminated string to receive any diagnostic message issued by this function * 022100000 
* - ARGV[24] = (output) pointer to a null-terminated string having the alias resolution * 022200000 
  terminated string having the object name and the object location name: * 022300000 
* - ARGV[25] = (input) pointer to a char[9], null-terminated string having the name of this program (DSNBUDUT1) * 022400000 
* - ARGV[26] = (input) pointer to a char[19], null-terminated string having the object name to be used in conjunction with the object schema and the object location name as the starting point of the alias resolution * 022500000 
* - ARGV[27] = (input) pointer to a char[9], null-terminated string having the object schema to be used in conjunction with the object name and the object location name as the starting point of the alias resolution * 022600000 
* - ARGV[28] = (input) pointer to a char[17], null-terminated string having the object location name to be used in conjunction with the object name and the object location name as the starting point of the alias resolution * 022700000 
* - ARGV[29] = (output) pointer to a null-terminated string to receive the result as follows: * 022800000 
  - char[19] for the TABLE_NAME UDF * 022900000 
  - char[9] for the TABLE_SCHEMA UDF * 023000000 
  - char[17] for the TABLE_LOCATION UDF * 023100000 
* - ARGV[30] = (output) pointer to a short integer * 023200000 
  having the null indicator for the object name * 023300000 
* - ARGV[31] = (output) pointer to a short integer * 023400000 
  having the null indicator for the object schema * 023500000 
* - ARGV[32] = (output) pointer to a short integer * 023600000 
  having the null indicator for the object location name * 023700000 
* - ARGV[33] = (output) pointer to a char[6], null-terminated string to receive the SQLSTATE * 023800000 
* - ARGV[34] = (output) pointer to a char[138], null-terminated string having the UDF family name of the function * 023900000 
* - ARGV[35] = (input) pointer to a char[129], null-terminated string having the UDF specific name of the function * 024000000 
* - ARGV[36] = (output) pointer to a char[79], null-terminated string to receive any diagnostic message issued by this function * 024100000 
* - ARGV[37] = (output) pointer to a null-terminated string having the alias resolution * 024200000 
  terminated string having the object name and the object location name: * 024300000 
* - ARGV[38] = (input) pointer to a char[9], null-terminated string having the name of this program (DSNBUDUT1) * 024400000 
* - ARGV[39] = (input) pointer to a char[19], null-terminated string having the object name to be used in conjunction with the object schema and the object location name as the starting point of the alias resolution * 024500000 
* - ARGV[40] = (input) pointer to a short integer * 024600000 
  having the null indicator for the object name * 024700000 
* - ARGV[41] = (input) pointer to a short integer * 024800000 
  having the null indicator for the object schema * 024900000 
* - ARGV[42] = (input) pointer to a short integer * 025000000 
  having the null indicator for the object location name * 025100000 
* - ARGV[43] = (output) pointer to a short integer * 025200000 
  having the null indicator for the result SQLSTATE * 025300000 
* - ARGV[44] = (output) pointer to a char[6], null-terminated string to receive the SQLSTATE * 025400000 
* - ARGV[45] = (input) pointer to a char[138], null-terminated string having the UDF family name of the function * 025500000 
* - ARGV[46] = (input) pointer to a char[129], null-terminated string having the UDF specific name of the function * 025600000 
* - ARGV[47] = (output) pointer to a char[79], null-terminated string to receive any diagnostic message issued by this function * 025700000 
* - ARGV[48] = (output) pointer to a null-terminated string having the alias resolution * 025800000 
  terminated string having the object name and the object location name: * 025900000
EXEC SQL BEGIN DECLARE SECTION;

/**************************** DB2 Host Variables ************************/

EXEC SQL INCLUDE SQLCA;

/******************** DB2 SQL Communication Area ********************/

#define NOT_OK 0          /* Run status indicator: Error*/
#define NULLCHAR '\0'         /* Null character             */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include <db2.h>
#include <sys/param.h>
#include <sys/socket.h>
#include <sys/types.h>
#include <unistd.h>
#include <fcntl.h>

EXEC SQL INCLUDE SQLCA;

Chapter 12. Sample data and applications supplied with Db2 for z/OS
```c
int main( int argc, char *argv[] )
{
    /************** local variables *************/
    short int minus1 = -1; /* default null indic setting */
    char *result; /* result of this function */
    short int *niResult; /* indic var, result */
    char *sqlstate; /* SQLSTATE */
    char fnName[138]; /* function name */
    char specificName[129]; /* specific name of function */
    char *message; /* diagnostic message */
    short int status = OK; /* DSN8DUTI run status */

    /* Walk down the argv list, locating the input and output parms */
    argc--; /* convert argc to base 0 index*/
    message = (char *)argv[argc--]; /* out: point to UDF diag msg */
    strcpy( specificName, argv[argc--] ); /* in: save UDF specific name */
    strcpy( fnName, argv[argc--] ); /* in: save UDF function name */
    sqlstate = (char *)argv[argc--]; /* out: point to UDF sqlstate */
    niResult = (short int *)argv[argc--]; /* variable for result */
    if( argc == 7 ) /* if 3 input parms passed */
        niObjLocation = (short int *)argv[argc--]; /* in: point to null indic. */
    else /* otherwise it wasn't passed */
        niObjLocation = &minus1; /* so define it as null */
    if( argc >= 5 ) /* if 2 or 3 input parms passed */
        niObjSchema = (short int *)argv[argc--]; /* var for object schema */
    else /* otherwise it wasn't passed */
        niObjSchema = &minus1; /* so define it as null */
    niObjName = (short int *)argv[argc--]; /* in: point to null indicator */
    result = (char *)argv[argc--]; /* out: point to object name */
    if( argc == 3 ) /* if 3 input parms passed */
        strcpy( hvObjLocation, argv[argc--] ); /* name */
    else /* otherwise it wasn't passed */
        hvObjLocation[0] = NULLCHAR; /* so define it as null */
    if( argc >= 2 ) /* if 2 or 3 input parms passed */
        strcpy( hvObjSchema, argv[argc--] ); /* in: save object schema */
    else /* otherwise it wasn't passed */
        hvObjSchema[0] = NULLCHAR; /* so define it as null */
    strcpy( hvObjName, argv[argc] ); /* in: save object name */

    /* Initialize output parms */
    message[0] = NULLCHAR; /* out: NULLCHAR */
    strcpy( sqlstate, "00000" ); /* in: NULLCHAR */
    *niResult = 0; /* out: NULLCHAR */
    result[0] = NULLCHAR; /* out: NULLCHAR */

    /* If no object name provided, return null result */
    if( ( *niObjName != 0 ) || ( strlen( hvObjName ) == 0 ) )
        status = NOT_OK;
    return status;
}
```

if( SQLCODE == 0 ) {
  * If such an alias was found ...
  ***********************************************************/
  * TABLE_NAME UDF: return true name of table or view
  ***********************************************************/
  strcpyp( result,hvTBNAME );
  else if( strncmp( specificName,"DSN8DUTIS",9 ) == 0 )
    ***********************************************************/
    * TABLE_SCHEMA UDF: return true schema of table or view
    ***********************************************************/
    strcpyp( result,hvTBCREATOR );
  else if( strncmp( specificName,"DSN8DUTIL",9 ) == 0 )
    ***********************************************************/
    * TABLE_LOCATION UDF: return true loc'n of table or view
    ***********************************************************/
    strcpyp( result,hvLOCATION );
  else
    ***********************************************************/
    * If such an alias was found ...
    ***********************************************************/
    if( strncmp( specificName,"DSN8DUTIN",9 ) == 0 )
      ***********************************************************/
      * TABLE_NAME UDF: return starting point
      ***********************************************************/
      strcpyp( result,hvObjName );
    else if( strncmp( specificName,"DSN8DUTIL",9 ) == 0 )
      ***********************************************************/
      * TABLE_SCHEMA UDF: return schema of starting point
      ***********************************************************/
      strcpyp( result,hvObjSchema );
    else if( strncmp( specificName,"DSN8DUTIS",9 ) == 0 )
      ***********************************************************/
      * TABLE_LOCATION UDF: Remove trailing search wildcard byte
      ***********************************************************/
      hvObjLocation[strlen(hvObjLocation)-1] = NULLCHAR;
      strcpyp( result,hvObjLocation );
}

if( status == NOT_OK ) {
  * If no object schema provided, assign default schema
  ***********************************************************/
  EXEC SQL SET :hvObjSchema = CURRENT SQLID;
  ***********************************************************/
  * Concatenate "wildcard" to object location
  ***********************************************************/
  strcat( hvObjLocation,"%" );

  ***********************************************************/
  * Look for alias with the object name (and schema (and location))
  ***********************************************************/
  if( status == OK ) {
    if( SQLCODE == 0 ) {
      * If such an alias was found ...
      ***********************************************************/
      if( strncmp( specificName,"DSN8DUTIN",9 ) == 0 )
        ***********************************************************/
        * TABLE_NAME UDF: return true name of table or view
        ***********************************************************/
        strcpyp( result,hvTBNAME );
      else if( strncmp( specificName,"DSN8DUTIS",9 ) == 0 )
        ***********************************************************/
        * TABLE_SCHEMA UDF: return true schema of table or view
        ***********************************************************/
        strcpyp( result,hvTBCREATOR );
      else if( strncmp( specificName,"DSN8DUTIL",9 ) == 0 )
        ***********************************************************/
        * TABLE_LOCATION UDF: return true loc'n of table or view
        ***********************************************************/
        strcpyp( result,hvLOCATION );
      else
        ***********************************************************/
        * Unknown UDF: signal error
        ***********************************************************/
        status = NOT_OK;
        strcpyp( sqlstate,"38001" );
        sprintf( message,
          "DSN8DUTI Error: Invocation by unexpected UDF ",
          specificName );
    }
  }
}

else if( SQLCODE == 100 ) {
  * If no such alias was found ...
  ***********************************************************/
  if( strncmp( specificName,"DSN8DUTIN",9 ) == 0 )
    ***********************************************************/
    * TABLE_NAME UDF: return starting point
    ***********************************************************/
    strcpyp( result,hvObjName );
  else if( strncmp( specificName,"DSN8DUTIL",9 ) == 0 )
    ***********************************************************/
    * TABLE_SCHEMA UDF: return schema of starting point
    ***********************************************************/
    strcpyp( result,hvObjSchema );
  else if( strncmp( specificName,"DSN8DUTIS",9 ) == 0 )
    ***********************************************************/
    * TABLE_LOCATION UDF: Remove trailing search wildcard byte
    ***********************************************************/
    hvObjLocation[strlen(hvObjLocation)-1] = NULLCHAR;
    strcpyp( result,hvObjLocation );
}

Chapter 12. Sample data and applications supplied with Db2 for z/OS 1233
else
	/**
* Unknown UDF: signal error
***************************************************************************/
	{
	status = NOT_OK;
	strcpy( sqlstate,"38601" );
	sprintf( message,
	"DSN8DUTI Error: Invocation by unexpected UDF ",
	specificName );
	}
else
	/**
* If unexpected SQLCODE, issue message
*************************************************************/
{
	status = NOT_OK;
	strcpy( sqlstate,"38602" );
	sprintf( message,
	"DSN8DUTI Error: Unexpected SQLCODE, %d, ",
	specificName );
	}
} /* end if( status == OK ) */

/**********************************************************
* If null starting point or unexpected SQLCODE, return null result
***********************************************************/
if( status == NOT_OK ){
	result[0] = NULLCHAR;
	*niResult = -1;
} else {
	*niResult = 0;
	sprintf( sqlstate,"00000" );
}
} /* end DSN8DUTI */

Related reference
“Sample applications in TSO” on page 1027
A set of Db2 sample applications run in the TSO environment.

DSN8DUWC
Invokes the sample UDF table function WEATHER to demonstrate how a UDF and UDF table handling using static SQL.

******************************************************************
* Module name = DSN8DUWC (DB2 sample program)
* * DESCRIPTIVE NAME = Client for sample UDF table function WEATHER
* * LICENSED MATERIALS - PROPERTY OF IBM
* 5645-DB2
* (C) COPYRIGHT 1998 IBM CORP. ALL RIGHTS RESERVED.
* STATUS = VERSION 6
* Function: Invokes the sample UDF table function WEATHER to demonstrate how a UDF and UDF table handling using static SQL.
* Notes:
* Dependencies: Requires IBM C/C++ for OS/390 V1R3 or higher
* Restrictions:
* Module type: C program
* Processor: IBM C/C++ for OS/390 V1R3 or higher
* Module size: See linkedit output
* Attributes: Re-entrant and re-usable
* Entry Point: DSN8DUWC
Purpose: See Function

Parameters: DSN8DUWC uses the C "main" argument convention of argv (argument vector) and argc (argument count).

- ARGV[0] = (input) pointer to a char[9], null-terminated string having the name of this program (DSN8DUWC)
- ARGV[1] = (input) pointer to a char[45], null-terminated string having the name of the input data set.

Parameters:

- DSN8DUWC uses the C "main" argument convention of argv (argument vector) and argc (argument count).
- ARGV[0] = (input) pointer to a char[9], null-terminated string having the name of this program (DSN8DUWC)
- ARGV[1] = (input) pointer to a char[45], null-terminated string having the name of the source data for the weather reports.

Normal Exit: Return Code: 0000
- Message: none

Error Exit: Return Code: 0008
- Message: DSN8DUWC failed: Invalid parameter count
- Message: <formatted SQL text from DSNTIAR>

External References:
- Routines/Services: DSNTIAR: DB2 msg text formatter
- Data areas: None
- Control blocks: None

Pseudocode:

DSN8DUWC:
- Verify that 2 input parameters (program name and weather data set name) were passed.
- if not, issue diagnostic message and end with code 0008.
- Open WEATHER_CURSOR, the client cursor for the WEATHER UDF table function, passing the weather data set name as a host variable.
- if unsuccessful, call sql_error to issue a diagnostic message, then end with code 0008.
- Do while not end of cursor
  - Read the cursor
  - If successful, print the result
  - else if not end of cursor, call sql_error to issue a diagnostic message, then end with code 0008.
- Close the cursor
- if unsuccessful, call sql_error to issue a diagnostic message, then end with code 0008.

sql_error:
- call DSNTIAR to format the unexpected SQLCODE.

C library definitions:
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

Equates:
#define NULLCHAR '\0' /* Null character */
define OUTLEN 80 /* Length of output line */
define DATA_DIM 10 /* Number of message lines */
define NOT_OK 0 /* Run status indicator: Error */
define OK 1 /* Run status indicator: Good */

DB2 SQL Communication Area

HOST VARIABLES:

EXEC SQL INCLUDE SQLCA;
EXEC SQL BEGIN DECLARE SECTION;
char hvWeatherDSN[44]; /* host var for weather dsn */
short int nIWeatherDSN = 0; /* indic var for weather dsn */
char hvCity[31]; /* host var for name of city */
short int nICity = 0; /* indic var for city name */
long int hvTemp_in_f = 0; /* host var, fahrenheit temp */
short int     niTemp_in_f     = 0;  /* indic var for temperature */
long int      hvHumidity      = 0;  /* host var, percent humidity */
short int     niHumidity      = 0;  /* indic var for humidity */
char          hvWind[5];            /* host var, wind direction */
short int     niWind          = 0;  /* indic var for wind direct */
long int      hvWind_velocity = 0;  /* host var, wind velocity */
short int     niWind_velocity = 0;  /* indic var for wind velocity*/
double        hvBarometer     = 0;  /* host var, barometric press */
short int     niBarometer     = 0;  /* indic var for baro pressure*/
char          hvForecast[26];       /* host var, forecast */
short int     niForecast      = 0;  /* indic var for forecast */
EXEC SQL END DECLARE SECTION;

/*************************** DB2 SQL Cursor Declarations ********************/
EXEC SQL BEGIN DECLARE SECTION;
EXEC SQL DECLARE  WEATHER_CURSOR CURSOR  FOR
SELECT  CITY,
        TEMP_IN_F,
        HUMIDITY,
        WIND,
        WIND_VELOCITY,
        BAROMETER,
        FORECAST
FROM  TABLE( DSN8.WEATHER(:hvWeatherDSN) ) AS W
WHERE  CITY = 'Juneau, AK';
EXEC SQL END DECLARE SECTION;

/****************************** DB2 Message Formatter ******************************/
struct          error_struct          /* DSNTIAR message structure */
{              short int     error_len;
                char          error_text[DATA_DIM][OUTLEN];
            }              error_message = {DATA_DIM * (OUTLEN)};
#pragma         linkage( dsntiar, OS )
extern short int dsntiar( struct      sqlca         *sqlca,
struct      error_struct  *msg,
int                       *len );

/*************************** DSN8DUWC Global Variables ***************************/
short int       status = OK;          /* DSN8DUWC run status */
long int        completion_code = 0;  /* DSN8DUWC return code */

/*************************** DSN8DUWC Function Prototypes ***************************/
int main( int argc, char *argv[] );
void sql_error( char locmsg[] );

int main( int argc, char *argv[] )
/************************** DSN8DUWC Global Variables ***************************/
if(argc == 2)
    strcpy( hvWeatherDSN,argv[1] );
else
    printf("DSN8DUWC failed: Invalid parameter count\n");
    status = NOT_OK;
if(status == OK)
    EXEC SQL OPEN WEATHER_CURSOR;
    if( SQLCODE != 0 )
        sql_error(" *** Open weather cursor");
while( SQLCODE == 0  &&  status == OK )
{
    EXEC SQL FETCH  WEATHER_CURSOR
       INTO :hvCity    :niCity,
             :hvTemp_in_f :niTemp_in_f,
             :hvHumidity  :niHumidity,
             :hvWind     :niWind,
             :hvWind_velocity:niWind_velocity,
             :hvBarometer :niBarometer,
             :hvForecast :niForecast;
    if( SQLCODE == 0 )
    {
        printf( "Weather Report for %s\n", hvCity );
        printf( "... Temperature   : %d\n", hvTemp_in_f);
        printf( "... Humidity      : %d\n", hvHumidity );
        printf( "... Wind direction: %s\n", hvWind );
        printf( "... Wind velocity : %d\n", hvWind_velocity );
        printf( "... Barometer     : %.2f\n", hvBarometer );
        printf( "... Forecast      : %s\n", hvForecast );
    }
    else if( SQLCODE != 100 )
        sql_error( " *** Fetch from weather cursor" );
}
if( status == OK )
{
    EXEC SQL CLOSE WEATHER_CURSOR;
    if( SQLCODE != 0 )
        sql_error( " *** Close weather cursor" );
}
if( status != OK )
    completion_code = 8;
return( completion_code );
} /* end main */

/*********************************************************************/
/* set status to prevent further processing */
********************************************************************/
status = NOT_OK;

/*********************************************************************/
/* print the locator message */
********************************************************************/
printf( "%.80s\n", locmsg );

/*********************************************************************/
/* format and print the SQL message */
********************************************************************/
rc = dsntiar( &sqlca, &error_message, &lrecl );
if( rc == 0 )
    for( j=0; j<DATA_DIM; j++ )
    {
        for( k=0; k<OUTLEN; k++ )
            putchar(error_message.error_text[j][k] );
        putchar(\'\n\' );
    }
else
    {
        printf( " *** ERROR: DSNTIAR could not format the message\n" );
        printf( " *** SQLCODE is %d\n", SQLCODE );
        printf( " *** SQLERRM is \n" );
        for( j=0; j<sqlca.sqlerrml; j++ )
            printf( "%c", sqlca.sqlerrmc[j] );
    
Chapter 12. Sample data and applications supplied with Db2 for z/OS 1237
Related reference
“Sample applications in TSO” on page 1027
A set of Db2 sample applications run in the TSO environment.

DSN8DUWF

Returns weather information for various cities, as read from the data set passed as the argument to input parameter 'weatherDSN'.

>Returns weather information for various cities, as read from the data set passed as the argument to input parameter 'weatherDSN'. The data includes the name of a city followed by its weather information: Temperature in Fahrenheit, percent humidity, wind direction, wind velocity, barometric pressure, and the forecast. See the structure 'weatherRec' for the record format.

File pointer information is retained between calls in the UDF scratchpad area.

Data read from the input data set is returned by this function as a DB2 table with the following structure:

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>CITY</td>
<td>VARCHAR(30)</td>
</tr>
<tr>
<td>TEMP_IN_F</td>
<td>INTEGER</td>
</tr>
<tr>
<td>HUMIDITY</td>
<td>INTEGER</td>
</tr>
<tr>
<td>WIND</td>
<td>VARCHAR(5)</td>
</tr>
<tr>
<td>WIND_VELOCITY</td>
<td>INTEGER</td>
</tr>
<tr>
<td>BAROMETER</td>
<td>FLOAT</td>
</tr>
<tr>
<td>FORECAST</td>
<td>VARCHAR(25)</td>
</tr>
</tbody>
</table>

Clients invoking this function can use standard SQL syntax to create a desired result set.

Example invocation:

```sql
EXEC SQL DECLARE WEATHER_CURSOR FOR CURSOR FOR SELECT CITY, TEMPERATURE_IN_FAHRENHEIT, PERCENT_HUMIDITY, WIND_DIRECTION, WIND_VELOCITY, BAROMETRIC_PRESSURE, FORECAST FROM TABLE( DSN8.WEATHER(:hvWeatherDSN) ) WHERE CITY = 'Juneau, AK';
```

Notes:
- Dependencies: Requires IBM C/C++ for OS/390 V1R3 or higher
- Restrictions:
- Module type: C program
- Processor: IBM C/C++ for OS/390 V1R3 or higher
- Module size: See linkedit output
- Attributes: Re-entrant and re-usable
- Entry Point: DSN8DUWF
- Purpose: See Function
- Linkage: DB2SQL
- Invoked via SQL UDF call
- Input: Parameters explicitly passed to this function:
  - *weatherDSN : pointer to a char[45], null-terminated string
- Message: DSN8DUWF Error: Error closing weather data
- Return Code: SQLSTATE = 38604

- Message: DSN8DUWF Error: Error opening weather data
- info code=<y>
- <ddname>: Error code=<x>,
- Message: DSN8DUWF Error: Unable to allocate DD
- Error Exit: Return Code: SQLSTATE = 38601
- Message: DSN8DUWF Error: Unable to allocate DD
- Error code=<x>,
- info code=<y>
- Message: DSN8DUWF Error: Error opening weather data
- set
- Message: DSN8DUWF Error: Error closing weather data
- set

Output: Parameters explicitly passed by this function:
- *city : pointer to a char[31], null-terminated string to receive the name of the city.
- *temp_in_f : pointer to a long integer to receive the temperature in fahrenheit for the city.
- *humidity : pointer to a long integer to receive the percent humidity for the city.
- *wind : pointer to a char[6], null-terminated string to receive the wind direction for the city.
- *wind_velocity: pointer to a long integer to receive the wind velocity for city.
- *barometer : pointer to a double word to receive the barometric pressure for the city.
- *forecast : pointer to a char[26], null-terminated string to receive the forecast for the city.
- *niCity : pointer to a short integer to receive the null indicator variable for *city.
- *niTemp_in_f : pointer to a short integer to receive the null indicator variable for *temp_in_f.
- *niHumidity : pointer to a short integer to receive the null indicator variable for *humidity.
- *niWind : pointer to a short integer to receive the null indicator variable for *wind.
- *niWind_velocity: pointer to a short integer to receive the null indicator variable for *wind_velocity.
- *niBarometer : pointer to a short integer to receive the null indicator variable for *barometer.
- *niForecast : pointer to a short integer to receive the null indicator variable for *forecast.
- *niWeatherDSN: pointer to a short integer having source data for the weather reports.
- *fnName: pointer to a char[138], null-terminated string having the UDF family name of this function.
- *specificName: pointer to a char[129], null-terminated string having the UDF specific name of this function.
- *sqlstate : pointer to a char[06], null-terminated string to receive a diagnostic nated string to receive the SQLSTATE.
- *niForecast : pointer to a long integer to receive rection for the city.
- *niBarometer : pointer to a double word to receive the barometric pressure for the city.
- *niWeatherDSN: pointer to a short integer having source data for the weather reports.
- *fnName: pointer to a char[138], null-terminated string having the UDF family name of this function.
- *specificName: pointer to a char[129], null-terminated string having the UDF specific name of this function.
- *sqlstate : pointer to a char[06], null-terminated string to receive a diagnostic nated string to receive the SQLSTATE.
- *niForecast : pointer to a long integer to receive rection for the city.
- *niBarometer : pointer to a double word to receive the barometric pressure for the city.
- *niWeatherDSN: pointer to a short integer having source data for the weather reports.
- *fnName: pointer to a char[138], null-terminated string having the UDF family name of this function.
- *specificName: pointer to a char[129], null-terminated string having the UDF specific name of this function.
- *sqlstate : pointer to a char[06], null-terminated string to receive a diagnostic nated string to receive the SQLSTATE.
- *niForecast : pointer to a long integer to receive rection for the city.
- *niBarometer : pointer to a double word to receive the barometric pressure for the city.
- *niWeatherDSN: pointer to a short integer having source data for the weather reports.
- *fnName: pointer to a char[138], null-terminated string having the UDF family name of this function.
- *specificName: pointer to a char[129], null-terminated string having the UDF specific name of this function.
- *sqlstate : pointer to a char[06], null-terminated string to receive a diagnostic nated string to receive the SQLSTATE.
- *niForecast : pointer to a long integer to receive rection for the city.
- *niBarometer : pointer to a double word to receive the barometric pressure for the city.
- *niWeatherDSN: pointer to a short integer having source data for the weather reports.
- *fnName: pointer to a char[138], null-terminated string having the UDF family name of this function.
- *specificName: pointer to a char[129], null-terminated string having the UDF specific name of this function.
- *sqlstate : pointer to a char[06], null-terminated string to receive a diagnostic nated string to receive the SQLSTATE.
- *niForecast : pointer to a long integer to receive rection for the city.
- *niBarometer : pointer to a double word to receive the barometric pressure for the city.
- *niWeatherDSN: pointer to a short integer having source data for the weather reports.
- *fnName: pointer to a char[138], null-terminated string having the UDF family name of this function.
- *specificName: pointer to a char[129], null-terminated string having the UDF specific name of this function.
- *sqlstate : pointer to a char[06], null-terminated string to receive a diagnostic nated string to receive the SQLSTATE.
- *niForecast : pointer to a long integer to receive rection for the city.
- *niBarometer : pointer to a double word to receive the barometric pressure for the city.
- *niWeatherDSN: pointer to a short integer having source data for the weather reports.
- *fnName: pointer to a char[138], null-terminated string having the UDF family name of this function.
- *specificName: pointer to a char[129], null-terminated string having the UDF specific name of this function.
- *sqlstate : pointer to a char[06], null-terminated string to receive a diagnostic nated string to receive the SQLSTATE.
- *niForecast : pointer to a long integer to receive rection for the city.
- *niBarometer : pointer to a double word to receive the barometric pressure for the city.
- *niWeatherDSN: pointer to a short integer having source data for the weather reports.
- *fnName: pointer to a char[138], null-terminated string having the UDF family name of this function.
- *specificName: pointer to a char[129], null-terminated string having the UDF specific name of this function.
- *sqlstate : pointer to a char[06], null-terminated string to receive a diagnostic nated string to receive the SQLSTATE.
### DSN8DUWF

**Pseudocode:**

- If SQLUDF call type is SQLUDF_TF_FIRST (-2)
  - Call allocWeatherDSN to allocate the data set name passed as the argument to the weatherDSN parameter
  - Else if SQLUDF call type is SQLUDF_TF_OPEN (-1)
    - Call openWeatherDS to open the weather data set
  - Else if SQLUDF call type is SQLUDF_TF_FETCH (0)
    - Call readWeatherDS to read the next record from the weather data set
      - If EOF, set sqlstate to 02000 to signal end of cursor to client
      - Else call buildReturnRow to populate the UDF table function output parameters with data from the input record
    - Else if SQLUDF call type is SQLUDF_TF_CLOSE (1)
      - Call closeWeatherDS to close the weather data set
    - Else (SQLUDF call type is SQLUDF_TF_FINAL (2))
      - Call freeWeatherDS to deallocate the weather data set
  * End DSN8DUWF

- AllocateWeatherDS:
  - If the data set name passed in as the argument to the input parameter weatherDSN is for a partitioned data set, extract the member name
  - Use dynalloc to dynamically allocate the data set (and member, if applicable)
  - If allocation error occurs, issue sqlstate 38601 and a diagnostic message
  * End allocateWeatherDS

- OpenWeatherDS:
  - Open the weather data set
  - If the data cannot be opened, issue sqlstate 38602 and a diagnostic message
  * End openWeatherDS

- ReadWeatherDS:
  - Read the next record from the data set
    - This implicitly updates the file pointer variable, which is maintained in the UDF scratchpad area
    - If the data set cannot be read, issue sqlstate 38603 and a diagnostic message
  * End readWeatherDS

- BuildReturnRow:
  - Extract weather data fields from the weather data set
  - Perform appropriate data type conversions
  - Copy the (converted) data to the appropriate output parameters
  * End buildReturnRow

- CloseWeatherDS:
  - Close the weather data set
    - If the data cannot be closed, issue sqlstate 38604 and a diagnostic message
  * End closeWeatherDS

- FreeWeatherDS:
  - Use dynfree to dynamically deallocate the weather data set
    - If deallocation error occurs, issue sqlstate 38605 and a diagnostic message
  * End FreeWeatherDS

**Return Code:** SQLSTATE = 38605

**Message:** DSN8DUWF Error: FREE failed for DDNAME <<>.<

**Error code<y>, info code=** 00014900

**<z>** 00014700

**00014800**

**External References:**

- Routines/Services: dyninit: IBM C/C++, dynit.h
  - Initializes control block for dynamic file allocation
  - Dynalloc: IBM C/C++, dynit.h
  - Dynamic file allocation
  - Dynfree: IBM C/C++, dynit.h
  - Dynamic file deallocation
- Data areas: None
  - Control blocks: __dyn_t: IBM C/C++, dynint.h
  - For dynamic file allocation

**Pseudocode:**

- AllocateWeatherDS:
  - If SQLUDF call type is SQLUDF_TF_FIRST (-2)
    - Call allocWeatherDSN to allocate the data set name passed as the argument to the weatherDSN parameter
  - Else if SQLUDF call type is SQLUDF_TF_OPEN (-1)
    - Call openWeatherDS to open the weather data set
  - Else if SQLUDF call type is SQLUDF_TF_FETCH (0)
    - Call readWeatherDS to read the next record from the weather data set
      - If EOF, set sqlstate to 02000 to signal end of cursor to client
    - Else call buildReturnRow to populate the UDF table function output parameters with data from the input record
  - Else if SQLUDF call type is SQLUDF_TF_CLOSE (1)
    - Call closeWeatherDS to close the weather data set
  - Else (SQLUDF call type is SQLUDF_TF_FINAL (2))
    - Call freeWeatherDS to deallocate the weather data set
  * End DSN8DUWF

- OpenWeatherDS:
  - Open the weather data set
    - If the data cannot be opened, issue sqlstate 38602 and a diagnostic message
  * End openWeatherDS

- ReadWeatherDS:
  - Read the next record from the data set
    - This implicitly updates the file pointer variable, which is maintained in the UDF scratchpad area
    - If the data set cannot be read, issue sqlstate 38603 and a diagnostic message
  * End readWeatherDS

- BuildReturnRow:
  - Extract weather data fields from the weather data set
  - Perform appropriate data type conversions
  - Copy the (converted) data to the appropriate output parameters
  * End buildReturnRow

- CloseWeatherDS:
  - Close the weather data set
    - If the data cannot be closed, issue sqlstate 38604 and a diagnostic message
  * End closeWeatherDS

- FreeWeatherDS:
  - Use dynfree to dynamically deallocate the weather data set
    - If deallocation error occurs, issue sqlstate 38605 and a diagnostic message
  * End FreeWeatherDS

**External References:**

- Routines/Services: dyninit: IBM C/C++, dynit.h
  - Initializes control block for dynamic file allocation
  - Dynalloc: IBM C/C++, dynit.h
  - Dynamic file allocation
  - Dynfree: IBM C/C++, dynit.h
  - Dynamic file deallocation
- Data areas: None
  - Control blocks: __dyn_t: IBM C/C++, dynint.h
  - For dynamic file allocation
Chapter 12. Sample data and applications supplied with Db2 for z/OS 1241
void openWeatherDS /* Opens weather data set */
( struct scr *scratchptr, /* in: ptr to scratch pad */
  char *sqlstate, /* out: sqlstate */
  char *msgtext /* out: diag message text */
);

void readWeatherDS /* Reads from weather data set */
( struct scr *scratchptr, /* in: ptr to scratch pad */
  char *sqlstate, /* out: sqlstate */
  char *msgtext /* out: diag message text */
);

void buildReturnRow /* Builds function return row */
( char *city, /* out: name of city */
  long int *temp_in_f, /* out: temp in fahrenheit */
  long int *humidity, /* out: relative humidity */
  char *wind, /* out: wind direction */
  long int *wind_velocity, /* out: wind velocity */
  double *barometer, /* out: barometric pressure */
  char *forecast, /* out: forecast */
  short int *niCity, /* out: indic var, city name */
  short int *niTemp_in_f, /* out: indic var, temperature */
  short int *niHumidity, /* out: indic var, humidity */
  short int *niWind, /* out: indic var, wind dir */
  short int *niWind_velocity, /* out: indic var, wind veloc */
  short int *niBarometer, /* out: indic var, baro press */
  short int *niForecast, /* out: indic var, forecast */
);

void closeWeatherDS /* Closes weather data set */
( struct scr *scratchptr, /* in: ptr to scratch pad */
  void closeWeatherDS /* Closes weather data set */
);

void freeWeatherDS /* Dynam frees the weather ds */
( char *sqlstate, /* out: sqlstate */
  char *msgtext /* out: diag message text */
);

void *DSNBDUWF
( char *weatherDSN, /* in: input ds, weather data */
  char *city, /* out: name of city */
  long int *temp_in_f, /* out: temp in fahrenheit */
  long int *humidity, /* out: relative humidity */
  char *wind, /* out: wind direction */
  long int *wind_velocity, /* out: wind velocity */
  double *barometer, /* out: barometric pressure */
  char *forecast, /* out: forecast */
  short int *niWeatherDSN, /* in: indic var, weather dsn */
  short int *niCity, /* out: indic var, city name */
  short int *niTemp_in_f, /* out: indic var, temperature */
  short int *niHumidity, /* out: indic var, humidity */
  short int *niWind, /* out: indic var, wind dir */
  short int *niWind_velocity, /* out: indic var, wind veloc */
  short int *niBarometer, /* out: indic var, baro press */
  short int *niForecast, /* out: indic var, forecast */
  char *sqlstate, /* out: sqlSTATE */
  char *fnName, /* in: family name of function*/
  char *specificName, /* in: specific name of func */
  struct scr *scratchptr, /* out: diagnostic message */
  long *callType /* i/o: call type parameter */
);
else if( *callType == SQLUDF_TF_OPEN )
{
    strcpy( sqlstate,"00000" );  /* Init sqlstate return var */
    *msgtext = NULLCHAR;  /* Init message text rtn var */
    moreWeatherRecs = YES;  /* EOF indicator, weather ds */
    openWeatherDS( scratchptr, sqlstate, msgtext );
}

readWeatherDS( scratchptr, sqlstate, msgtext );
if( moreWeatherRecs == NO )  /* If no more weather data */
    sprintf( sqlstate,"02000" );  /* ..signal for FINAL CALL */
else
{
    buildReturnRow( city,
                    temp_in_f,
                    humidity,
                    wind,
                    wind_velocity,
                    barometer,
                    forecast,
                    nCity,
                    nTemp_in_f,
                    nHumidity,
                    nwind,
                    nWind_velocity,
                    nBarometer,
                    nForecast );
}

void allocWeatherDS
{
    char  *weatherDSN,  /* in: name of weather ds */
    *sqlstate,  /* out: sqlstate */
    *msgtext;  /* out: diag message text */
}

_dyn_t ip;  /* pointer to control block */
char   DName[45];  /* recv's copy of weather dsn */
char   *tokPtr;  /* string ptr for token parser*/

dyninit( &ip );  /* Initialize control block */
ip->__dsname = "WEATHRIN";  /* Specify DDNAME of WEATHRIN */
/*******************************************************************/
/* Use the strtok func to separate the PDS member name, if any, */
/* from the data set name */
/*******************************************************************/
strncpy( DName,weatherDSN );/* Get workcopy of weather dsn*/
tokPtr = strtok( DName,"(" );/* Parse for open parenthesis */
 if( tokPtr == NULL )/* If none found then */
    ip->__dsname = DName; /* ...data set is not a PDS */
else
    ip->__dsname = tokPtr;  /* ...token is name of a PDS */
tokPtr = strtok( NULL,")" );/* ...parse for close paren */
ip->__member = tokPtr;  /* ...token is name of member */
short int *niBarometer, /* out: indic var, baro press */
short int *niWind_velocity, /* out: indic var, wind veloc */
short int *niWind, /* out: indic var, wind dir */
short int *niTemp_in_f, /* out: indic var, temperature */
short int *niCity, /* out: indic var, city name */
char *forecast, /* out: forecast */
double *barometer, /* out: barometric pressure */
long int *wind_velocity, /* out: wind velocity */
long int *humidity, /* out: relative humidity */
long int *temp_in_f, /* out: indic var, temp in fahrenheit */
short int *niHumidity, /* out: indic var, humidity */
short int *niBarometer, /* out: indic var, baro press */
short int *niForecast, /* out: indic var, forecast */
short int *niCity, /* out: indic var, city name */
short int *niTemp_in_f, /* out: indic var, temperature */
short int *niHumidity, /* out: indic var, humidity */
short int *niBarometer, /* out: indic var, baro press */
short int *niForecast, /* out: indic var, forecast */
short int *niCity, /* out: indic var, city name */
short int *niTemp_in_f, /* out: indic var, temperature */
short int *niHumidity, /* out: indic var, humidity */
short int *niBarometer, /* out: indic var, baro press */
short int *niForecast, /* out: indic var, forecast */
short int *niCity, /* out: indic var, city name */
short int *niTemp_in_f, /* out: indic var, temperature */
short int *niHumidity, /* out: indic var, humidity */
short int *niBarometer, /* out: indic var, baro press */
short int *niForecast, /* out: indic var, forecast */
short int *niCity, /* out: indic var, city name */
short int *niTemp_in_f, /* out: indic var, temperature */
short int *niHumidity, /* out: indic var, humidity */
short int *niBarometer, /* out: indic var, baro press */
short int *niForecast, /* out: indic var, forecast */
void closeWeatherDS

    { if( fclose(scratchptr->WEATHRin) != 0 )
        {
            strcpy( msgtext,"Error closing weather data set ");
            strcpy( sqlstate,"38604" );

        }
    }
/* scratchpad area. Closes the weather data set and resets the file pointer in the */
/* scratchpad area. */
```c
void freeWeatherDS
    (char *sqlstate,        /* out: sqlstate      */
    char *msgtext          /* out: diag message text */
)
{
    __dyn_t free_ip;       /* pointer to control block */

    dyninit(&free_ip);     /* Initialize control block */

    free_ip.__ddname = "WEATHRIN"; /* Set DD name of weather ds */

    if (dynfree(&free_ip) != 0 )
    {
        sprintf( msgtext,"FREE failed for DDNAME %s. ",
                 free_ip.__errcode, free_ip.__infocode);
        strcpy( sqlstate,"38605");
    }
} /* end freeWeatherDS */
```

Related reference

“Sample applications in TSO” on page 1027
A set of Db2 sample applications run in the TSO environment.

**DSN8EUDN**

Returns the day of the week (Monday through Sunday) on which a given date in ISO format (YYYY-MM-DD) falls.

```plaintext
.existsSync("FREE failed for DDNAME %s.
"                 free_ip.__errcode,                 free_ip.__infocode);
    strcpy( sqlstate,"38605");
}
```

/* Sample applications in TSO */
A set of Db2 sample applications run in the TSO environment.

**DSN8EUDN**

Returns the day of the week (Monday through Sunday) on which a given date in ISO format (YYYY-MM-DD) falls.

```c
/******************
* Module name = DSN8EUDN (DB2 sample program) *
* * DESCRIPTIVE NAME = Query day of the week (UDF) *
* * LICENSED MATERIALS - PROPERTY OF IBM *
* 5675-DB2 *
* (C) COPYRIGHT 2000 IBM CORP. ALL RIGHTS RESERVED. *
* STATUS = VERSION 7 *
* Function: Returns the day of the week (Monday through Sunday) on *
* which a given date in ISO format (YYYY-MM-DD) falls. *
* Example invocation: EXEC SQL SET :dayname = DAYNAME( "2000-01-29" ); *
* => dayname = Tuesday *
* Notes: *
* Dependencies: Requires IBM C/C++ for OS/390 V1R3 or higher *
* Restrictions: Assumes the Gregorian calendar was adopted in *
* September, 1752. Code modifications are required to handle a different adoption date. *
* Module type: C++ program *
* Processor: IBM C/C++ for OS/390 V1R3 or higher *
* Module size: See linkedit output *
* Attributes: Re-entrant and re-usable *
* Entry Point: DSN8EUDN *
* Purpose: See Function *
* Linkage: DB2SQL *
* Invoked via SQL UDF call *
* Input: Parameters explicitly passed to this function: *
* - *ISOdateIn  : pointer to a char[11], null-terminated string having a date in ISO *
*   format. *
* - *niISOdateIn : pointer to a short integer having the null indicator variable for *
*                 *ISOdateIn. *
* - *InName     : pointer to a char[138], null-terminated string having the UDF family *
*                 name of this function. *
```
- *specificName: pointer to a char[129], null-terminated string having the UDF specific name of this function.

Output: Parameters explicitly passed by this function:
- *dayNameOut: pointer to a char[10], null-terminated string to receive the dayname for ISOdateIn.
- *niDayNameOut: pointer to a short integer to receive the null indicator variable for *dayNameOut.
- *sqlstate: pointer to a char[06], null-terminated string to receive the SQLSTATE.
- *message: pointer to a char[70], null-terminated string to receive a diagnostic message if one is generated by this function.

Normal Exit: Return Code: SQLSTATE = 00000
- Message: none

Error Exit: Return Code: SQLSTATE = 38601
- Message: DSN8EUDN Error: No date entered
- Return Code: SQLSTATE = 38602
- Message: DSN8EUDN Error: Input date not valid or not in ISO format

External References:
- Routines/Services:
  - strftime: Formatted time conversion routine
    - from IBM C/C++ for z/OS run-time library
  - strptime: Date and time conversion routine
    - from IBM C/C++ for z/OS run-time library
- Data areas: None
- Control blocks: None

Pseudocode:
DSN8EUDN:
- Verify that a date was passed in:
  - if *ISOdateIn blank or *niISOdateIn is not 0, no date passed:
    - issue SQLSTATE 38601 and a diagnostic message.
  - Use strptime to validate the entry
    - if *ISOdateIn is not a valid ISO date:
      - issue SQLSTATE 38602 and a diagnostic message
    - Parse out the year, month, and day
  - Compute the weekday number (0=Sunday, ..., 6=Saturday)
  - Use strftime and strftime to convert the day number to the full weekday name of the current locale.
End DSN8EUDN

Change log:
2004-02-25: Rewritten due to demise of IBM Open Class library

*********************************************************************/
extern "C" void DSN8EUDN /* Establish linkage */
(char        *ISOdateIn,             /* in: date to look up */
char        *dayNameOut,            /* out: ISOdateIn's day name */
short int   *niISOdateIn,           /* in: indic var, ISOdateIn */
short int   *niDayNameOut,          /* out: indic var, dayNameOut */
char        *sqlstate,              /* out: SQLSTATE */
char        *fnName,                /* in: family name of function*/
char        *specificName,          /* in: specific name of func */
char        *message                /* out: diagnostic message */);
#pragma linkage(DSN8EUDN,fetchable) /* Establish linkage */

/******************** C library definitions ********************/
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

/******************** Equates *******************/
#define NULLCHAR     '\0'       /* Null character */
#define MATCH        0           /* Comparison status: Equal */
#define NOT_OK        0           /* Run status indicator: Error*/
#define OK           1           /* Run status indicator: Good */

/******************** DSN8EUDN functions *******************/
void DSN8EUDN                         /* main routine               */
(char        *ISOdateIn,             /* in: date to look up        */
char        *dayNameOut,            /* out: ISOdateIn's day name  */
short int   *niISOdateIn,           /* in: indic var, ISOdateIn   */
short int   *niDayNameOut,          /* out: indic var, dayNameOut */
char        *sqlstate,              /* out: SQLSTATE              */
char        *fnName,                /* in: family name of function*/
char        *specificName,          /* in: specific name of funct */
char        *message)               /* out: diagnostic message    */

/***************************************************************************/
* Returns the weekday name of the date in ISOdateIn.                     *
* Assumptions:                                                           *
* - *ISOdateIn        points to a char[11], null-terminated string       *
* - *dayNameOut       points to a char[10], null-terminated string       *
* - *niISOdateIn      points to a short integer                         *
* - *niDayNameOut     points to a short integer                         *
* - *sqlstate         points to a char[06], null-terminated string       *
* - *fnName           points to a char[138], null-terminated string      *
* - *specificName     points to a char[129], null-terminated string      *
* - *message          points to a char[70], null-terminated string       *
***************************************************************************/
{
  short int   status = OK;            /* DSN8EUDN run status        */
  struct tm   tmbuff;                 /* buffer for time.h tm struct*/
  char        *rc;                    /* gets strptime return code*/
  char        *tokPtr;                /* string ptr for token parser*/
  char        workStr[11];            /* work copy of ISOdateIn parm*/
  int         yearInt;                /* numeric copy of 4-digit yr  */
  char        yearStr[05];            /* string copy of 4-digit year*/
  int         monthInt;               /* numeric copy of month no.   */
  char        monthStr[03];           /* string copy of month no.    */
  int         dayInt;                 /* numeric copy of day no.     */
  char        dayStr[03];             /* string copy of day no.      */
  int         weekDayInt;             /* week day no (0=Sun...6=Sat)*/
  char        weekDayStr[02];         /* string copy of week day no.*/
  char        isoFormat = "%Y-%m-%d";  /* format of isoDate:         */
  char        weekDayFormat = "%w";   /* format of weekday:         */
  char        weekDayLongNameFormat = "%A";     /* format of weekday long name*/

  if( *niISOdateIn != 0 || ( strlen( ISOdateIn ) == 0 ) )
    { status = NOT_OK;
      strcpy( message, "DSN8EUDN Error: No date entered" );
      strcpy( sqlstate, "38601" );
    }

  if( status == OK )
    { rc = strptime( ISOdateIn,isoFormat,&tmbuff );
      if( rc == NULL )                /* Unable to convert ISOdateIn*/
        { status = NOT_OK;
          strcpy( message, "DSN8EUDN Error: Input date not valid "
                  "or not in ISO format" );
          strcpy( sqlstate, "38602" );
        }
    }
}
* Parse the 4-digit year, the month no., and day no. from ISOdateIn*
******************************************************************************************
if( status == OK )
{
  strcpy( workStr,ISOdateIn );
  tokPtr  = strtok( workStr,"-" );
  strcpy( yearStr,tokPtr );
  yearInt  = atoi( yearStr );
  tokPtr  = strtok( NULL,"-" );
  strcpy( monthStr,tokPtr );
  monthInt = atoi( monthStr );
  tokPtr  = strtok( NULL,"-" );
  strcpy( dayStr,tokPtr );
  dayInt   = atoi( dayStr );
}

/******************************************************************************************
* Get the weekday name of ISOdateIn
******************************************************************************************
if( status == OK )
{
 ="/*******************************
* Leap year allowance: Shift Jan and Feb to end of prev year
="/*******************************
if( monthInt < 3 )
{ monthInt += 12;
  yearInt--;
}

="/*******************************
* Calculate weekday no. with Sunday basis
="/*******************************
weekdayInt = ((13 * monthInt) + 3) / 5 /* xform months */
+ dayInt /* + days */
+ yearInt /* + years */
+ yearInt / 4 /* + leapyear/4 */
- yearInt / 100 /* - leapyear/100 */
+ yearInt / 400 /* + leapyear/400 */
+ 1 /* + Sunday basis */
) % 7; /* % days per wk */

="/*******************************
* adjust for pre-gregorian calendar (September 1752)
="/*******************************
if( (yearInt < 1752) || (yearInt == 1752 && monthInt < 9) )
{ if( weekdayInt > 3 )
  weekdayInt = weekdayInt - 4;
  else
    weekdayInt = weekdayInt + 3;
}

="/*******************************
* convert day of week from numeric to string
="/*******************************
sprintf( weekdayStr,"%02d",weekdayInt );

="/*******************************
* Convert day of week from numeric string to day name
="/*******************************
rc = strftime( weekdayStr,weekdayFormat,&tmbuff );
*rc = strftime( dayNameOut,10,weekdayLongNameFormat,&tmbuff );

}

="/*******************************
* If weekday name was obtained, clear the message buffer and sql- *
* state, and unset the SQL null indicator for dayNameOut.
="/*******************************
if( status == OK )
{
  *niDayNameOut = 0;
  message[0] = NULLCHAR;
  strcpy( sqlstate,"00000" );
}

="/*******************************
* If errors occurred, clear the dayNameOut buffer and set the SQL *
* state, and unset the SQL null indicator. A diagnostic message and the SQLSTATE have been *
* set where the error was detected.
="/*******************************
else
{
Related reference
“Sample applications in TSO” on page 1027
A set of Db2 sample applications run in the TSO environment.

**DSN8EUMN**

Returns the calendar name of the month name in which a given date in ISO format (YYYY-MM-DD) falls.

```c
/* Module name = DSN8EUMN (DB2 sample program) */
/* DESCRIPTIVE NAME = Query calendar month name (UDF) */
/* LICENSED MATERIALS - PROPERTY OF IBM */
/* 5675-DB2 */
/* (C) COPYRIGHT 1998, 2000 IBM CORP. ALL RIGHTS RESERVED. */
/* STATUS = VERSION 7 */
/* */
/* * Function: Returns the calendar name of the month name in which a given date in ISO format (YYYY-MM-DD) falls. */
/* */
/* Example invocation: */
/* EXEC SQL SET :monthname = MONTHNAME( "2000-01-29" ); */
/* ==> monthname = January */
/* Notes: */
/* Dependencies: Requires IBM C/C++ for OS/390 V1R3 or higher */
/* Restrictions: */
/* */
/* Module type: C++ program */
/* Processor: IBM C/C++ for OS/390 V1R3 or higher */
/* Module size: See linkedit output */
/* Attributes: Re-entrant and re-usable */
/* */
/* Entry Point: DSN8EUMN */
/* Purpose: See Function */
/* Linkage: DB2SQL */
/* Invoked via SQL UDF call */
/* */
/* Input: Parameters explicitly passed to this function: */
/* - *ISODateIn : pointer to a char[11], null-terminated string having a date in ISO format. */
/* - *niISODateIn: pointer to a short integer having the null indicator variable for *ISODateIn. */
/* - *fnName : pointer to a char[138], null-terminated string having the UDF family name of this function. */
/* - *specificName: pointer to a char[129], null-terminated string having the UDF specific name of this function. */
/* */
/* Output: Parameters explicitly passed by this function: */
/* - *monthNameOut: pointer to a char[10], null-terminated string to receive the month name for ISODateIn. */
/* - *niMonthNameOut: pointer to a short integer to receive the null indicator variable for *monthNameOut. */
/* - *sqlstate : pointer to a char[06], null-terminated string to receive the SQLSTATE. */
/* - *message : pointer to a char[70], null-terminated string to receive a diagnostic message if one is generated by this function. */
```
Normal Exit: Return Code: SQLSTATE = 00000
- Message: none

Error Exit: Return Code: SQLSTATE = 38601
- Message: DSN8EUMN Error: No date entered
Return Code: SQLSTATE = 38602
- Message: DSN8EUMN Error: Input date not valid
  or not in ISO format

External References:
- Routines/Services:
  - strftime: Formatted time conversion routine
  - from IBM C/C++ for z/OS run-time library
  - strptime: Date and time conversion routine
  - from IBM C/C++ for z/OS run-time library
- Data areas: None
- Control blocks: None

Pseudocode:
DSN8EUMN:
- Verify that a date was passed in:
  - if *ISOdateIn blank or niISOdateIn is not 0, no date passed:
  - issue SQLSTATE 38601 and a diagnostic message.
- Use strftime to validate the entry and convert the date to tm:
  - if *ISOdateIn is not a valid ISO date:
  - issue SQLSTATE 38602 and a diagnostic message
- Use strftime to get the full monthname for the locale
End DSN8EUMN

Change log:
2004-02-25: Rewritten due to demise of IBM Open Class library

extern "C" void DSN8EUMN              /* Establish linkage          */
(char        *ISOdateIn,             /* in: date to look up        */
 char        *monthNameOut,          /* out: ISOdateIn's month name*/
 short int   *niISOdateIn,           /* in: indic var, ISOdateIn   */
 short int   *niMonthNameOut,        /* out: indic var,monthNameOut*/
 char        *sqlstate,              /* out: SQLSTATE              */
 char        *fnName,                /* in: family name of function*/
 char        *specificName,          /* in: specific name of func   */
 char        *message                /* out: diagnostic message    */
);
+ * fnName           points to a char[138], null-terminated string  *
+ * specificName     points to a char[129], null-terminated string  *
+ * message          points to a char[70], null-terminated string   *
+ *********************************************************************/
{

  /**************************************************************************
  */  
  short int   status = OK;            /* DSN8EUMN run status        */
  struct tm   tmbuff;                 /* buffer for time.h tm struct*/
  char        *rc;                    /* gets strftime return code*/
  char        *isoFormat              /* format of isoDate:         */
            = "%Y-%m-%d";           /* %Y = YYYY, %m = MM, %d = DD*/
  char        *fullMonthName          /* format of fullMonthName    */
            = "%B";                 /* %B = full month name       */
  //**********************************
  // Verify that something has been passed in
  //**********************************
  if( *niISOdateIn != 0 || ( strlen( ISOdateIn ) == 0 ) )
  {
    status = NOT_OK;
    strcpy( message, "DSN8EUMN Error: No date entered" );
    strcpy( sqlstate, "38601" );
  }
  //**********************************
  // Convert ISOdateIn to C tm format
  //**********************************
  if( status == OK )
  { rc = strptime( ISOdateIn,isoFormat,&tmbuff );
    if( rc == NULL ) /* Unable to convert ISOdateIn*/
    {
      status = NOT_OK;
      strcpy( message, "DSN8EUMN Error: Input date not valid 
      "or not in ISO format" );
      strcpy( sqlstate, "38602" );
    }
  }
  //**********************************
  // Convert the date from C tm format to the locale's full monthname
  //**********************************
  if( status == OK )
  { *rc = strftime( monthNameOut,10,fullMonthName,&tmbuff );
  }
  //**********************************
  // If month name was obtained, clear the message buffer and sql-
  // state, and unset the SQL null indicator for monthNameOut.
  //**********************************
  if( status == OK )
  { *niMonthNameOut = 0;
    message[0] = NULLCHAR;
    strcpy( sqlstate,"00000" );
  }
  //**********************************
  // If errors occurred, clear the monthNameOut buffer and set the SQL*
  // NULL indicator. A diagnostic message and the SQLSTATE have been
  // set where the error was detected.
  //**********************************
  else
  { monthNameOut[0] = NULLCHAR;
    *niMonthNameOut = -1;
  }
  return;
}  /* end DSN8EUMN */

Related reference
“Sample applications in TSO” on page 1027
A set of Db2 sample applications run in the TSO environment.

**DSN8DLPL**

Populates the PSEG_PHOTO (500K BLOB) and BMP_PHOTO (100K BLOB) columns of the EMP_PHOTO_RESUME sample table with data read from sequential data sets.

```plaintext
/***************************************************************************/
/* Module name = DSN8DLPL (DB2 sample program) */
/* */
/* DESCRIPTIVE NAME = Populate LOB columns that exceed 32K with data */
/* read from sequential data sets. */
/* */
/* LICENSED MATERIALS - PROPERTY OF IBM */
/* 5655-DB2 */
/* (C) COPYRIGHT 1997 IBM CORP. ALL RIGHTS RESERVED. */
/* */
/* STATUS = VERSION 6 */
/* */
/* Function: Populates the PSEG_PHOTO (500K BLOB) and BMP_PHOTO (100K */
/* BLOB) columns of the EMP_PHOTO_RESUME sample table with */
/* data read from sequential data sets. */
/* */
/* LOB locators are used to avoid having to contain all the */
/* data in the application's storage. */
/* */
/* Notes: */
/* */
/* Dependencies: Requires IBM C/C++ for OS/390 V1R3 or higher */
/* */
/* Restrictions: */
/* */
/* Module type: C program */
/* */
/* Processor: IBM C/C++ for OS/390 V1R3 or subsequent release */
/* */
/* Module size: See linkedit output */
/* Attributes: Re-entrant and re-usable */
/* */
/* Entry Point: CEESTART (Language Environment entry point) */
/* */
/* Purpose: See Function */
/* */
/* Linkage: Standard MVS program invocation, no parameters */
/* */
/* Input: Symbolic label/name = PSEGINnn, where 00 <= nn <= 99 */
/* Description = PSEG photo image data */
/* */
/* Symbolic label/name = BMPINnn, where 00 <= nn <= 99 */
/* Description = BMP photo image data */
/* */
/* Output: Symbolic label/name = SYSPRINT */
/* Description = Report and messages */
/* */
/* Normal Exit: Return Code = 0000 */
/* - Message: none */
/* */
/* Error Exit: Return Code = 0008 */
/* - Message: *** ERROR: DSN8DLPL DB2 Sample Program */
/* Unable to open BMPINnn DD data set. Processing terminated. */
/* */
/* - Message: *** ERROR: DSN8DLPL DB2 Sample Program */
/* Unexpected SQLCODE encountered at location xxx */
/* Error detailed below */
/* Processing terminated */
/* (DSNTIAR-formatted message follows). */
/* */
/* External References: */
/* - Routines/Services: DSNTIAR */
/* - Data areas : DSNTIAR error_message */
/* - Control blocks : None */
/* */
/* Pseudocode: */
/* DSN8DLPL: */
/* - Set DD counter (nn) to 00 */
/* - Do while more PSEGINnn DD’s to process */
/* - Call openPSEGfile to open the data set associated with */
/* DD PSEGINnn */
```
- Call getPSEGrec to read the first record of the data set
  - Extract the employee serial from this record
- Call openBMPfile to open the data set associated with DD BMPINnn
- Call getBMPrec to read the first record of the data set
- Call primeBLOBCols to:
  - UPDATE the PSEG_PHOTO and BMP_PHOTO columns of the employee's row in the EMP_PHOTO_RESUME table with the contents of these first records
  - SELECT the PSEG_PHOTO and BMP_PHOTO columns back into BLOB locators
- Call getPSEGrec to read the next record from the PSEGINnn DD
- Do while not end of file for the PSEGINnn DD
  - Call buildPSEGcol to append the current PSEGINnn record to the PSEG BLOB locator
  - Call getPSEGrec to read the next record from PSEGINnn
  - Call getBMPrec to read the next record from the BMPINnn DD
  - Do while not end of file for the BMPINnn DD
  - Call buildBMPcol to append the current BMPINnn record to the BMP BLOB locator
  - Call getBMPrec to read the next record from BMPINnn
  - Call updateBLOBCols to apply the BLOB locators to the PSEG_PHOTO and BMP_PHOTO columns of the employee's row in the EMP_PHOTO_RESUME table
  - If all went well, call rollbackWorkUnit to commit the changes
  - Else call rollbackWorkUnit to roll back the changes
  - Print a status line
  - Close the PSEGINnn and BMPINnn DD's
  - Increment DD counter (nn) by 1.
  - If an SQL error occurs, invoke the sql_error routine to generate and display message text
- End DSN8DLPL

openPSEGfile:
  - Open the data set associated with the PSEGINnn DD
  - If the open fails, set validDD to false
- End openPSEGfile

getPSEGrec:
  - Read a record from the data set associated with the PSEGINnn DD
  - If end of file, set morePSEGrecs to false
- End getPSEGrec

openBMPfile:
  - Open the data set associated with the BMPINnn DD
  - End openBMPfile

getBMPrec:
  - Read a record from the data set associated with the BMPINnn DD
  - If end of file, set moreBMPrecs to false
- End getBMPrec

primeBLOBCols:
  - Extract the employee serial from bytes 10-15 of the PSEG buffer.
  - UPDATE the PSEG_PHOTO and BMP_PHOTO columns for the employee's row in the EMP_PHOTO_RESUME table from the PSEG and BMP records
  - SELECT the PSEG_PHOTO and BMP_PHOTO columns for the employee into LOB locators biPSEG1 and biBMP1
- End primeBLOBCols

buildPSEGcol:
  - Append the contents of the PSEG input record to the PSEG BLOB
    - locator biPSEG1 and assign to BLOB locator biPSEG2
    - free BLOB locator biPSEG1
    - set BLOB locator biPSEG1 from BLOB locator biPSEG2
    - free BLOB locator biPSEG2
  - End buildPSEGcol

buildBMPcol:
  - Append the contents of the BMP input record to the BMP BLOB
    - locator biBMP1 and assign to BLOB locator biBMP2
    - free BLOB locator biBMP1
    - set BLOB locator biBMP1 from BLOB locator biBMP2
    - free BLOB locator biBMP2
  - End buildBMPcol

updateBLOBCols:
  - UPDATE the PSEG_PHOTO and BMP_PHOTO columns for the employee's row in the EMP_PHOTO_RESUME table from the PSEG and BMP BLOB
    - locators BMP biPSEG1 and biBMP1
  - End updateBLOBCols
**commitWorkUnit:**
- commit the changes
* End commitWorkUnit
*
**rollbackWorkUnit:**
- roll back the changes
* End rollbackWorkUnit
*
**sql_error:**
- call DSNTIAR to format the unexpected SQLCODE.
* End sql_error
*
********************************************************************************
/****** C library definitions *************/
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

/**************************** Equates ***************************/
#define NO 0        /* False */
#define YES 1       /* True */
#define NOT_OK 0    /* Run status indicator: Error */
#define OK 1        /* Run status indicator: Good */
#define TIAR_DIM 10 /* Max no. of DSNTIAR msgs */
#define TIAR_LEN 80 /* Length of DSNTIAR messages */

/**************************** Files ******************************/
FILE *BMPin;                 /* pointer to BMP input file */
FILE *PSEGin;                /* pointer to PSEG input file */

/**************************** Global Storage ***************************/
int status = OK;      /* run status flag */
char PSEGinDD[12];    /* PSEGin DD template */
char BMPinDD[12];     /* BMPin DD template */
short int DDcounter = 0; /* DD allocation counter */
char DNum[2];         /* DD number string template */
short int validDD = YES; /* unprocessed DD indicator */
short int morePSEGrecs = YES; /* eof indicator for PSEGINnn */
short int moreBMPrecs = YES; /* eof indicator for BMPINnn */
short int PSEGblklLen = 0; /* length of PSEG input block */
short int PSEGblkpPos = 0; /* offset in PSEG input block */
short int PSEGrrecPos = 0; /* offset in PSEG input record */
long int PSEGcolLen = 0;  /* length of PSEG column data */
short int BMPblklLen = 0; /* length of BMP input block */
short int BMPblkpPos = 0; /* offset in BMP input block */
short int BMPrecPos = 0;  /* offset in BMP input record */
long int BMPcolLen = 0;   /* length of BMP column data */
int byteIn;           /* current incoming byte */

/**************************** DB2 SQL Communication Area ****************************/
EXEC SQL INCLUDE SQLCA;

/**************************** DB2 Tables *****************************/
EXEC SQL DECLARE  EMP_PHOTO_RESUME  TABLE
  (  EMPNO  CHAR(06)     NOT NULL,
      EMP_ROWID ROWID,
      PSEG_PHOTO BLOB( 500K ),
      BMP_PHOTO BLOB( 100K ),
      RESUME  CLOB( 5K ),
  );

/**************************** DB2 Host and Null Indicator Variables *****************************/
EXEC SQL BEGIN DECLARE SECTION;
char hvEMPNO[7]; /* Host var for employee no. */
SQL TYPE IS BLOB(0K) PSEGinRec; /* Area for PSEG input record */
short int niPSEG_PHOTO = 0; /* Null ind for PSEG photo col*/

Chapter 12. Sample data and applications supplied with Db2 for z/OS 1255
SQL TYPE IS BLOB(8K) BMPinRec;       /* Area for BMP input record */
short int   niBMP_PHOTO  = 0;       /* Null ind for BMP photo col */
EXEC SQL END DECLARE SECTION;

/*********************** DB2 LOB Locator Variables ***********************
EXEC SQL BEGIN DECLARE SECTION;
SQL TYPE IS BLOB_LOCATOR blPSEG1;   /* BLOB loc for PSEG photo col*/
SQL TYPE IS BLOB_LOCATOR blPSEG2;   /* BLOB loc for PSEG photo col*/
SQL TYPE IS BLOB_LOCATOR blBMP1;    /* BLOB loc for BMP photo col */
SQL TYPE IS BLOB_LOCATOR blBMP2;    /* BLOB loc for BMP photo col */
EXEC SQL END DECLARE SECTION;

******************** DB2 Message Formatter ********************
struct    error_struct {            /* DSNTIAR message structure */
short int   error_len;
char        error_text[TIAR_DIM][TIAR_LEN];
}         error_message = {TIAR_DIM * (TIAR_LEN)};

#pragma linkage( dsntiar, OS )
extern  short int dsntiar( struct    sqlca         *sqlca,
struct    error_struct  *msg,
int                       *len );

************************************************************************
Global Functions **********************************************************
************************************************************************
int  main( void );                    /* main routine               */
void openPSEGfile( void );            /* open PSEGInnn DD file      */
void getPSEGrec( void );              /* read PSEG image file       */
void openBMPfile( void );             /* open BMPInnn DD file       */
void getBMPrec( void );               /* read BMP image file        */
void primeBLOBcols( void );           /* set PSEG and BMP BLOB locs */
void buildPSEGcol( void );            /* add to PSEG BLOB locator   */
void buildBMPcol( void );             /* add to BMP BLOB locator    */
void updateBLOBcols( void );          /* apply PSEG and BMP locs    */
void commmitWorkUnit( void );         /* commit changes             */
void rollbackWorkUnit( void );        /* roll back changes          */
void sql_error( char *locmsg );       /* generate msg for SQL error */

************************************************************************
main routine **************************************************************
************************************************************************
int main( void )
{
    /************************************************************************
    Initialization **********************************************************
    ************************************************************************/

    /************************************************************************
    Processing **************************************************************
    ************************************************************************/

    /************************************************************************
    Cycle through PSEGInnn and BMPInnn DD pairs, incrementing the DD *
    * counter, until all pairs have been processed.                      *
    ************************************************************************/
    for( DDcounter=0; validDD == YES && status == OK; DDcounter++ )
    {
        /************************************************************************
        * Fetch the PSEG data from the current PSEGInnn DD.  The first *
        * record contains the serial number of the employee associated *
        * with the photo.                                                   *
        /************************************************************************
        openPSEGfile();
        if( validDD == YES && status == OK )
            getPSEGrec();

        /************************************************************************
        * Fetch the first record from the BMPInnn DD                       *
        ************************************************************************/
if( validDD == YES && status == OK )
  openBMPfile();
if( validDD == YES && status == OK )
  getBMPrec();

/****** Init the PSEG and BMP BLOB table columns for the employee ******/
if( validDD == YES && status == OK )
  primeBLOBcols();

/****** Read the second records from the PSEG and BMP data sets ******/
if( validDD == YES && status == OK )
  getPSEGrec();
if( validDD == YES && status == OK )
  getBMPrec();

/****** Append remaining PSEG recs to PSEG photo col using BLOB loc ******/
while( morePSEGrecs == YES && validDD == YES && status == OK )
{
  buildPSEGcol();
  if( status == OK )
    getPSEGrec();
}

/****** Append remaining BMP recs to BMP photo col using BLOB locator ******/
while( moreBMPrecs == YES && validDD == YES && status == OK )
{
  buildBMPcol();
  if( status == OK )
    getBMPrec();
}

/****** Apply the data associated with the PSEG and BMP BLOB locators ******/
if( validDD == YES && status == OK )
  updateBLOBcols();

/****** If clear status, commit the work unit; otherwise, rollback ******/
if( validDD == YES )
  if( status == OK )
    commitWorkUnit();
else
  rollbackWorkUnit();

/****** Print report line ******/
if( validDD == YES && status == OK )
{
  printf( "*   LOB population statistics for employee "
      " number %s follow:\n", hvEMPNO );
  printf( "*   - PSEG photo bytes: %d\n",PSEGcolLen );
  printf( "*   - BMP photo bytes: %d\n",BMPcolLen );
  printf( "****************************************\n" );
}

/****** Close data sets for current PSEGInnn and BMPINnn DDs ******/
close(PSEGin);
close(BMPin);
} /* end for( DDcounter=0; validDD == YES & status == OK ... */
if( status == OK )
  return( 0 );
else
  return( 8 );
} /* end main */

void openPSEGfile( void )
//*****************************************************************************/
/* Opens the data set associated with the PSEGINnn DD, where "nn" is the current setting of the DD counter from the main loop. */
/* If the DD cannot be allocated, then no further data sets remain to be processed so signal end of job. */
//*****************************************************************************/
{
  /*******************************************************************
  * initialize work variables
  ************************************************************************/
  morePSEGrecs = YES;
PSEGblkPos = 0;
PSEGblkLen = 0;
  /*******************************************************************
  * form the DD name for the next PSEG data set
  ************************************************************************/
  strcpy( PSEGinDD,"DD:PSEGIN\0" ); /* init PSEGin DD template */
  sprintf( DDnum,"%02d",DDcounter ); /* convert DD cntr to string */
  strcat( PSEGinDD,DDnum ); /* form PSEGINnn DD name */
  /*******************************************************************
  * open the PSEGINnn DD data set
  ************************************************************************/
  PSEGin = fopen( PSEGinDD,"rb,recfm=u" );
  if( PSEGin == NULL ) /* if no pointer returned */
    validDD = NO; /* .. no more data sets left */
} /* end openPSEGfile */

void getPSEGrec( void )
//*****************************************************************************/
/* Called by the main routine to read the next record from the data set associated with the current PSEGINnn DD into a buffer, PSEGin-Rec. */
/* If this is the first record from the PSEGINnn DD data set, it contains the serial number of an employee in bytes 10-15 and it will be UPDATED into the PSEG_PHOTO column of that employee's row in the sample EMP_PHOTO_RESUME table. This column and row will then be SELECTed into a BLOB locator, blPSEG1, to be used for accumulating the remaining records from the current PSEGINnn DD data set to form a complete PSEG_PHOTO entry for the current employee. */
/* If this is not the first record from the PSEGINnn DD data set, it will be appended to previously read records for this data set in a DB2 data area associated with the BLOB locator, blPSEG1. */
/* When all records of the data set have been read and accumulated in the locator area, the locator will be applied to the PSEG_PHOTO column of the current employee's row in the EMP_PHOTO_RESUME table. Because the C language is not record-oriented in the sense of MVS data sets, it's necessary to treat the PSEG data set, which has a variable-blocked format, as an unformatted dataset in order to access the block descriptor word (BDW) of each input block and the record descriptor word (RDW) of each input record. */
/* Each RDW provides the number of bytes of data in its record, including 4 bytes for itself. */
/* Each BDW provides the number of bytes of data in its block, including 4 bytes for each RDW in the block and 4 bytes for itself. */
//*****************************************************************************/
{
  /*******************************************************************
  * initialize work variables
  ************************************************************************/
  PSEGrecLen = 0;
  ...
PSEGrecPos = 0;
PSEGinRec.length = 0;

/*****************************************************************************
* read the 1st byte of the record                                          *
*****************************************************************************
byteIn = getc(PSEGin);
/*****************************************************************************
* get remaining bytes of the record if not EOF                             *
*****************************************************************************
if( byteIn != EOF ) {
  /*****************************************************************************
* if at end of block, read next BDW                                       *
*****************************************************************************
  if( PSEGblkPos >= PSEGblkLen & PSEGrecPos >= PSEGrecLen ) {
    /*****************************************************************************
    * length of block = (16**2) * BDW[0]                                   *
    * ................ + (16**0) * BDW[1]                                    *
    * ................ - 4 (length of BDW)                                   *
    *****************************************************************************
    PSEGblkLen = 256 * byteIn;
    byteIn = getc(PSEGin);
    PSEGblkLen = PSEGblkLen + byteIn - 4;
    /*****************************************************************************
    * skip remainder of BDW                                               *
    *****************************************************************************
    byteIn = getc(PSEGin);
    byteIn = getc(PSEGin);
    PSEGblkPos = 0;
  } /* end if( PSEGblkPos >= PSEGblkLen & PSEGrecPos >= PSEGrecLen ) */
  /*****************************************************************************
  * read first byte of RDW                                                 *
  *****************************************************************************
  byteIn = getc(PSEGin);
} /* end if( byteIn != EOF ) */
/*****************************************************************************
* process the RDW                                                         *
*****************************************************************************
PSEGrecLen = 256 * byteIn;
byteIn = getc(PSEGin);
PSEGrecLen = PSEGrecLen + byteIn - 4;
/*****************************************************************************
* skip remainder of RDW                                                  *
*****************************************************************************
byteIn = getc(PSEGin);
byteIn = getc(PSEGin);
PSEGrecPos = 0;
/*****************************************************************************
* update position in block                                               *
*****************************************************************************
PSEGblkPos = PSEGblkPos + PSEGrecLen + 4;
} /* end getPSEGrec */

void openBMPfile( void )

Chapter 12. Sample data and applications supplied with Db2 for z/OS 1259
* Opens the data set associated with the BMPINnn DD, where "nn" is * * the current setting of the DD counter from the main loop. * * * * If the DD cannot be allocated, then an error has occurred because * * each BMPINnn DD must be paired with a PSEGINnn data set. * * *********************************************************************************/
{
/* intialize work variables */
moreBMPrecs = YES;
BMPblkPos = 0;
BMPblkLen = 0;

/* form the DD name for the next BMP data set */
strcpy( BMPinDD,"DD:BMPIN\0" );  /* init BMPin DD template */
sprintf( DDnum,"%02d",DDcounter );  /* convert DD cntr to string */
strcat( BMPinDD,DDnum );            /* form BMPINnn DD name */

/* open the current BMPINnn DD data set */
BMPin = fopen( BMPinDD,"rb,recfm=u" );
if( BMPin == NULL )
{
    printf( "************************************************
" );
    printf( "*** ERROR: DSN8DLPL DB2 Sample Program
" );
    printf( "***        Unable to open BMPIN%s DD data set
", DDnum );
    printf( "***        Processing terminated.
" );
    status = NOT_OK;
}
} /* end openBMPfile */

void getBMPrec( void )
{ /* Called by the main routine to read the next record from the data */
    set associated with the current BMPINnn DD into a buffer, BMPinRec.
    * * If this is the first record from the BMPINnn DD data set, it con-
    * tains the serial number of an employee in bytes 10-15 and it will * * be UPDATEd into the BMP_PHOTO column of that employee’s row in * * the EMP_PHOTO_RESUME table. This column and row will then * * be SELECTed into a BLOB locator, blBMP1, to be used for accumulat-
    * ing the remaining records from the current BMPINnn DD data set to * * form a complete BMP_PHOTO entry for the current employee. * * * * If this is not the first record from the BMPINnn DD data set, it * * will be appended to previously read records for this data set in * * a DB2 data area associated with the BLOB locator, blBMP1. * * * * When all records of the data set have been read and accumulated in * * the locator area, the locator will be applied to the BMP_PHOTO * * column of the current employee’s row in the EMP_PHOTO_RESUME table. * * Because the C language is not record-oriented in the sense of MVS * * data sets, it’s necessary to treat the BMP data set, which has a * * variable-blocked format, as an unformatted dataset in order to * * access the block descriptor word (BDW) of each input block and the * * record descriptor word (RDW) of each input record. * * * * Each RDW provides the number of bytes of data in its record, * * including 4 bytes for itself. * * * * Each BDW provides the number of bytes of data in its block, * * including 4 bytes for each RDW in the block and 4 bytes for * * itself. */
{
/* intialize work variables */
BMPrecLen = 0;
BMPrecPos = 0;
BMPinRec.length = 0;
/* read the 1st byte of the record */
byteIn = getc(BMPin);
/* get remaining bytes of the record if not EOF */
if( byteIn != EOF )
{
    /* if at end of block, read next BDW */
    if( BMPblkPos >= BMPblkLen )
    {
        /* length of block = (16**2) * BDW[0] */
        /* .............. + (16**0) * BDW[1] */
        /* .............. - 4 (length of BDW) */
        BMPblkLen = 256 * byteIn;
        BMPblkLen = BMPblkLen + byteIn - 4;
    }
    /* skip remainder of BDW */
    byteIn = getc(BMPin);
    byteIn = getc(BMPin);
    BMPblkPos = 0;
    /* read first byte of RDW */
    byteIn = getc(BMPin);
}
/* process the RDW */
/* length of record = (16**2) * RDW[0] */
/* .............. + (16**0) * RDW[1] */
/* .............. - 4 (length of RDW) */
BMPrecLen = 256 * byteIn;
byteIn = getc(BMPin);
BMPrecLen = BMPrecLen + byteIn - 4;
/* skip remainder of RDW */
byteIn = getc(BMPin);
byteIn = getc(BMPin);
BMPrecPos = 0;
/* update position in block */
BMPblkPos = BMPblkPos + BMPrecLen + 4;
}
/* build the BMP record according to the record length */
while( BMPrecPos < BMPrecLen && byteIn != EOF )
{
    byteIn = getc(BMPin);
    BMPinRec.data[BMPinRec.length++] = byteIn;
    BMPrecPos++;
}
/* signal end of file when applicable */
if( byteIn == EOF )
    moreBMPrecs = NO;
} /* end getBMPrec */

void primeBLOBcols( void )

*(from getPSEGrec) and the first BMP input record (from getBMPrec)*
* to the PSEG_PHOTO and BMP_PHOTO BLOB columns, respectively, and    *
* then fetch those columns using BLOB locators.                     *
*
* The PSEG BLOB locator will be used by the buildPSEGcol function    *
* to build a BLOB entity of up to 500K bytes from the remaining      *
* PSEGin records without consuming application workspace.            *
*
* The BMP BLOB locator will be used by the buildBMPcol function to  *
* build a BLOB entity of up to 500K bytes from the remaining BMPin   *
* records, again without consuming application workspace.            *
*
* When all PSEG and BMP records have been processed, the data will  *
* be applied from the BLOB locators to the EMP_PHOTO_RESUME table by *
* the updateBLOBcols function.                                      *
**********************************************************************/

{ char *empser; /* */  
  /*********************************************************************/
  /* Extract the employee number from bytes 10-15 of the PSEG record */
  /*********************************************************************/
  empser = &PSEGinRec.data[9];
  strncpy( hvEMPNO,empser,6 );
  /*********************************************************************/
  /* Initialize the BLOB columns with data from the 1st input records */
  /*********************************************************************/
  EXEC SQL UPDATE  EMP_PHOTO_RESUME
  SET  PSEG_PHOTO = :PSEGinRec,
       BMP_PHOTO  = :BMPinRec
  WHERE  EMPNO = :hvEMPNO;
  if( SQLCODE != 0 )
  {
    status = NOT_OK;
    sql_error( "primeBLOBcols @ UPDATE" );
  }
  /*********************************************************************/
  /* Select the initial BLOB data into locators */
  /*********************************************************************/
  if( status == OK )
  {
    EXEC SQL SELECT  PSEG_PHOTO, BMP_PHOTO
        INTO :blPSEG1 :niPSEG_PHOTO,
             :blBMP1  :niBMP_PHOTO
        FROM  EMP_PHOTO_RESUME
        WHERE  EMPNO = :hvEMPNO;
    if( SQLCODE != 0 )
    {
      status = NOT_OK;
      sql_error( "primeBLOBcols @ SELECT" );
    }
  }
  /*********************************************************************/
  /* Set initial lengths of PSEG_PHOTO anf BMP_PHOTO columns */
  /*********************************************************************/
  PSEGcolLen = PSEGinRec.length;
  BMPcolLen  = BMPinRec.length;
} /* end primeBLOBcols */

void buildPSEGcol( void )
/***************************************************************************/
/* Called by the main routine to build a PSEG_PHOTO column entry for */
/* the current employee. */
/* This is done by appending the current record of the PSEG input file*/
/* (from getPSEGrec) to the entity associated with blPSEG1, the BLOB  */
/* locator for the PSEG_PHOTO column. */
/* When all PSEG input records have been appended to this entity, the */
/* updateBLOBcols function will be invoked to update the PSEG_PHOTO */
/* column in the EMP_PHOTO_RESUME table from blPSEG1. */
/***************************************************************************/
{ /* Generate a new BLOB locator that contains the current input */  
  /*********************************************************************/
* record appended to the current PSEG_PHOTO locator                      *
*-------------------------------------------------------------------*/
EXEC SQL SET  :blPSEG2 = SUBSTR( :blPSEG1,1,LENGTH(:blPSEG1) )
||  :PSEGinRec;

if( SQLCODE != 0 )
{
  status = NOT_OK;
  sql_error( "buildPSEGcol @ SET LOCATOR #2" );
}

/*************************************************************/
* Regenerate the PSEG_PHOTO locator from the updated locator  *
*************************************************************/
if( status == OK )
{
  EXEC SQL FREE LOCATOR :blPSEG1;
  if( SQLCODE != 0 )
  {
    status = NOT_OK;
    sql_error( "buildPSEGcol @ FREE LOCATOR #1" );
  }
}

if( status == OK )
{
  EXEC SQL SET :blPSEG1 = :blPSEG2;
  if( SQLCODE != 0 )
  {
    status = NOT_OK;
    sql_error( "buildPSEGcol @ SET LOCATOR #1" );
  }
}

if( status == OK )
{
  EXEC SQL FREE LOCATOR :blPSEG2;
  if( SQLCODE != 0 )
  {
    status = NOT_OK;
    sql_error( "buildPSEGcol @ FREE LOCATOR #2" );
  }
}

/*************************************************************/
* Update length of PSEG_PHOTO column                          *
*************************************************************/
if( status == OK )
  PSEGcolLen = PSEGcolLen + PSEGinRec.length;
} /* end buildPSEGcol */

void buildBMPcol( void )
/*************************************************************/
* Called by the main routine to build a BMP_PHOTO column entry for  *
* the current employee.                                         *
*                                                              *
* This is done by appending the current record of the BMP input file *
* (from getBMPrec) to the entity associated with blBMP1, the BLOB   *
* locator for the BMP_PHOTO column.                             *
*                                                              *
* When all BMP input records have been appended to this entity, the *
* updateBLOBcols function will be invoked to update the BMP_PHOTO  *
* column in the EMP_PHOTO_RESUME table from blBMP1.              *
*************************************************************/
{
  /* Generate a new BLOB locator that contains the current input  *
  * record appended to the current BMP_PHOTO locator             *
  *************************************************************/
  EXEC SQL SET  :blBMP2 = SUBSTR( :blBMP1,1,LENGTH(:blBMP1) )
||  :BMPinRec;

if( SQLCODE != 0 )
{
  status = NOT_OK;
  sql_error( "buildBMPcol @ SET LOCATOR #2" );
}

/*************************************************************/
* Regenerate the BMP_PHOTO locator from the updated locator    *
*************************************************************/
*/
if( status == OK )
{
  EXEC SQL FREE LOCATOR :blBMP1;
  if( SQLCODE != 0 )
  {
    status = NOT_OK;
    sql_error( "buildBMPcol @ FREE LOCATOR #1" );
  }
}
if( status == OK )
{
  EXEC SQL SET :blBMP1 = :blBMP2;
  if( SQLCODE != 0 )
  {
    status = NOT_OK;
    sql_error( "buildBMPcol @ SET LOCATOR #1" );
  }
}
if( status == OK )
{
  EXEC SQL FREE LOCATOR :blBMP2;
  if( SQLCODE != 0 )
  {
    status = NOT_OK;
    sql_error( "buildBMPcol @ FREE LOCATOR #2" );
  }
}
/* Update length of BMP_PHOTO column */
if( status == OK )
BMPcolLen = BMPcolLen + BMPinRec.length;
} /* end buildBMPcol */

void updateBLOBcols( void )
{
  EXEC SQL UPDATE  EMP_PHOTO_RESUME
  SET  PSEG_PHOTO = :blPSEG1,
      BMP_PHOTO  = :blBMP1
  WHERE  EMPNO = :hvEMPNO;
  if( SQLCODE != 0 )
  {
    status = NOT_OK;
    sql_error( "updateBLOBcols @ UPDATE" );
  }
} /* end updateBLOBcols */

void commitWorkUnit( void )
{
  EXEC SQL COMMIT;
  if( SQLCODE != 0 )
  {
    status = NOT_OK;
    sql_error( "commitWorkUnit @ COMMIT" );
  }
} /* end commitWorkUnit */

void rollbackWorkUnit( void )
{
  EXEC SQL ROLLBACK;
  if( SQLCODE != 0 )
  {
    status = NOT_OK;
    sql_error( "rollbackWorkUnit @ ROLLBACK" );
  }
} /* end rollbackWorkUnit */
which is composed of a fully-built PSEG entry and a fully-built BMP entry for the current employee.

```sql
EXEC SQL ROLLBACK;
if( SQLCODE != 0 )
{
    status = NOT_OK;
    sql_error( "RollbackWorkUnit @ ROLLBACK" );
}
/* end rollbackWorkUnit */

void sql_error( char *locmsg )
/{
    short int   rc;                     /* DSNTIAR Return code */
    int         j,k;                    /* Loop control */
    static int  lrecl = TIAR_LEN;       /* Width of message lines */

    /*****************************************************************************/
    * print the location message                                           *
    ****************************************************************************/
    printf( "*****************************************************\n" );
    printf( *** ERROR: DSN8DLPL DB2 Sample Program\n" );
    printf( *** Unexpected SQLCODE encountered at location\n" );
    printf( *** Error detailed below\n" );
    printf( *** Processing terminated\n" );
    printf( "*****************************************************\n" );

    /*****************************************************************************/
    * format and print the SQL message                                     *
    ****************************************************************************/
    rc = dsntiar( &sqlca, &error_message, &lrecl );
    if( rc == 0 )
    {
        for( j=0; j<TIAR_DIM; j++ )
        {
            for( k=0; k<TIAR_LEN; k++ )
                putchar(error_message.error_text[j][k] );
        }
    }
    else
    {
        printf( " *** ERROR: DSNTIAR could not format the message\n" );
        printf( " *** SQLCODE is %d\n",SQLCODE );
        printf( " *** SQLERRM is \n" );
        for( j=0; j<sqlca.sqlerrml; j++ )
            printf( "%c", sqlca.sqlerrmc[j] );
        printf( \n" );
    }
} /* end sql_error */
```

Related reference
“Sample applications in TSO” on page 1027
A set of Db2 sample applications run in the TSO environment.

**DSN8DLRV**

Prompts the user to choose an employee, then retrieves the resume data for that employee from the RESUME (CLOB) column of the EMP_PHOTO_RESUME table into a CLOB locator, uses LOB locator-handling functions to locate and break out data elements, and puts them in fields for display by ISPF.
* STATUS = VERSION 7

* Function: Prompts the user to choose an employee, then retrieves
  the resume data for that employee from the RESUME (CLOB) column of the
  EMP_PHOTO_RESUME table into a CLOB locator, uses LOB locator-handling
  functions to locate and break out data elements, and puts them in fields for display
  by ISPF.

* Notes:
  * Dependencies: Requires IBM C/C++ for OS/390 V1R3 or higher
  * Restrictions:
  * Module type: C program
  * Processor: IBM C/C++ for OS/390 V1R3 or subsequent release
  * Module size: See linked output
  * Attributes: Re-entrant and re-usable
  * Entry Point: CEESTART (Language Environment entry point)
  * Purpose: See Function
  * Linkage: Standard MVS program invocation, no parameters
  * Normal Exit: Return Code = 0000
    - Message: none
  * Error Exit: Return Code = 0008
    - Message: *** ERROR: DSN8DLRV DB2 Sample Program
      Unexpected SQLCODE encountered at location xxx
      Error detailed below
      Processing terminated (DSNTIAR-formatted message here)

* External References:
  * Routines/Services: DSNTIAR, ISPF
  * Data areas : DSNTIAR error_message
  * Control blocks : None

* Pseudocode:
  * DSN8DLRV:
    - Call initISPFVars to establish ISPF variable sharing
    - Do until the user indicates termination
      - Call clearISPFVars to reset the ISPF shared variables
      - Call getEmpNum to request an employee id
      - Call getEmpResume to retrieve the resume
      - Call formatEmpResume to populate the ISPF display panel
      - Call showEmpResume to display the resume
      - Call freeISPFVars to terminate ISPF variable sharing
  * initISPFVars:
    - Establish ISPF variable sharing
  * clearISPFVars:
    - Set ISPF vars to blank if character type or 0 if numeric
  * getEmpNum:
    - prompt user to select an employee whose resume is to be viewed
  * getEmpResume:
    - Fetch the specified employee's resume from DB2 using a CLOB locator
  * getEmplResume

1266  Db2 12 for z/OS: Application Programming and SQL Guide
formatEmplResume:
  - call getPersonalData to extract personal data from the resume
  - call getDepartmentData to extract department data
  - call getEducationData to extract education data
  - call getWorkHistoryData to extract work history data
  End formatEmplResume

showEmplResume:
  - Display the ISPF panel with the specified employee's resume
  End showEmplResume

freeISPFvars:
  - Terminate variable sharing with ISPF
  End freeISPFvars

getPersonalData:
  - Parse the employee's name, address, home telephone no., birthdate, sex, marital status, height, and weight into ISPF display variables
  End getPersonalData

getDepartmentData:
  - Parse the employee's department number, manager, job position, work telephone no., and hire date into ISPF display variables
  End getDepartmentData

getEducationData:
  - Parse the employee's degree dates, descriptions, and schools into ISPF display variables
  End getEducationData

getWorkHistoryData:
  - Parse the employee's job dates, titles, and descriptions into ISPF display variables
  End getWorkHistoryData

sql_error:
  - call DSNTIAR to format an unexpected SQLCODE
  End sql_error

Assumptions:
(1) Each employee has exactly 2 entries under "Education"
(2) Each employee has exactly 3 entries under "Work History"
(3) Each job description consists of a single sentence and that sentence ends with a period and that period is the only period in the sentence.

C Program Product Libraries
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

#include <string.h>

Chapter 12. Sample data and applications supplied with Db2 for z/OS 1267
error_message = {TIAR_DIM * (TIAR_LEN)};

#pragma linkage(dsntiar, OS)
extern short int dsntiar(
    struct sqlca *sqlca,
    struct error_struct *msg,
    int *len
);

/**************************** DB2 Tables ******************************/
EXEC SQL DECLARE EMP_PHOTO_RESUME TABLE
( EMPNO CHAR(06) NOT NULL,
  EMP_ROWID ROWID,
  PSEG_PHOTO BLOB(500K),
  BMP_PHOTO BLOB(100K),
  RESUME CLOB(5K)
);

/******************** DB2 Host and Null Indicator Variables **************/
EXEC SQL BEGIN DECLARE SECTION;
char hvEMPNO[7];    /* host var for emp ser no. */
long int begSection; /* ptr to beg of resume sec'n */
char *begField;     /* ptr to beg of fld in sec'n */
long int endSection; /* ptr to end of resume sec'n */
char *endField;     /* ptr to end of fld in sec'n */
short int niRESUME = 0; /* indic var for RESUME CLOB */
EXEC SQL END DECLARE SECTION;

/******************** DB2 LOB Locator Variables *********************/
EXEC SQL BEGIN DECLARE SECTION;
SQL TYPE IS CLOB_LOCATOR clRESUME; /* CLOB loc for RESUME column */
EXEC SQL END DECLARE SECTION;

/**************************** ISPF Linkage ***************************/
#pragma linkage(isplink, OS)

/**************************** ISPF Syntax ******************************/
char CHAR[9] = "CHAR    ";
char DISPLAY[9] = "DISPLAY ";
char VDEFINE[9] = "VDEFINE ";
char VGET[9] = "VGET    ";
char VRESET[9] = "VRESET  ";

/**************************** ISPF Shared Variables *******************/
char DBEMNAME[25];    /* employee's name */
char DBEMNUMB[7];     /* employee's serial number */
char DBEMADR1[25];    /* employee's address line 1 */
char DBEMDEPT[5];     /* employee's department */
char DBEMADR2[25];    /* employee's address line 2 */
char DBMGRNM[22];     /* employee's manager's name */
char DBEMADR3[15];    /* employee's address line 3 */
char DBEMPSON[22];    /* employee's job position */
char DBEBORN[19];     /* employee's date of birth */
char DBEMPHON[15];    /* employee's home phone no. */
char DBEMSEX[7];      /* employee's gender */
char DBEMHIRE[11];    /* employee's hire date */
char DBEMHGT[6];      /* employee's height */
char DBEMWGT[9];      /* employee's weight */
char DBEMPST[9];      /* employee's marital status */
char DBEMDDY[15];     /* date of most recent degree */
char DBEMDEDD1[35];   /* type of most recent degree */
char DBEMEDY2[5];     /* date of previous degree */
char DBEMEDD2[35];    /* type of previous degree */
char DBEMEDI[35];     /* name of most recent school */
char DBEMWHD1[17];    /* dates of 1st previous job */
char DBEMWHJ[63];     /* descr. of 1st previous job */
char DBEMWHT[63];     /* title of 2nd previous job */
char DBEMWHJD[63];    /* descr. of 2nd previous job */
char DBEMWHD2[17];    /* dates of 2nd previous job */
char DBEMWHJ2[63];    /* descr. of 3rd previous job */
char DBEMWHD3[17];    /* dates of 3rd previous job */
char DBEMWHJ3[63];    /* title of 3rd previous job */
char DBEMWHJ3[63];    /* descr. of 3rd previous job */
/************************* Global Functions *************************/
int  main( void );              /* main logic */
void initISPFvars( void );      /* establish ISPF vars */
void clearISPFvars( void );     /* blank/zero ISPF disp vars */
void getEmplNum( void );        /* prompt for employee ser no */
void getEmplResume( void );     /* get resume from database */
void formatEmplResume( void );  /* build display panel */
void getPersonData( void );     /* get personal data from res */
void getDeptData( void );       /* get dept data from resume */
void getEducationData( void );  /* get educ data from resume */
void getWorkHistoryData( void );
void showEmplResume( void );    /* display the ISPF panel */
void freeISPFvars( void );      /* drop ISPF vars */
void sql_error( char *locmsg ); /* generate SQL messages */

/*****************************************************************************/
/**************************** main routine *************************************/
int main( void )
{
  /* Establish variable sharing with ISPF */
  initISPFvars();
  /* Display employee resumes until user indicates completion */
  keepViewing = YES;
  while( keepViewing == YES )
  {
    clearISPFvars();
    /* prompt user to select employee whose resume is to be viewed */
    getEmplNum();
    if( keepViewing == YES && status == OK )
    {
      /* retrieve the employee's resume from DB2 */
      getEmplResume();
      /* if successful, format the resume on ISPF */
      if( status == OK )
        formatEmplResume();
      /* if successful, display the resume on ISPF */
      if( status == OK )
        showEmplResume();
      /* otherwise, exit this program */
      else
        keepViewing = NO;
    }
  }
  /* Terminate variable sharing with ISPF */
  freeISPFvars();
}/* end main */

void initISPFvars( void )
/* Called by the main routine. Establishes variable sharing between ISPF and this program. */
{
  ISPFC = isplink( VDEFINE, "DBEMNAME", DBEMNAME, CHAR, 24 );
  ISPFC = isplink( VDEFINE, "DBEMNUMB", DBEMNUMB, CHAR, 6 );
  ISPFC = isplink( VDEFINE, "DBEMADR1", DBEMADR1, CHAR, 24 );
  ISPFC = isplink( VDEFINE, "DBEMDEPT", DBEMDEPT, CHAR, 4 );
}
clearISPFvars( void )

void getEmplNum( void )

} /* end initISPFvars */
strcpy( hvEMPNO, DBEMNUMB );
}

void getEmplResume( void )
{  
  /*********************************************************************
   * Called by the main routine. Extracts a specified employee's
   * resume data from a CLOB column in the sample EMP_PHOTO_RESUME
   * table to a CLOB locator.  
   */

  /*******************************************************************
   * Establish a CLOB locator on the resume of the specified empno
   */
  EXEC SQL SELECT RESUME INTO :clRESUME FROM EMP_PHOTO_RESUME
  WHERE EMPNO = :hvEMPNO;

  if( SQLCODE == 100 )
  {
    status = NOT_OK;
    printf( "*************************************************
" );
    printf( "*** ERROR: DSN8DLRV DB2 Sample Program\n" );
    printf( "*** No entry in the Employee Photo/Resume\n" );
    printf( "*** table for employee with empno = %s\n", hvEMPNO );
    printf( "*** Processing terminated\n" );
    printf( "********************************************************************************\n" );
  } else if( SQLCODE == -305 )
  {
    status = NOT_OK;
    printf( "*************************************************
" );
    printf( "*** ERROR: DSN8DLRV DB2 Sample Program\n" );
    printf( "*** No resume data exists in the\n" );
    printf( "*** Employee Photo/Resume table for the\n" );
    printf( "*** employee with empno = %s\n", hvEMPNO );
    printf( "*** Processing terminated\n" );
    printf( "********************************************************************************\n" );
  } else if( SQLCODE != 0 )
  {
    status = NOT_OK;
    sql_error( "getEmplResume @ SELECT" );
  }
}

void formatEmplResume( void )
{  
  /*********************************************************************
   * Called by the main routine. Calls routines to parse out the
   * contents of the resume into ISPF-shared variables.  
   */

  /*******************************************************************
   * Get the employee's name, address, and other personal information
   */
  getPersonalData();

  /*******************************************************************
   * Get the employee's department no., manager, and other dept data
   */
  if( status == OK )
    getDepartmentData();

  /*******************************************************************
   * Get the employee's education data
   */
  if( status == OK )
    getEducationData();

  /*******************************************************************
   * Get the employee's employment history
   */
  if( status == OK )
    getWorkHistoryData();
}
/* Free the CLOB locator for the resume */
if( status == OK )
{
    EXEC SQL FREE LOCATOR :clRESUME;
    if( SQLCODE != 0 )
    {
        status = NOT_OK;
        sql_error( "formatEmplResume @ FREE LOCATOR" );
    }
}
} /* end formatEmplResume */

void getPersonalData( void )
					
/* Called by the formatEmplResume routine to parse the CLOB locator data for the employee's name, address, home telephone no., birth- * date, sex, marital status, height, and weight into ISPF variables. */
{
					/* Extract the Personal Data section from the CLOB locator */
EXEC SQL SET :begSection /* locate start of pers. data */
= POSSTR( :clRESUME, '    Resume:  ' );
if( SQLCODE != 0 )
{
    status = NOT_OK;
    sql_error( "getPersonalData @ POSSTR 1" );
}
if( status == OK )
{
    EXEC SQL SET :endSection /* locate start of dept. data */
= POSSTR( :clRESUME, '    Department Information   ' );
    if( SQLCODE != 0 )
    {
        status = NOT_OK;
        sql_error( "getPersonalData @ POSSTR 2" );
    }
    if( status == OK )
    {
        EXEC SQL SET :hvRESUME /* extract what's in between */
= SUBSTR( :clRESUME, :begSection, :endSection-:begSection );
        if( SQLCODE == 0 )
            hvRESUME.data[hvRESUME.length] = '\0';
        else
        {
            status = NOT_OK;
            sql_error( "getPersonalData @ SUBSTR" );
        }
    }
					/* Get the employee's name */
if( status == OK )
{
    phvRESUME = &hvRESUME.data[0]; /* set pointer to the data */
    begField /* find Resume: label */
= strstr( phvRESUME," Resume: " );
    begField = begField + 11; /* skip past label */
    endField /* find Personal Inf... label */
= strstr( phvRESUME," Personal Information ");
    strncpy( D8EMNAME, /* get name from in between */
    begField, endField- begField );
}
					/* Get the employee's street address */
if( status == OK )
{
    begField /* find Address: label */
= strstr( phvRESUME," Address: " );
    begField = begField + 22; /* skip past label */
    endField /* find end of street addr */
= strstr( phvRESUME," ");
    strncpy( D8EMADR1, /* get addr from in between */
    begField,
endField - begField );

} /* Get the employee's city, state, and zipcode */
*******************************************************************
if( status == OK )
{
    begField = endField + 22; /* set loc to city/st/zip dat */
    endField /* find end of city/st/zip */
    = strstr( phvRESUME,"  Phone:              ");
    strncpy( DBMADR2, /* get data from in between */
    begField,
    endField - begField );
}
************************************************************************
/* Get the employee's home telephone number */
*******************************************************************
if( status == OK )
{
    begField = endField + 22; /* set loc to home phone data */
    endField /* find end of home phone no. */
    = strstr( phvRESUME,"  Birthdate:          ");
    strncpy( DBMADR3, /* get phone# from in between */
    begField,
    endField - begField );

} /* Get the employee's birthdate */
*******************************************************************
if( status == OK )
{
    begField = endField + 22; /* set loc to birthdate data */
    endField /* find end of birthdate data */
    = strstr( phvRESUME,"  Sex:                ");
    strncpy( DBEMBORN, /* get birthdate from in betw */
    begField,
    endField - begField );

} /* Get the employee's sex */
*******************************************************************
if( status == OK )
{
    begField = endField + 22; /* set loc to sex data */
    endField /* find end of sex data */
    = strstr( phvRESUME,"  Marital Status:     ");
    strncpy( DBEMSEX, /* get sex data from in betw */
    begField,
    endField - begField );

} /* Get the employee's marital status */
*******************************************************************
if( status == OK )
{
    begField = endField + 22; /* set loc to marital status */
    endField /* find end of marital stat. */
    = strstr( phvRESUME,"  Height:             ");
    strncpy( DBEMPMT, /* get mar stat from in betw */
    begField,
    endField - begField );

} /* Get the employee's height */
*******************************************************************
if( status == OK )
{
    begField = endField + 22; /* set loc to height data */
    endField /* find end of height data */
    = strstr( phvRESUME,"  Weight:             ");
    strncpy( DBEMHGT, /* get height from in between */
    begField,
    endField - begField );

} /* Get the employee's weight */
*******************************************************************
if( status == OK )
{
    begField = endField + 22; /* set loc to weight data */
    strcpy( DBEMWGT, /* weight is at end of string */
    begField );

}
void getDepartmentData( void )
{ /* end getPersonalData */

/* Called by the formatEmpResume routine to parse the CLOB locator */
/* data for the employee's department number, manager, job position, * 
/* work telephone no., and hire date into ISPF variables. */
*********************************************************************/
{ /* Extract the Department Data section from the CLOB locator */
EXEC SQL SET :endSection = POSSTR( :clRESUME, '    Education   ' );
if( SQLCODE != 0 )
{
    status = NOT_OK;
    sql_error( "getDepartmentData @ POSSTR" );
}
if( status == OK )
{
    EXEC SQL SET :hvRESUME = SUBSTR( :clRESUME, :begSection, :endSection-:begSection );
    if( SQLCODE == 0 )
    {
        hvRESUME.data[hvRESUME.length] = '\0';
    }
    else
    {
        status = NOT_OK;
        sql_error( "getDepartmentData @ SUBSTR" );
    }
}

/* Get the employee's department number */
*********************************************************************/
if( status == OK )
{
    phvRESUME = &hvRESUME.data[0]; /* set pointer to the data */
    begField = strstr( phvRESUME," Dept Number:        ");
    begField = begField + 22; /* skip past label */
    endField = strstr( phvRESUME," Manager:            ");
    strncpy( D8EMDEPT, /* get dept# from in between */
             begField,
             endField - begField );
}

/* Get the employee's manager's name */
*********************************************************************/
if( status == OK )
{
    begField = endField + 22; /* set loc to manager data */
    endField = strstr( phvRESUME," Manager:            ");
    strncpy( D8MGRNAM, /* get mgr name from in betw */
             begField,
             endField - begField );
}

/* Get the employee's job position */
*********************************************************************/
if( status == OK )
{
    begField = endField + 22; /* set loc to position data */
    endField = strstr( phvRESUME," Phone:              ");
    strncpy( D8EMPOSN, /* get position from in betw */
             begField,
             endField - begField );
}

/* Get the employee's work telephone number */
*********************************************************************/
if( status == OK )
{
    begField = endField + 22; /* set loc to work phone data */
    endField = strstr( phvRESUME," Hire Date:          ");
}
void getEducationData( void )
/* Called by the formatEmplResume routine to parse the CLOB locator */
/* data for the employee's degree dates, descriptions, and schools */
/* into ISPF variables. */
{
/*******************************************************************
* Extract the Education Data section from the CLOB locator       
* *******************************************************************/
begSection = endSection;       /* Locate start of Educ data */
EXEC SQL SET :endSection = POSSTR( :clRESUME, '    Work History   ' );
if( SQLCODE != 0 )
{
    status = NOT_OK;
    sql_error( "getEducationData @ POSSTR" );
}
if( status == OK )
{
    EXEC SQL SET :hvRESUME = SUBSTR( :clRESUME, :begSection, :endSection-:begSection );
    if( SQLCODE == 0 )
    {
        hvRESUME.data[hvRESUME.length] = '\0';
    }
    else
    {
        status = NOT_OK;
        sql_error( "getEducationData @ SUBSTR" );
    }
}
/*******************************************************************
* Get year and description of employee's most recent degree       
* *******************************************************************/
if( status == OK )
{
    phvRESUME = &hvRESUME.data[0];  /* set pointer to the data */
    begField = strstr( phvRESUME,"    Education   ");
    begField = begField + 16;       /* skip past label */
    endField = strstr( phvRESUME,"                      ");
    strncpy( DBEMEDY1, /* get dept# from in between */
             begField, endField - begField );
    begField = endField + 16;       /* set loc to degree descript */
    endField = strstr( phvRESUME,"                      ");
    strncpy( DBEMEDD1, /* get deg descr from in betw */
             begField, endField - begField );
}
/*******************************************************************
* Get institution that granted employee's most recent degree       
* *******************************************************************/
if( status == OK )
{
    begField = endField + 22;       /* set loc to inst name data */
    phvRESUME = begField;          /* point to beginning */
    endField = strstr( phvRESUME,"    ");
    strncpy( DBEMEDI1, /* get inst name from in betw */
             begField, endField - begField );
}
 /*******************************************************************
* Get year and description of employee's previous degree           
* *******************************************************************/
if( status == OK )
{
    begField = endField + 22;       /* set loc to inst name data */
    phvRESUME = begField;          /* point to beginning */
    endField = strstr( phvRESUME,"    ");
    strncpy( DBEMEDI1, /* get inst name from in betw */
             begField, endField - begField );
}
 /*******************************************************************
* Get year and description of employee's previous degree           
* *******************************************************************/
if( status == OK )
{
    begField = endField + 3; /* set loc to grad year data */
    endField = strstr( phvRESUME," */
    strncpy( DBEMEDY2, /* get hire data from in betw */
             begField,
             endField - begField );
    begField = endField + 16; /* set loc to degree descript */
    endField = strstr( phvRESUME," */
                     begField,
                     endField - begField );
}

/*********************************************/
* Get institution that granted employee's previous degree          *
*********************************************/
if( status == OK )
{
    begField = endField + 22; /* set loc to inst name data */
    strncpy( DBEMEDI2, /* inst name is at end of str */
             begField );
}
} /* end getEducationData */

void getWorkHistoryData( void )
/***************************************************************************
* Called by the formatEmplResume routine to parse the CLOB locator   *
* data for the employee's job dates, titles, and descriptions into   *
* ISPF variables.                                                    *
***************************************************************************/
{
    begSection = endSection; /* Locate start of Work Hist */
    EXEC SQL SET :endSection = POSSTR( :clRESUME, '    Interests   ' );
    if( SQLCODE != 0 )
    {
        status = NOT_OK;
        sql_error( "getWorkHistoryData @ POSSTR" );
    }
    if( status == OK )
    {
        EXEC SQL SET :hvRESUME = SUBSTR( :clRESUME, :begSection, :endSection-:begSection );
        if( SQLCODE == 0 )
        {
            hvRESUME.data[hvRESUME.length] = '\0';
        }
        else
        {
            status = NOT_OK;
            sql_error( "getWorkHistoryData @ SUBSTR" );
        }
    }
}

/*********************************************/
* Get dates and title of employee's most recent job                *
*********************************************/
if( status == OK )
{
    phvRESUME = &hvRESUME.data[0]; /* set pointer to the data */
    begField = strstr( phvRESUME," Work History ");
    begField = begField + 19; /* set loc to job 1 dates */
    strncpy( DBEMWHD1, /* job 1 dates, next 15 bytes */
             begField,
             15 );
    endField = begField + 20; /* set loc to job 1 title */
    strncpy( DBEMWHT1, /* get job 1 title from betw */
             begField,
             endField - begField );
}

/*********************************************/
* Get description of employee's most recent job                    *
/*********************************************/
if( status == OK )
{
    begField = endField + 22;  /* set loc to job 1 descr. */
    phvRESUME = begField;  /* reset starting point */
    endField = strstr( phvRESUME, "." );  /* find end of job 1 descr. */
    if( endField - begField < 62 )  /* job 1 descr has 1 part */
        strncpy( D8EMWHJ1, begField, endField - begField);
    else  /* job 1 descr has 2 parts */
    {
        endField = strstr( phvRESUME, " ");  /* find 1st part of job descr */
        strncpy( D8EMWHJ1, begField, endField - begField );
        begField = endField + 22;  /* set loc to 2nd part job descr */
        endField = strstr( phvRESUME, " ");  /* find end of job 1 descr. */
        strncat( D8EMWHJ1, begField-1, endField - (begField-1) );
    }
}
}  /* Get dates and title of employee's previous job */

if( status == OK )
{
    begField = endField + 4;  /* set loc to job 2 dates */
    phvRESUME = begField;  /* reset starting point */
    strncpy( D8EMWHD2, begField, 15 );
    begField = begField + 20;  /* set loc to job 2 title */
    endField = strstr( phvRESUME, " ");  /* find end of job 2 title */
    strncpy( D8EMWHT2, begField, endField - begField );
}
}  /* Get description of employee's previous job */

if( status == OK )
{
    begField = endField + 22;  /* set loc to job 2 descr. */
    phvRESUME = begField;  /* reset starting point */
    endField = strstr( phvRESUME, "." );  /* find end of job 2 descr. */
    if( endField - begField < 62 )  /* job 2 descr has 1 part */
        strncpy( D8EMWHJ2, begField, endField - begField);
    else  /* job 2 descr has 2 parts */
    {
        endField = strstr( phvRESUME, " ");  /* find 1st part of job descr */
        strncpy( D8EMWHJ2, begField, endField - begField );
        begField = endField + 22;  /* set loc to 2nd part job descr */
        endField = strstr( phvRESUME, " ");  /* find end of job 2 descr. */
        strncat( D8EMWHJ2, begField-1, endField - (begField-1) );
    }
}
}  /* Get dates and title of employee's other previous job */

if( status == OK )
{
    begField = endField + 4;  /* set loc to job 3 dates */
    phvRESUME = begField;  /* reset starting point */
    strncpy( D8EMWHD3, begField, 15 );
    begField = begField + 20;  /* set loc to job 3 title */
    endField = strstr( phvRESUME, " ");  /* find end of job 3 title */
}
= strstr( phvRESUME,"                      ");
strncpy( D8EMWHT3, /* get job 3 title from betw */
    begField,
    endField - begField);
}

/*******************************************************************
* Get description of employee's other previous job                 *
*******************************************************************/
if( status == OK )
{
    begField = endField + 22;       /* set loc to job 3 descr.    */
    phvRESUME = begField;           /* reset starting point       */
    begField = phvRESUME;           /* reset starting point       */
    endField = strstr( phvRESUME,"." );
    if( endField - begField < 62 )  /* job 3 descr has 1 part     */
        strncpy( D8EMWHJ3,          /* get job 3 title from betw */
            begField,
            endField - begField);
    else                            /* job 3 descr has 2 parts    */
    {
        endField = strstr( phvRESUME,"                      ");
        strncpy( D8EMWHJ3,          /* get job 3 descr from betw */
            begField,
            endField - begField );
        begField = endField + 22;   /* set loc to 2nd part job des*/
        endField = strstr( phvRESUME,".   ");
        strncat( D8EMWHJ3,          /* get rest of job 3 descr.   */
            begField-1,
            endField - (begField-1) );
    }
}
} /* end getWorkHistoryData */

void showEmplResume( void )
/**************************
* Called by the main routine. Displays an ISPF panel that is for- *
* matted with the resume data for the employee specified.          *
**************************/
{
    ISPFrc = isplink( "DISPLAY ","DSNBSSR ");
} /* end showEmplResume */

void freeISPFvars( void )
/* Called by the main routine. Frees the ISPF variables that were    *
* established for running this application.                          */
{
    ISPFrc = isplink( VRESET );
} /* end freeISPFvars */

void sql_error( char *locmsg )
/******************************
* SQL error handler         *
******************************
{
    short int   xc;                     /* DSNTIAR Return code       */
    int         j,k;                    /* Loop control               */
    static int  lrecl = TIAR_LEN;       /* Width of message lines     */

    /* print the location message */
    printf( "*****************************************************
    *** ERROR: DSN8DLRV DB2 Sample Program\n" );
    printf("*** Unexpected SQLCODE encountered at location\n" );
    printf("*** %68s\n", locmsg );
    printf("*** Error detailed below\n" );
    printf("*** Processing terminated\n" );
    printf("*****************************************************\n");
} /* szError */
Related reference
“Sample applications in TSO” on page 1027
A set of Db2 sample applications run in the TSO environment.

**DSN8DLPV**

Prompts the user to choose an employee, then retrieves the PSEG photo image for that employee from the PSEG_- PHOTO column of the EMP_PHOTO_RESUME table and passes it to GDDM for formatting and display.

Chapter 12. Sample data and applications supplied with Db2 for z/OS 1279
No entry in the Employee Photo/Resume table for employee with empno = xxxxxx.
Processing terminated.

- Message: *** ERROR: DSN8DLPV DB2 Sample Program
  No PSEG photo image exists in the Employee Photo/Resume table for the employee with empno = xxxxxx.
Processing terminated.

External References:
- Routines/Services: DSNTIAR, GDDM, ISPF
- Data areas : DSNTIAR error_message
- Control blocks : None

Pseudocode:
DSN8DLPV:
- Do until the user indicates termination
  - Call getEmplNum to request an employee id
  - Call getEmplPhoto to retrieve the PSEG photo image
  - Call showEmplPhoto to display the photo
  - Call sql_error for unexpected SQLCODEs
  - Call DSNTIAR to format the unexpected SQLCODE.

getEmplNum:
- prompt user to select an employee whose photo is to be viewed

getEmplPhoto:
- Fetch the specified employee's PSEG photo image from DB2
- Call sql_error for unexpected SQLCODEs

showEmplPhoto:
- Use GDDM calls to format and display the PSEG photo image

sql_error:
- Call DSNTIAR to format the unexpected SQLCODE.

/*****************************************************/
C Program Product Libraries ********************/
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

GDDM Program Product Libraries (Reentrant Versions) ******
#include <ADMUCIRA>
#include <ADMTSTRC>
#include <ADMUCIRF>
#include <ADMUCIRG>
#include <ADMUCIRI>

Equates *****************************************************/
#define NO 0 /* Boolean: False */
define YES 1 /* Boolean: True */
#define NOT_OK 0 /* Run status indicator: Error*/
define OK 1 /* Run status indicator: Good */
define TIAR_DIM 10 /* Max no. of DSNTIAR msgs */
define TIAR_LEN 80 /* Length of DSNTIAR messages */

Global Storage *****************************************************/
int keepViewing = YES; /*
int status = OK; /* run status */
short int ISPFrc; // For ISPF return code */

DB2 SQL Communication Area *****************************/
EXEC SQL INCLUDE SQLCA;

DB2 Message Formatter *****************************************************/
struct error_struct { /* DSNTIAR message structure */
  short int error_len;
  char error_text[TIAR_DIM][TIAR_LEN];

extern short int dsntiar( struct sqlca *sqlca, struct error_struct *msg, int *len );

EXEC SQL DECLARE  EMP_PHOTO_RESUME  TABLE
( EMPNO CHAR(06)     NOT NULL, EMP_ROWID ROWID, PSEG_PHOTO BLOB( 500K ), BMP_PHOTO BLOB( 100K ), RESUME CLOB( 5K ) );

EXEC SQL BEGIN DECLARE SECTION;
char hvEMPNO[7]; /*                            */
short int niPSEG_PHOTO = 0; /*                            */
EXEC SQL END DECLARE SECTION;

union{ Admaab AABtag; char AABstr[8]; } AAB;
int appl_id; /* id for application image */
/* appl. image vars follow */
int ih_pixels = 800; /* -horiz size in # of pixels */
int iv_pixels = 750; /* -vert size in # of pixels */
int iim_type = 1; /* -pixel type (1=bi-level) */
int ires_type = 1; /* -defined resolution indic. */
int ires_unit = 0; /* -resolution units (0=inches) */
float ih_res = 100.00; /* -horizontal resolution */
float iv_res = 100.00; /* -vertical resolution */
int PSEGformat = -3; /* indicates PSEG/GDDM conver'n factors*/
int PSEGcompression = 4; /* indicates IBM 3800 compresn*/
int attype; /* type of attn/interrupt key */
int attval; /* value of attn/interrupt key*/
int count; /* number of fields modified */

EXEC SQL END DECLARE SECTION;

Chapter 12. Sample data and applications supplied with Db2 for z/OS
/* Display employee photos until user indicates completion */
keepViewing = YES;
while( keepViewing == YES )
{
    /* prompt user to select an employee whose photo is to be viewed */
    getEmplNum();
    if( keepViewing == YES && status == OK )
    {
        /* extract the employee's PSEG photo image from BLOB storage */
        getEmplPhoto();
        /* if okay, convert PSEG image to GDDM format and display it */
        if( status == OK )
            showEmplPhoto();
        /* otherwise, exit this program */
        else
            keepViewing = NO;
    }
}

/* end main */

void getEmplNum( void )
{
    /* Called by the main routine. Displays an ISPF panels to prompt the */
    /* user to select an employee whose photo image is to be displayed. */
    {
        /* Share the ISPF var having the employee number */
        ISPFRc = isplink( VDEFINE, "D8EMNUMB", D8EMNUMB, CHAR, 6 );
        strcpy( D8EMNUMB,"      ");
        /* Display the prompt panel */
        ISPFRc = isplink( DISPLAY,"DSN8SSE ");
        if( ISPFRc != 0 )
            keepViewing = NO;
        /* Save off the value of the ISPF shared variable */
        strcpy( hvEMPNO,D8EMNUMB );
        /* And release it */
        ISPFRc = isplink( VRESET );
    } /* end getEmplNum */

void getEmplPhoto( void )
{
    /* Called by the main routine. Extracts a specified employee's PSEG */
    /* photo image from a BLOB column in the sample EMP_PHOTO_RESUME. */
    /* This image will be converted to GDDM format and displayed by the */
    /* routine showEmplPhoto. */
    {
        EXEC SQL SELECT PSEG_PHOTO
                       INTO :hvPSEG_PHOTO
                       FROM EMP_PHOTO_RESUME
                       WHERE EMPNO = :hvEMPNO;

        if( SQLCODE == 0 )
            hvPSEG_PHOTO.data[hvPSEG_PHOTO.length] = '\n';
        else if( SQLCODE == 100 )
            
    }
void showEmplPhoto( void )
{
    /**********************************************************
    * Called by the main routine. Converts the employee's photo from  *
    * PSEG format to a GDDM image and then displays it until the user  *
    * depresses any PF key or the <enter> key.                     *
    **********************************************************/

    isplink( CONTROL_DISPLAY_LINE );

    /****************************************************************************
    * Signal ISPF to full-screen refresh when GDDM session terminates          *
    ****************************************************************************/
    
    /************************************************************
    * Initialize GDDM                                           *
    ************************************************************/
    fsinit( AAB.AABstr ); /* GDDM anchor block */

    /*******************************
    * Obtain a GDDM application image id                         *
    *******************************
    imagid( AAB.AABstr, 
            /* GDDM anchor block */
            &appl_id ); /* application id for image */

    /**********************************************************
    * Create a GDDM application image to receive the employee photo *
    **********************************************************/
    imacrt( AAB.AABstr, 
            /* GDDM anchor block */
            appl_id, 
            /* target: application image */
            ih_pixels, /* horiz size in # of pixels */
            iv_pixels, /* vert size in # of pixels */
            iim_type, /* pixel type (1=bi-level) */
            irres_type, /* defined resolution indic. */
            irres_unit, /* resolut'n units (0-inches) */
            ih_res, /* horizontal resolution */
            iv_res /* vertical resolution */
            );

    /****************************************************************************
    * Set up conversion of photo from PSEG format to GDDM format          *
    ****************************************************************************/
    imapts( AAB.AABstr, 
            /* GDDM anchor block */
            appl_id, 
            /* target: application image */
            0, 
            /* GDDM proj. id (0=identity) */
            PSEGformat, /* source format (PSEG) */
            PSEGcompression ); /* source compression (3800) */

    /*****************************
    * Perform conversion       *
    *****************************
    imapt( AAB.AABstr, 
            /* GDDM anchor block */
            appl_id, /* target: application image */
            hvPSEG_PHOTO.length, /* source length */
            hvPSEG_PHOTO.data ); /* source: employee PSEG photo*/
}}
/* Terminate conversion */
impte( AAB.AABstr, /* GDDM anchor block */
    appl_id ); /* target: application image */

/* Transfer the GDDM application image to the display */
imxfer( AAB.AABstr, /* GDDM anchor block */
    appl_id, /* source: application image */
    0, /* target: 0=display */
    0 ); /* GDDM proj. id (0=identity) */

/* Disable user updates to the image on the display */
fsenab( AAB.AABstr, /* GDDM anchor block */
    1, /* type of input (1=alphanum) */
    0 ); /* type of control (0=disable) */

fsenab( AAB.AABstr, /* GDDM anchor block */
    2, /* type of input (2=graphic) */
    0 ); /* type of control (0=disable) */

fsenab( AAB.AABstr, /* GDDM anchor block */
    3, /* type of input (3=image) */
    0 ); /* type of control (0=disable) */

/* Display the image until attn or interrupt key depressed */
asread( AAB.AABstr, /* GDDM anchor block */
    &attype, /* type of attn/interrupt key */
    &attval, /* value of attn/interrupt key */
    &count ); /* number of fields modified */

/* Delete the GDDM application image */
imadel( AAB.AABstr, /* GDDM anchor block */
    appl_id ); /* target: application image */

/* Terminate GDDM */
fsterm( AAB.AABstr ); /* GDDM anchor block */

} /* end showEmplPhoto */

void sql_error( char *locmsg )

/* SQL error handler */
{
    short int  rc; /* DSNTIAr Return code */
    int        j,k; /* Loop control */
    static int lrecl = TIAR_LEN; /* Width of message lines */

    /* print the location message */
    printf( "*****************************************************\n");
    printf( "*** ERROR: DSN8DLPV DB2 Sample Program\n");
    printf( "***        Unexpected SQLCODE encountered at location\n");
    printf( "***         %.68s\n", locmsg);
    printf( "***        Error detailed below\n");
    printf( "***        Processing terminated\n");
    printf( "*****************************************************\n");

    /* format and print the SQL message */
    rc = dsntiai( &sqlca, &error_message, &lrecl );
    if( rc == 0 )
        for( j=0; j<TIAR_DIM; j++ )
            for( k=0; k<TIAR_LEN; k++ )
                putchar( error_message.error_text[j][k] );
        \n");
else
{
    printf( " *** ERROR: DSNTIAR could not format the message\n" ) ;
    printf( " *** SQLCODE is %d\n",SQLCODE ) ;
    printf( " *** SQLERRM is \n" ) ;
    for( j=0; j<sqlca.sqlerrml; j++ )
        printf( "%c", sqlca.sqlerrmc[j] ) ;
    printf( "\n" ) ;
}
} /* end sql_error */

Related reference
“Sample applications in TSO” on page 1027
A set of Db2 sample applications run in the TSO environment.

DSNTEJ2C
THIS JCL PERFORMS THE PHASE 2 COBOL SETUP FOR THE SAMPLE APPLICATIONS.

******************************************************************************
* NAME = DSNTEJ2C
* DESCRIPTIVE NAME = DB2 SAMPLE APPLICATION
* PHASE 2
* COBOL
* LICENSED MATERIALS - PROPERTY OF IBM
* 5650-DB2
* (C) COPYRIGHT 1982, 2016 IBM CORP.  ALL RIGHTS RESERVED.
* STATUS = VERSION 12
* FUNCTION = THIS JCL PERFORMS THE PHASE 2 COBOL SETUP FOR THE
*            SAMPLE APPLICATIONS. IT PREPARES AND EXECUTES
*            COBOL BATCH PROGRAMS.
*            THIS JOB IS RUN AFTER PHASE 1.
* CHANGE ACTIVITY =
* 08/18/2014 Single-phase migration s21938_inst1 s21938
******************************************************************************

//JOBLIB  DD  DISP=SHR,DSN=DSN!!0.SDSNEXIT
//        DD  DISP=SHR,DSN=DSN!!0.SDSNLOAD
//        DD  DISP=SHR,DSN=CEE.V!R!M!.SCEERUN
//STEP  1: CREATE COPY FILE TABLE DESCRIPTIONS (DCLGEN)
//PH02CS01 EXEC PGM=IKJEFT01,DYNAMNBR=20
//SYSTSPRT DD SYSOUT=*,DCB=(RECFM=F,LRECL=200,BLKSIZE=200)
//SYSTSIN  DD *
//DELETE 'DSN!!0.SRCLIB.DATA(DSN8MCDP)'
//DELETE 'DSN!!0.SRCLIB.DATA(DSN8MCEM)'
//DELETE 'DSN!!0.SRCLIB.DATA(DSN8MCDM)'
//DELETE 'DSN!!0.SRCLIB.DATA(DSN8MCA2)'
//DELETE 'DSN!!0.SRCLIB.DATA(DSN8MCAD)'
//DELETE 'DSN!!0.SRCLIB.DATA(DSN8MCA2)'
//DELETE 'DSN!!0.SRCLIB.DATA(DSN8MCAD)'
//DELETE 'DSN!!0.SRCLIB.DATA(DSN8MCED)'
//DSN SYSTEM(DSN)
//DCLGEN TABLE(VDEPT) +
//   OWNER(DSN8!!0) +
//   LIBRARY('DSN!!0.SRCLIB.DATA(DSN8MCDP)') +
//   ACTION(ADD) APOST +
//   LANGUAGE(IBMCOB) +
//   STRUCTURE(PDEPT)
//DCLGEN TABLE(VEMP) +
//   OWNER(DSN8!!0) +
//   LIBRARY('DSN!!0.SRCLIB.DATA(DSN8MCEM)') +
//   ACTION(ADD) APOST +
//   LANGUAGE(IBMCOB) +
//   STRUCTURE(PEMP)
//DCLGEN TABLE(VDEPMG1) +
OWNER('DSN8!!0') +
LIBRARY('DSN8!!0.SRCLIB.DATA(DSN8MCDM)') +
ACTION(ADD) APOST +
LANGUAGE(IBMCOB) +
STRUCTURE(PDEPMGR)
DCLGEN TABLE(VASTRDE1) +
OWNER('DSN8!!0') +
LIBRARY('DSN8!!0.SRCLIB.DATA(DSN8MCAD)') +
ACTION(ADD) APOST +
LANGUAGE(IBMCOB) +
STRUCTURE(PASTRDET)
DCLGEN TABLE(VASTRDE2) +
OWNER('DSN8!!0') +
LIBRARY('DSN8!!0.SRCLIB.DATA(DSN8MCA2)') +
ACTION(ADD) APOST +
LANGUAGE(IBMCOB) +
NAMES(ADE2) +
STRUCTURE(PASTRDE2)
DCLGEN TABLE(VCONA) +
OWNER('DSN8!!0') +
LIBRARY('DSN8!!0.SRCLIB.DATA(DSN8MCCS)') +
ACTION(ADD) APOST +
LANGUAGE(IBMCOB) +
STRUCTURE(PCONA)
DCLGEN TABLE(VOPTVAL) +
OWNER('DSN8!!0') +
LIBRARY('DSN8!!0.SRCLIB.DATA(DSN8MCOV)') +
ACTION(ADD) APOST +
LANGUAGE(IBMCOB) +
STRUCTURE(POPTVAL)
DCLGEN TABLE(VDSPTXT) +
OWNER('DSN8!!0') +
LIBRARY('DSN8!!0.SRCLIB.DATA(DSN8MCDT)') +
ACTION(ADD) APOST +
LANGUAGE(IBMCOB) +
STRUCTURE(PDSPTXT)
DCLGEN TABLE(VEMPDPT1) +
OWNER('DSN8!!0') +
LIBRARY('DSN8!!0.SRCLIB.DATA(DSN8MCED)') +
ACTION(ADD) APOST +
LANGUAGE(IBMCOB) +
STRUCTURE(PEMPDPT1)
END

//**
//**        STEP  2: PREPARE ERROR MESSAGE ROUTINE
//PH02CS02 EXEC DSNHICOB,MEM=DSN8MCGB,
//         COND=(4,LT),
//         PARM.PC=('HOST(IBMCOB)',APOST,APOSTSQL,SOURCE,
//         NOXREF,'SQL(DB2)', 'DEC(31)'),
//         PARM.COB=('NOSEQUENCE',QUOTE,RENT,'PGMNAME(LONGUPPER)'),
//         PARM.LKED='LIST,XREF,MAP,RENT'
//PC.DBRMLIB   DD DSN=DSN!!0.DBRMLIB.DATA(DSN8MCGB),
//             DISP=SHR
//PC.SYSLIB    DD DSN=DSN!!0.SRCLIB.DATA,
//             DISP=SHR
//PC.SYSIN     DD DSN=DSN!!0.SDSNSAMP(DSN8MCGB),
//             DISP=SHR
//LKED.SYSLMOD DD DSN=DSN!!0.RUNLIB.LOAD(DSN8MCGB),
//             DISP=SHR
//**
//**        STEP  3: PREPARE COBOL PHONE PROGRAM
//PH02CS03 EXEC DSNHICOB,MEM=DSN8BCC3,
//         COND=(4,LT),
//         PARM.PC=('HOST(IBMCOB)',APOST,APOSTSQL,SOURCE,
//         NOXREF,'SQL(DB2)', 'DEC(31)'),
//         PARM.COB=('NOSEQUENCE',QUOTE,RENT,'PGMNAME(LONGUPPER)'),
//         PARM.LKED='LIST,XREF,MAP,RENT'
//PC.DBRMLIB   DD DSN=DSN!!0.DBRMLIB.DATA(DSN8BCC3),
//             DISP=SHR
//PC.SYSLIB    DD DSN=DSN!!0.SRCLIB.DATA,
//             DISP=SHR
//PC.SYSIN     DD DSN=DSN!!0.SDSNSAMP(DSN8BCC3),
//             DISP=SHR
//LKED.SYSLMOD DD DSN=DSN!!0.RUNLIB.LOAD(DSN8BCC3),
//             DISP=SHR
//**
//**        STEP  4: BIND AND RUN PROGRAMS
//PH02CS04 EXEC PGM=IKJEFT01,DYNAMNBR=20,COND=(4,LT)
Related reference
“Sample applications in TSO” on page 1027
A set of Db2 sample applications run in the TSO environment.

**DSNTEJ2D**

THIS JCL PERFORMS THE PHASE 2 C LANGUAGE SETUP FOR THE SAMPLE APPLICATIONS.
Related reference

“Sample applications in TSO” on page 1027
A set of Db2 sample applications run in the TSO environment.

**DSNTEJ2E**

THIS JCL PERFORMS THE PHASE 2 C++ LANGUAGE SETUP FOR THE SAMPLE APPLICATIONS.
FUNCTION = THIS JCL PERFORMS THE PHASE 2 C++ LANGUAGE SETUP FOR 
THE SAMPLE APPLICATIONS. IT PREPARES AND EXECUTES 
C++ BATCH PROGRAMS.

NOTES = ENSURE THAT LINE NUMBER SEQUENCING IS SET 'ON' IF 
THIS JOB IS SUBMITTED FROM AN ISPF EDIT SESSION

CHANGE ACTIVITY =
10/16/2013 Don't use prelinker by default PI13612 DM1812
08/18/2014 Single-phase migration s21938_inst1 s21938

********************************************************************

JOBLIB DD DSN=DSN!0.SDSNLOAD,DISP=SHR
    DD DSN=CEEV!R!M!.SCEERUN,DISP=SHR

STEP 1 : PREPARE ERROR MESSAGE ROUTINE
PH02ES01 EXEC DSNHC,MEM=DSN8MDG,
PARM.PC=('HOST(C),CCSID(1047),MARGINS(1,72),STDSQL(NO)',
    'SOURCE,XREF'),
PARM.C='SOURCE XREF MARGINS(1,72) OPTFILE(DD:CCOPTS)',
PARM.LKED='NCAL,MAP,AMODE=31,RMODE=ANY'
    PC.DBRMLIB DD DSN=DSN!0.DBRMLIB.DATA(DSN8MDG),
    DISP=SHR
    PC.SYSLIB DD DSN=DSN!0.SRCLIB.DATA,
    DISP=SHR
    PC.SYSIN DD DSN=DSN!0.SDSNSAMP(DSN8MDG),
    DISP=SHR
    C.SYSLIN DD DSN=&&LOADSET2,
    DISP=(MOD,PASS),
    UNIT=SYSDA,SPACE=(32000,(30,30)),
    DCB=(RECFM=FB,LRECL=80,BLKSIZE=3200)
/LKED.SYSLIN DD DSN=&&LOADSET2,DISP=(OLD,PASS)
/LKED.SYSLMOD DD DSN=DSN!0.RUNLIB.LOAD(DSN8MDG),
    DISP=SHR

STEP 2 : PREPARE CLASSES USED BY C++ PHONE PROGRAM
PH02ES02 EXEC DSNHCPP,MEM=DSN8BECL,COND=(4,LT),
PARM.PC=('HOST(CPP),CCSID(1047),MARGINS(1,72),STDSQL(NO)',
    'SOURCE,XREF'),
PARM.CP=('/CXX SOURCE XREF OPTFILE(DD:CCOPTS)',
    'LANGLVL(EXTENDED)'),
PARM.LKED='NCAL,MAP,AMODE=31,RMODE=ANY'
    PC.DBRMLIB DD DSN=DSN!0.DBRMLIB.DATA(DSN8BE3),
    DISP=SHR
    PC.SYSLIB DD DSN=DSN!0.SRCLIB.DATA,
    DISP=SHR
    PC.SYSIN DD DSN=DSN!0.SDSNSAMP(DSN8BECL),
    DISP=SHR
    CP.USERLIB DD DSN=DSN!0.SRCLIB.DATA,
    DISP=SHR
    CP.SYSLIN DD DSN=&&LOADSET1,
    DISP=(MOD,PASS),
    UNIT=SYSDA,SPACE=(32000,(30,30)),
    DCB=(RECFM=FB,LRECL=80,BLKSIZE=3200)
/LKED.SYSLIN DD DSN=&&LOADSET1,DISP=(OLD,PASS)
/LKED.SYSLMOD DD DSN=DSN!0.RUNLIB.LOAD(DSN8BECL),
    DISP=SHR
/LKED.RUNLIB DD DSN=DSN!0.RUNLIB.LOAD,
    DISP=SHR

STEP 3 : PREPARE C++ PHONE PROGRAM
PH02ES03 EXEC DSNHCPP,MEM=DSN8BE3,
    COND=(4,LT),
PARM.PC=('HOST(CPP),CCSID(1047),MARGINS(1,72),STDSQL(NO)',
    'SOURCE,XREF'),
PARM.CP=('/CXX SOURCE XREF OPTFILE(DD:CCOPTS)',
    'LANGLVL(EXTENDED)'),
PARM.LKED='AMODE=31,RMODE=ANY,MAP,UPCASE'
    PC.DBRMLIB DD DSN=DSN!0.DBRMLIB.DATA(DUMMY),
    DISP=SHR
    PC.SYSLIB DD DSN=DSN!0.SRCLIB.DATA,
    DISP=SHR
    PC.SYSIN DD DSN=DSN!0.SDSNSAMP(DSN8BE3),
Related reference

“Sample applications in TSO” on page 1027
A set of Db2 sample applications run in the TSO environment.

**DSNTEJ2P**

THIS JCL PERFORMS THE PHASE 2 SETUP FOR THE SAMPLE APPLICATIONS AT SITES WITH PL/I.

```sql
/* ** NAME = DSNTEJ2P  
 ** DESCRIPTIVE NAME = DB2 SAMPLE APPLICATION
 ** PHASE 2 
 ** PL/I 
 ** LICENSED MATERIALS - PROPERTY OF IBM 
 ** 5656-DB2 
 ** (C) COPYRIGHT 1982, 2016 IBM CORP. ALL RIGHTS RESERVED.
 ** STATUS = VERSION 12
 ** FUNCTION = THIS JCL PERFORMS THE PHASE 2 SETUP FOR THE SAMPLE APPLICATIONS AT SITES WITH PL/I. IT PREPARES AND EXECUTES THE PL/I BATCH PROGRAM.
 ** THIS JOB IS RUN AFTER PHASE 1.
 ** CHANGE ACTIVITY = 08/18/2014 Single-phase migration s21938_inst1 s21938 */
```
Chapter 12. Sample data and applications supplied with Db2 for z/OS 1291
LANGUAGE(PLI) +
NAMES(ASTD) +
STRUCTURE(PASTRDE2)
DCLGEN TABLE(VPROJRE1) +
OWNER(DSN8!!0) +
LIBRARY('DSN8!!0.SRCLIB.DATA(DSN8MPPR)') +
ACTION(ADD) +
LANGUAGE(PLI) +
STRUCTURE(APPROJRES)
DCLGEN TABLE(VPSTRDE1) +
OWNER(DSN8!!0) +
LIBRARY('DSN8!!0.SRCLIB.DATA(DSN8MPPD)') +
ACTION(ADD) +
LANGUAGE(PLI) +
STRUCTURE(PPSTRDET)
DCLGEN TABLE(VPSTSTRDE2) +
OWNER(DSN8!!0) +
LIBRARY('DSN8!!0.SRCLIB.DATA(DSN8MPPD)') +
ACTION(ADD) +
LANGUAGE(PLI) +
NAMES(PSTD) +
STRUCTURE(PPSTRDE2)
DCLGEN TABLE(VSTAFAC1) +
OWNER(DSN8!!0) +
LIBRARY('DSN8!!0.SRCLIB.DATA(DSN8MPSA)') +
ACTION(ADD) +
LANGUAGE(PLI) +
NAMES(STAF) +
STRUCTURE(PSTAFAC1)
DCLGEN TABLE(VSTAFAC2) +
OWNER(DSN8!!0) +
LIBRARY('DSN8!!0.SRCLIB.DATA(DSN8MPS2)') +
ACTION(ADD) +
LANGUAGE(PLI) +
STRUCTURE(PSTAFAC2)
DCLGEN TABLE(VCONA) +
OWNER(DSN8!!0) +
LIBRARY('DSN8!!0.SRCLIB.DATA(DSN8MPCS)') +
ACTION(ADD) +
LANGUAGE(PLI) +
STRUCTURE(PCONA)
DCLGEN TABLE(VOPTVAL) +
OWNER(DSN8!!0) +
LIBRARY('DSN8!!0.SRCLIB.DATA(DSN8MPOV)') +
ACTION(ADD) +
LANGUAGE(PLI) +
STRUCTURE(POPTVAL)
DCLGEN TABLE(VDSPTXT) +
OWNER(DSN8!!0) +
LIBRARY('DSN8!!0.SRCLIB.DATA(DSN8MPDT)') +
ACTION(ADD) +
LANGUAGE(PLI) +
STRUCTURE(PDSPTXT)
DCLGEN TABLE(VEMPDPT1) +
OWNER(DSN8!!0) +
LIBRARY('DSN8!!0.SRCLIB.DATA(DSN8MPED)') +
ACTION(ADD) +
LANGUAGE(PLI) +
STRUCTURE(PEMPDPT1)
DCLGEN TABLE(VFORPLA) +
OWNER(DSN8!!0) +
LIBRARY('DSN8!!0.SRCLIB.DATA(DSN8MPFP)') +
ACTION(ADD) +
LANGUAGE(PLI) +
STRUCTURE(PFORPLA)
END

END

//* STEP 2: PREPARE PLI MESSAGE ROUTINE
//* PH02PS02 EXEC DSNHPLI, MEM=DSN8MPG,
//  COND=(4,LT),
//  PARM.PC='HOST(PLI),CCSID(37),SOURCE,XREF,STDSQL(NO)',
//  PARM.PLI=('NOPT,SOURCE,OBJECT,MARGINS(2,72,0),NORENT',
//             'LIMITS(EXTNAME(7)),OPTIONS'),
//  PARM.LKED='NCAL,XREF'
//PPLI.SYSIN   DD DSN=DSN8!!0.SDSNSAMP(DSN8MPG),
//             DISP=SHR
//PC.DBRMLIB   DD DSN=DSN8!!0.Runlib.LOAD(DSN8MPG),
//             DISP=SHR
//LKED.SYSLMOD DD DSN=DSN8!!0.RUNLIB.LOAD(DSN8MPG),
//             DISP=SHR
STEP 3: PREPARE TELEPHONE PROGRAM

PH02PS03 EXEC DSNHPLI, MEM=DSNBPP3,
    COND=(4,LT),
    PARM.PC='HOST(PLI),CCSID(37),SOURCE,XREF,STDSQL(NO)',
    PARM.PLI=('NOPT,SOURCE,OBJECT,MARGINS(2,72,0)',
    'LIMITS(EXTNAM(7)),OPTIONS')

PPLI.SYSIN   DD DSN=DSN!!0.SDSNSAMP(DSN8BP3),
    DISP=SHR
PC.DBRMLIB   DD DSN=DSN!!0.DBRMLIB.DATA(DSN8BP3),
    DISP=SHR
PC.SYSLIB    DD DSN=DSN!!0.SRCLIB.DATA,
    DISP=SHR
LKED.SYSLMOD DD DSN=DSN!!0.RUNLIB.LOAD(DSN8BP3),
    DISP=SHR
LKED.RUNLIB  DD DSN=DSN!!0.RUNLIB.LOAD,
    DISP=SHR
LKED.SYSIN   DD *
    INCLUDE SYSLIB(DSNELI)
    INCLUDE RUNLIB(DSN8MPG)

STEP 4: BIND PROGRAMS

PH02PS04 EXEC PGM=IKJEFT01, DYNAMNBR=20, COND=(4,LT)
    DBRMLIB DD DSN=DSN!!0.DBRMLIB.DATA,
    DISP=SHR
SYSTSPRT DD SYSOUT=*
SYSPRINT DD SYSOUT=*
SYSSDUMP DD SYSOUT=*
SYSSIN DD *
    SET CURRENT SQLID = 'SYSADM';
    GRANT BIND, EXECUTE ON PLAN DSN8BP!!
    TO PUBLIC;

SYSTSPRT DD SYSOUT=*  
SYSPRINT DD SYSOUT=*  
SYSSDUMP DD SYSOUT=*  
SYSSIN DD *  
    BIND PACKAGE(DSN8BP!!) MEMBER(DSN8BP3) APPLCOMPAT(V!!R1) +
        ACT(REPLACE) ISO(CS) CURRENTDATA(YES) ENCODING(EBCDIC) 
    BIND PLAN(DSN8BP!!) PKLIST(DSN8BP!!.*) +
        ACTION(REPLACE) RETAIN +
        ISO(CS) CURRENTDATA(YES) ENCODING(EBCDIC) 
    RUN PROGRAM(DSNTIAD) PLAN(DSN8BP!!) -
        LIB('DSN!!0.RUNLIB.LOAD') 
END

STEP 5: RUN PROGRAMS

PH02PS05 EXEC PGM=IKJEFT01, DYNAMNBR=20, COND=(4,LT)
    SYSTSPRT DD SYSOUT=*  
    SYSPRINT DD SYSOUT=*  
    REPORT DD SYSOUT=*  
    SYSSDUMP DD SYSOUT=*  
    SYSSIN DD *  
    L-*  
    L30%  
    L%SON  
    LSMITH  
    LBROWN   ALAN  
    LBROWN   DAVID  
    U 0002304265  
    SYSTSPRT DD SYSOUT=*  
    SYSPRINT DD SYSOUT=*  
    REPORT DD SYSOUT=*  
    SYSSDUMP DD SYSOUT=*  
    CARDIN DD *
    DSN SYSTEM(DSN)
    RUN PROGRAM(DSN8BP3) PLAN(DSN8BP!!) -
        LIB('DSN!!0.RUNLIB.LOAD') 
END

Related reference
“Sample applications in TSO” on page 1027
A set of Db2 sample applications run in the TSO environment.

DSNTEJ2F

THIS JCL PERFORMS THE PHASE 2 SETUP FOR THE SAMPLE APPLICATIONS AT SITES WITH FORTRAN.
** LICENSED MATERIALS - PROPERTY OF IBM
** 5650-DB2
** (C) COPYRIGHT 1982, 2016 IBM CORP. ALL RIGHTS RESERVED.
**
** STATUS = VERSION 12
**
** FUNCTION = THIS JCL PERFORMS THE PHASE 2 SETUP FOR THE SAMPLE
** APPLICATIONS AT SITES WITH FORTRAN. IT PREPARES
** AND EXECUTES THE FORTRAN BATCH PROGRAM.
**
** THIS JOB IS RUN AFTER PHASE 1.
**
** CHANGE ACTIVITY =
** 08/18/2014 Single-phase migration s21938_inst1 s21938
**
********************************************************************

** JOBLIB   DD  DSN=DSN!!0.SDSNLOAD,DISP=SHR
** DD  DSN=SYS1.VSF2FORT,DISP=SHR
**
** STEP 1: PREPARE DSNTIR ROUTINE
**
** PH02FS01 EXEC DSNHASM,MEM=DSNTIR,
** PARM.PC='HOST(ASM),STDSQL(NO)',
** PARM.ASM='RENT,OBJECT,NODECK',
** PARM.LKED='XREF,NCAL'
**
** PC.DBRMLIB   DD DSN=DSN!!0.DBRMLIB.DATA(DSNTIR),
** DISP=SHR
**
** PC.SYSLIB    DD DSN=DSN!!0.SRCLIB.DATA,
** DISP=SHR
** DD DSN=DSN!!0.SDSNSAMP,
** DISP=SHR
**
** PC.SYSIN     DD DSN=DSN!!0.SDSNSAMP(DSNTIR),
** DISP=SHR
**
** ASM.SYSLIB   DD DSN=DSN!!0.SDSNSAMP,
** DISP=SHR
** DD DSN=SYS1.MACLIB,DISP=SHR
**
** LKED.SYLMOD DD DSN=DSN!!0.RUNLIB.LOAD(DSNTIR),
** DISP=SHR
**
** LKED.SYSIN   DD *
** ENTRY DSNTIR
** NAME DSNTIR(R)
**
** STEP 2: PREPARE TELEPHONE PROGRAM
**
** PH02FS02 EXEC DSNHFOR,MEM=DSN8BF3,
** COND=(4,LT),
** PARM.PC='HOST(FORTRAN),SOURCE,XREF,STDSQL(NO)',
** PARM.FORT='MAP,GOSTMT,SOURCE,XREF'
**
** PC.DBRMLIB   DD DSN=DSN!!0.DBRMLIB.DATA(DSN8BF3),
** DISP=SHR
**
** PC.SYSLIB    DD DSN=DSN!!0.SRCLIB.DATA,
** DISP=SHR
** DD DSN=DSN!!0.SDSNSAMP,
** DISP=SHR
**
** PC.SYSIN     DD DSN=DSN!!0.SDSNSAMP(DSN8BF3),
** DISP=SHR
**
** LKED.SYLMOD DD DSN=DSN!!0.RUNLIB.LOAD,
** DISP=SHR
**
** LKED.SYSIN   DD *
** INCLUDE SYSLIB(DSNHFT)
**
** STEP 3: BIND AND RUN PROGRAM
**
** PH02FS03 EXEC PGM=IKJEFT01,DYNAMNBR=20,COND=(4,LT)
** DBRMLIB DD DSN=DSN!!0.DBRMLIB.DATA,DISP=SHR
** SYSTSPRT DD SYSOUT=* 
** FT06F001 DD SYSOUT=* 
** SYSUDUMP DD SYSOUT=* 
** SYSPRINT DD SYSOUT=* 
** SYSSIN DD *
**
** SET CURRENT SQLID = 'SYSADM';
** GRANT BIND, EXECUTE ON PLAN DSN8BF!!
** TO PUBLIC;
**
** SYSSIN DD *
** DSN SYSTEM(DSN)
** BIND PACKAGE(DSN8BF!!) MEMBER(DSN8BF3) APPLCOMPAT(V!!R1) +
** ACT(REP) ISO(CS) CURRENTDATA(YES) ENCODING(EBCDIC)
** BIND PLAN(DSN8BF!!) PKLIST(DSN8BF!!.*|)
** ACT(REPLACE) RETAIN +
** ISO(CS) CURRENTDATA(YES) ENCODING(EBCDIC)
** RUN PROGRAM(DSN8BF3) PLAN(DSN8BF!!) -
** LIB('DSN!!0.RUNLIB.LOAD')
RUN PROGRAM(DSNTIAD) PLAN(DSNTIA!!) -
LIB('DSN!0.RUNLIB.LOAD')
END

//FT05F001 DD *
L+LJ%
LXSON
LSMITH
LBROWN ALAN
LBROWN DAVID
U 0002304265

Related reference
“Sample applications in TSO” on page 1027
A set of Db2 sample applications run in the TSO environment.

**DSNTEJ3C**

THIS JCL PERFORMS THE PHASE 3 SETUP FOR THE SAMPLE APPLICATIONS AT SITES WITH COBOL.

```alma
//**************************************************************
// NAME = DSNTEJ3C
//**************************************************************
// DESCRIPTIVE NAME = DB2 SAMPLE APPLICATION
// PHASE 3
// COBOL, ISPF, CAF
// LICENSED MATERIALS - PROPERTY OF IBM
// 5650-DB2
// (C) COPYRIGHT 1982, 2016 IBM Corp. All Rights Reserved.
// STATUS = Version 12
// FUNCTION = THIS JCL PERFORMS THE PHASE 3 SETUP FOR THE SAMPLE
// APPLICATIONS AT SITES WITH COBOL. IT PREPARES THE
// COBOL ISPF CAF TELEPHONE APPLICATION AND THE REMOTE
// COBOL ORGANIZATION APPLICATION.
// NOTE: DDF MUST BE UP FOR STEP PH03CS06 TO EXECUTE
// RUN THIS JOB ANYTIME AFTER PHASE 2.
//**************************************************************
// JOBLIB DD DSN=DSN!!0.SDSNLOAD,DISP=SHR
//**************************************************************
// STEP 1: PREPARE THE COBOL CAF INTERFACE
// PH03CS01 EXEC DSNHICOB, MEM=DSNBCC,
// COND=(4,LT),
// PARM.PC=('HÔST(IBMCOB)',APOST,APOSTSQL,SOURCE,
// NOXREF,'SQL(DB2)','DEC(31)'),
// PARM.COB=(NOSEQUENCE,QUOTE,RENT,'PGMNAME(LONGUPPER)')
// PC.DBRMLIB DD DSN=DSN!!0.DBRMLIB.DATA(DSNBCC),
// DISP=SHR
// PC.SYSLIB DD DSN=DSN!!0.SRCLIB.DATA,
// DISP=SHR
// PC.SYSIN DD DSN=DSN!!0.SRSLIB.SDSNSAMP(DSNBCC),
// DISP=SHR
// LKED.SYSLMOD DD DSN=DSN!!0.RUNLIB.LOAD(DSNBCC),
// DISP=SHR
// LKED.SYSIN DD *
// INCLUDE SYSLIB(DSNALI)
//**************************************************************
// STEP 2: PREPARE THE CONNECTION MANAGER
// PH03CS02 EXEC DSNHICOB, MEM=DSN8SCM,
// COND=(4,LT),
// PARM.PC=('HÔST(IBMCOB)',APOST,APOSTSQL,SOURCE,
// NOXREF,'SQL(DB2)','DEC(31)'),
// PARM.COB=(NOSEQUENCE,QUOTE,RENT,'PGMNAME(LONGUPPER)')
```
//PC.DBRMLIB DD DSN=DSN!!0.DBRMLIB.DATA(DSN8SCM), DISP=SHR
//PC.SYSLIB DD DSN=DSN!!0.SRCLIB.DATA, DISP=SHR
//PC.SYSIN DD DSN=DSN!!0.SDSNSAMP(DSN8SCM), DISP=SHR
//LKED.SYSLMOD DD DSN=DSN!!0.RUNLIB.LOAD(DSN8SCM), DISP=SHR
//
/*
*   STEP  3: PREPARE THE TELEPHONE APPLICATION
*
*/
//PH03CS03 EXEC DSNHICOB,MEM=DSN8SC3,
//    COND=(4,LT),
//    PARM.PC=('HOST(IBMCOB)',APOST,APOSTSQL,SOURCE,
//         NOXREF,'SQL(DB2)','DEC(31)'),
//    PARM.COB=(NOSEQUENCE,QUOTE,RENT,'PGMNAME(LONGUPPER)')
//PC.DBRMLIB DD DSN=DSN!!0.DBRMLIB.DATA(DSN8SC3), DISP=SHR
//PC.SYSLIB DD DSN=DSN!!0.SRCLIB.DATA, DISP=SHR
//PC.SYSIN DD DSN=DSN!!0.SDSNSAMP(DSN8SC3), DISP=SHR
//LKED.SYSLMOD DD DSN=DSN!!0.RUNLIB.LOAD(DSN8SC3), DISP=SHR
//LKED.RUNLIB DD DSN=DSN!!0.RUNLIB.LOAD,
//    DISP=SHR
//LKED.SYSIN DD *
//   INCLUDE SYSLIB(DSNALI)
//   INCLUDE RUNLIB(DSN8MC0)
//
/*
*   STEP  4: PREPARE THE REMOTE ORGANIZATION APPLICATION
*
*/
//PH03CS04 EXEC DSNHICOB,MEM=DSN8HC3,
//    COND=(4,LT),
//    PARM.PC=('HOST(IBMCOB)',APOST,APOSTSQL,SOURCE,
//         NOXREF,'SQL(DB2)','DEC(31)'),
//    PARM.COB=(NOSEQUENCE,QUOTE,RENT,'PGMNAME(LONGUPPER)')
//PC.DBRMLIB DD DSN=DSN!!0.DBRMLIB.DATA(DSN8HC3), DISP=SHR
//PC.SYSLIB DD DSN=DSN!!0.SRCLIB.DATA, DISP=SHR
//PC.SYSIN DD DSN=DSN!!0.SDSNSAMP(DSN8HC3), DISP=SHR
//LKED.SYSLMOD DD DSN=DSN!!0.RUNLIB.LOAD(DSN8HC3),
//    DISP=SHR
//LKED.RUNLIB DD DSN=DSN!!0.RUNLIB.LOAD,
//    DISP=SHR
//LKED.SYSIN DD *
//   INCLUDE SYSLIB(DSNALI)
//   INCLUDE RUNLIB(DSN8MC0)
//
/*
*   STEP  5: BIND THE TELEPHONE APPLICATION PROGRAM
*
*/
//PH03CS05 EXEC PGM=IKJEFT01,DYNAMNBR=20,COND=(4,LT)
//DBRMLIB DD DSN=DSN!!0.DBRMLIB.DATA, DISP=SHR
//SYSTSRSPT DD SYSOUT=*
//SYSPRINT DD SYSOUT=*
//SYSUDUMP DD SYSOUT=*
//SYSIN DD *
//   SET CURRENT SQLID = 'SYSADM';
//   GRANT BIND, EXECUTE ON PLAN DSN8SC!! TO PUBLIC;
//SYSTSIN DD *
//DSN SYSTEM(DSN)
//BIND PACKAGE(DSN8SC!!) MEMBER(DSN8SC3) APPLCOMPAT(V!!R1) +
//   ACT(REP) ISO(CS) CURRENTDATA(YES) ENCODING(EBCDIC)
//BIND PLAN(DSN8SC!!) PKLIST(DSN8SC!!) +
//   ACTION(REPLACE) RETAIN +
//   ISO(CS) CURRENTDATA(YES) ENCODING(EBCDIC)
//RUN PROGRAM(DSNTIAD) PLAN(DSNTIA!!) -
//   LIB('DSN!!0.RUNLIB.LOAD')
//
/*
*   STEP  6: BIND THE REMOTE ORGANIZATION APPLICATION
*
*/
//PH03CS06 EXEC PGM=IKJEFT01,DYNAMNBR=20,COND=(4,LT)
//DBRMLIB DD DSN=DSN!!0.DBRMLIB.DATA, DISP=SHR
//SYSTSRSPT DD SYSOUT=*
//SYSPRINT DD SYSOUT=*
//SYSUDUMP DD SYSOUT=*
//SYSOUT DD SYSOUT=*
//SYSTSIN DD *
Related reference
“Sample applications in TSO” on page 1027
A set of Db2 sample applications run in the TSO environment.

DSNTEJ6

RUN THIS JOB AT THE REMOTE LOCATION TO UPDATE THE SAMPLE LOCATION IN DEPARTMENT TABLE
RUN THIS JOB ANYTIME AFTER PHASE 3.

Related reference
“Sample applications in TSO” on page 1027
A set of Db2 sample applications run in the TSO environment.

**DSNTEJ3P**

This JCL performs the Phase 3 setup for the sample applications at sites with PL/I.

```plaintext
//********************************************************************
//*  NAME = DSNTEJ3P
//*  DESCRIPTIVE NAME = DB2 SAMPLE APPLICATION                              
//*                     PHASE 3                                        
//*                     PL/I, ISPF, CAF                                  
//*  LICENSED MATERIALS - PROPERTY OF IBM                                
//*  5650-DB2                                                          
//*  (C) COPYRIGHT 1982, 2016 IBM CORP.  ALL RIGHTS RESERVED.           
//*  STATUS = VERSION 12                                                
//*  FUNCTION = THIS JCL PERFORMS THE PHASE 3 SETUP FOR THE SAMPLE       
//*             APPLICATIONS AT SITES WITH PL/I.  IT PREPARES THE         
//*             PL/I ISPF CAF TELEPHONE PROGRAM.                         
//*  CHANGE ACTIVITY =                                                 
//*    08/18/2014 Single-phase migration s21938_inst1 s21938            
//********************************************************************

//JOBLIB  DD  DISP=SHR,DSN=DSN!!0.SDSNLOAD
//        DD  DISP=SHR,DSN=CEE.V!R!M!.SCEERUN

//* STEP  1: PREPARE THE ISPF CAF CONNECTION MANAGER                   
PH03PS01 EXEC DSNHPLI,MEM=DSN8SPM,
   COND=(4,LT),
   PARM.PC=('HOST(PLI),CCSID(37),SOURCE,XREF,STDSQL(NO)'),
   PARM.PLI=('NOPT,SOURCE,OBJECT,MARGINS(2,72,0)',
   'LIMITS(EXTNAME(7)),OPTIONS')
PPLI.SYSIN  DD DSN=DSN!!0.SDSNSAMP(DSN8SPM),
            DISP=SHR
PC.DBRMLIB  DD DSN=DSN!!0.DBRMLIB.DATA(DSN8SPM),
            DISP=SHR
PC.SYSLIB   DD DSN=DSN!!0.SRCLIB.DATA,
            DISP=SHR
LKED.SYSLMOD DD DSN=DSN!!0.RUNLIB.LOAD(DSN8SPM),
               DISP=SHR
LKED.SYSIN   DD *
ENTRY CEESTART

//* STEP  2: PREPARE THE ISPF CAF TELEPHONE APPLICATION                
PH03PS02 EXEC DSNHPLI,MEM=DSN8SP3,
   COND=(4,LT),
   PARM.PC=('HOST(PLI),CCSID(37),SOURCE,XREF,STDSQL(NO)'),
   PARM.PLI=('NOPT,SOURCE,OBJECT,MARGINS(2,72,0)',
   'LIMITS(EXTNAME(7)),OPTIONS')
PPLI.SYSIN  DD DSN=DSN!!0.SDSNSAMP(DSN8SP3),
            DISP=SHR
PC.DBRMLIB  DD DSN=DSN!!0.DBRMLIB.DATA(DSN8SP3),
            DISP=SHR
PC.SYSLIB   DD DSN=DSN!!0.SRCLIB.DATA,
            DISP=SHR
LKED.SYSLMOD DD DSN=DSN!!0.RUNLIB.LOAD(DSN8SP3),
               DISP=SHR
LKED.RUNLIB DD DSN=DSN!!0.RUNLIB.LOAD,
              DISP=SHR
LKED.SYSIN   DD *
   INCLUDE SYSLIB(DSNALI)
   INCLUDE RUNLIB(DSN8MPG)

//* STEP  3: BIND THE ISPF CAF TELEPHONE APPLICATION                   
PH03PS03 EXEC PGM=IKJEFT01,DYNAMNBR=20,COND=(4,LT)
  /DBRMLIB DD DISP=SHR,DSN=DSN!!0.DBRMLIB.DATA
  //SYSTSPRT DD SYSOUT=* 
  //SYSPRINT DD SYSOUT=* 
  //SYSDUMP DD SYSOUT=* 
```

1298  Db2 12 for z/OS: Application Programming and SQL Guide
Related reference

“Sample applications in TSO” on page 1027
A set of Db2 sample applications run in the TSO environment.

**DSNTEJ2A**

THIS JCL PERFORMS THE PHASE 2 SETUP FOR THE SAMPLE APPLICATIONS.

```sql
//SYSIN DD *
SET CURRENT SQLID = 'SYSADM';
GRANT BIND, EXECUTE ON PLAN DSN8SP!! TO PUBLIC;
//SYSTSIN DD *
DSN SYSTEM(DSN)
BIND PACKAGE(DSN8SP!!) MEMBER(DSN8SP3) APPLCOMPAT(V11R1) +
ACT(REP) ISO(CS) CURRENTDATA(YES) ENCODING(EBCDIC)
BIND PLAN(DSN8SP!!) PKLIST(DSN8SP!!.*) +
ACTION(REPLACE) RETAIN +
ISO(CS) CURRENTDATA(YES) ENCODING(EBCDIC)
RUN PROGRAM(DSNTIAD) PLAN(DSNTIA!!) -
LIB('DSN!0.RUNLIB.LOAD')
END
/**

Related reference

“Sample applications in TSO” on page 1027
A set of Db2 sample applications run in the TSO environment.

**DSNTEJ2A**

THIS JCL PERFORMS THE PHASE 2 SETUP FOR THE SAMPLE APPLICATIONS.

```
/* BIND THE UNLOAD PROGRAM AND GRANT EXECUTE AUTHORITY */

BINDUNL EXEC PGM=IKJEFT01, DYNAMNBR=20, COND=(4,LT)
//DBRMLIB DD DSN=DSN!!0.DBRMLIB.DATA,
// DISP=SHR
//SYSIN DD SYSOUT=* 
//SYSUDEPTH DD SYSOUT=* 
//SYSIN DD * 
DSN SYSTEM(DSN)
BIND PACKAGE(DSNTIB!!) MEM(DSNTIAUL) APPLCOMPAT(V!!R1) +
CURRENTDATA(NO) ACT(REP) ISO(CS) ENCODING(EBCDIC) +
BIND PLAN(DSNTIB!!) PKLIST(DSNTIB!!) +
ACTION(REPLACE) RETAIN +
CURRENTDATA(NO) ISO(CS) ENCODING(EBCDIC)
RUN PROGRAM(DSNTIA) PLAN(DSNTIB!!) +
LIB('DSN!!0.RUNLIB.LOAD')
END

/* SET CURRENT SQLID = 'SYSADM'; 
GRANT EXECUTE ON PLAN DSNTIB!! 
TO PUBLIC; */

/* DELETE DATA SETS, DROP TABLES TO ALLOW RERUNS */

DELETE EXEC PGM=IKJEFT01, DYNAMNBR=20, COND=(4,LT)
//SYSIN DD SYSOUT=* 
//SYSIN DD * 
DELETE 'DSN!!0.DSN8UNLD.SYSREC00'
DELETE 'DSN!!0.DSN8UNLD.SYSREC01'
DELETE 'DSN!!0.DSN8UNLD.SYSPUNCH'
DSN SYSTEM(DSN)
RUN PROGRAM(DSNTIA) PLAN(DSNTIA!!) -
LIB('DSN!!0.RUNLIB.LOAD')
//SYSPRINT DD SYSOUT=* 
//SYSIN DD * 
SET CURRENT SQLID = 'SYSADM';
DROP TABLE DSN8!!0.NEWDEPT ;
DROP TABLE DSN8!!0.NEWPHONE ;
DROP DATABASE DSN8D!!U ;
DROP STOGROUP DSN8G!!U ;
COMMIT;

/* CREATE NEW TABLES */

CREATE EXEC PGM=IKJEFT01, DYNAMNBR=20, COND=(4,LT)
//SYSIN DD SYSOUT=* 
//SYSIN DD * 
DSN SYSTEM(DSN)
RUN PROGRAM(DSNTIA) PLAN(DSNTIA!!) -
LIB('DSN!!0.RUNLIB.LOAD')
//SYSPRINT DD SYSOUT=* 
//SYSUDEPTH DD SYSOUT=* 
//SYSIN DD * 
SET CURRENT SQLID = 'SYSADM';
CREATE STOGROUP DSN8G!!U VOLUMES (DSNV01) VCAT DNSC!!0;
CREATE DATABASE DSN8D!!U STOGROUP DSN8G!!U CCSID EBCDIC;

CREATE TABLE DSN8!!0.NEWDEPT
(DEPTNO CHAR(3) NOT NULL,
DEPTNAME VARCHAR(36) NOT NULL,
MGRNO CHAR(6) ,
ADMREPT CHAR(3) NOT NULL,
LOCATION CHAR(16))
IN DATABASE DSN8D!!U CCSID EBCDIC;

CREATE TABLE DSN8!!0.NEWPHONE
(LASTNAME VARCHAR(15) NOT NULL,
FIRSTNAME VARCHAR(12) NOT NULL,
MIDDLEINITIAL CHAR(1) NOT NULL,
PHONENUMBER CHAR(4) ,
EMPLOYEECHAR(6) NOT NULL,
DEPTNUMBER CHAR(3) NOT NULL,
DEPTNAME VARCHAR(36) NOT NULL )
IN DATABASE DSN8D!!U
CCSID EBCDIC;

/*
 * RUN UNLOAD PROGRAM
 */
/*
* UNLOAD EXEC PGM=IKJEFT01,DYNAMNBR=20,COND=(4,LT)
* SYSTSPRT DD SYSOUT=*  
* SYSTSPRT DD SYSOUT=*  
* SYSREC00 DD DSN=DSN8!!0.DSN8UNLD.SYSREC00, 
*   DISP=(,CATLG), 
*   UNIT=SYSDA, 
*   SPACE=(1024,(10,10))
* SYSREC01 DD DSN=DSN8!!0.DSN8UNLD.SYSREC01, 
*   DISP=(,CATLG), 
*   UNIT=SYSDA, 
*   SPACE=(1024,(10,10))
* SYSPUNCH DD DSN=DSN8!!0.DSN8UNLD.SYSPUNCH, 
*   DISP=(,CATLG), 
*   UNIT=SYSDA, 
*   SPACE=(800,(15,15))
* SYSPRINT DD SYSOUT=*  
* SYSDUMP DD SYSOUT=*  
SET CURRENT SQLID = 'SYSADM';
LOCK TABLE DSN8!!0.DEPT IN SHARE MODE;
SELECT * FROM DSN8!!0.DEPT;
SELECT * FROM DSN8!!0.VPHONE;

//*
/* EDIT THE OUTPUT FROM THE PROGRAM
/*****
* EDIT EXEC PGM=IKJEFT01,DYNAMNBR=20,COND=((4,LT),(4,LE,UNLOAD))
* SYSTSPRT DD SYSOUT=*  
* SYSTSPRT DD SYSOUT=*  
EDIT 'DSN8!!0.DSN8UNLD.SYSPUNCH' DATA NONUM 
CHANGE * 30 /DSN8!!0.DEPT/DSN8!!0.NEWDEPT/ 
CHANGE * 30 /DSN8!!0.VPHONE/DSN8!!0.NEWPHONE/  
TOP 
LIST = 999 
END SAVE
/****
* RUN LOAD UTILITY TO LOAD TABLES
/*****
* LOAD EXEC DSNUPROC,PARM='DSN,DSNTEX', 
*   COND=((4,LT),(4,LE,UNLOAD))
* DSNTRACE DD SYSOUT=*  
* SORTLIB DD DSN=SYS1.SORTLIB,DISP=SHR 
* SORTWK01 DD UNIT=SYSDA,SPACE=(4000,(20,20),,,ROUND) 
* SORTWK02 DD UNIT=SYSDA,SPACE=(4000,(20,20),,,ROUND) 
* SORTWK03 DD UNIT=SYSDA,SPACE=(4000,(20,20),,,ROUND) 
* SORTWK04 DD UNIT=SYSDA,SPACE=(4000,(20,20),,,ROUND) 
* SYSREC00 DD DSN=DSN8!!0.DSN8UNLD.SYSREC00, 
*   DISP=(OLD,KEEP) 
* SYSREC01 DD DSN=DSN8!!0.DSN8UNLD.SYSREC01, 
*   DISP=(OLD,KEEP) 
* SYSUT1 DD UNIT=SYSDA,SPACE=(4000,(20,20),,,ROUND) 
* SYSIN DD DSN=DSN8!!0.DSN8UNLD.SYSREC00, 
*   DISP=(OLD,KEEP) 
/****

Related reference
“Sample applications in TSO” on page 1027
A set of Db2 sample applications run in the TSO environment.

DSNTEJ1P
THIS JCL PERFORMS THE PHASE 1 SETUP FOR SAMPLE APPLICATIONS AT SITES WITH PL/I.

/*******************************
* NAME = DSNTEJ1P
*******************************
* DESCRIPTIVE NAME = DB2 SAMPLE APPLICATION
* PHASE 1
* PL/I
***/
FUNCTION = THIS JCL PERFORMS THE PHASE 1 SETUP FOR SAMPLE APPLICATIONS AT SITES WITH PL/I.

CHANGE ACTIVITY = 08/18/2014 Single-phase migration

PH01PS01 EXEC DSNHPLI, MEM=DSNTEP2, PARM.PC=('HOST(PLI),CCSID(37),STDSQL(NO),CONNECT(2)', 'TPWOPASS', 'VERSION(AUTO)'), PARM.PLI=('NOPT','MAR(2,72,0)', 'GS,OBJ,S', 'LIMITS(FIXDEDBIN(31,63))', 'LANGNLVL(SPROG)', 'OFFSET')

PPLI.SYSIN DD DSN=DSN!!0.SDSNSAMP(DSNTEP2), DISP=SHR

PC.DBRMLIB DD DSN=DSN!!0.DBRMLIB.DATA(DSNTEP2), DISP=SHR

PC.SYSLIB DD DSN=DSN!!0.SRCLIB.DATA, DISP=SHR

LKED.SYSLMOD DD DSN=DSN!!0.RUNLIB.LOAD(DSNTEP2), DISP=SHR

LKED.SYSLIN DD *

INCLUDE SYSLIB(DSNELI)

PH01PS02 EXEC PG=IKJEFT01, DYNAMNBR=20, COND=(4,LT)

DBRMLIB DD DSN=DSN!!0.DBRMLIB.DATA, DISP=SHR

SYSTSPRT DD SYSOUT=

SYSPRINT DD SYSOUT=

SYSUDUMP DD SYSOUT=

SYSTSIN DD *

DSN SYSTEM(DSN)

BIND PACKAGE (DSNTEP2) MEMBER(DSNTEP2) APPLCOMPAT(V!!R1) +
CURRENTDATA(NO) ACT(REP) ISO(CS) ENCODING(EBCDIC)

BIND PLAN(DSNTEP2) PKLIST(DSNTEP2.*) +
ACTION(REPLACE) RETAIN +
CURRENTDATA(NO) ISO(CS) ENCODING(EBCDIC) SQLRULES(DB2)

RUN PROGRAM(DSNTEP2) PLAN(DSNTEP2) +
LIB('DSN!!0.RUNLIB.LOAD') +
PARMS('/ALIGN(MID)')

SY SIN DD *

SET CURRENT SQLID = 'SYSADM';

GRANT EXECUTE, BIND ON PLAN DSNTEP2 TO PUBLIC;

SELECT EMPNO, FIRSTNAME, MIDDLE, LASTNAME,
WORKDEPT, PHONE, HIREDATE, JOB, EDLEVEL,
SEX, BIRTHDATE, SALARY, BONUS, COMM,
SALARY+BONUS+COMM AS TOTAL_SALARY
FROM EMP
ORDER BY TOTAL_SALARY;

SELECT * FROM DEPT;

SELECT * FROM ACT;

SELECT * FROM EMPPROJECT;

SELECT * FROM PROJECT;

SELECT * FROM EMPPROJECT2;

SYSTSIN DD *

PH01PS03 EXEC DSNHPLI, MEM=DSNTEP4, COND=(4,LT),
PARM.PC=('HOST(PLI),CCSID(37),STDSQL(NO),CONNECT(2)', 'TPWOPASS', 'VERSION(AUTO)'),
PARM.PLI=('NOPT','MAR(2,72,0)', 'GS,OBJ,S', 'LIMITS(FIXDEDBIN(31,63))', 'LANGNLVL(SPROG)', 'OFFSET')

PPLI.SYSIN DD DSN=DSN!!0.SDSNSAMP(DSNTEP4), DISP=SHR

PC.DBRMLIB DD DSN=DSN!!0.DBRMLIB.DATA(DSNTEP4), DISP=SHR
```
//PC.SYSLIB DD DSN=DSN!!0.SRCLIB.DATA,
// DISP=SHR
//LKED.SYSLMOD DD DSN=DSN!!0.RUNLIB.LOAD(DSNTEP4),
// DISP=SHR
//LKED.SYSIN DD *,
// INCLUDE SYSLIB(DSNELI)

/*
 *        STEP 4 : BIND AND RUN PROGRAM DSNTEP4, TO
 *                 PRINT THE TABLES
 *PH01PS04 EXEC PGM=IKJEFT01,DYNAMNBR=20,COND=(4,LT)
//DBRMLIB DD DSN=DSN!!0.DBRMLIB.DATA,
// DISP=SHR
//SYSTSPRT DD SYSOUT=*
//SYSPRINT DD SYSOUT=*
//SYSUDUMP DD SYSOUT=*
//SYSTSIN DD *
DSN SYSTEM(DSN)
BIND PACKAGE (DSNTEP4) MEMBER(DSNTEP4) APPLCOMPAT(V!!R1) +
CURRENTDATA(NO) ACT(REP) ISO(CS) ENCODING(EBCDIC)
BIND PLAN(DSNTP4!!)  PKLIST(DSNTEP4.*) +
ACTION(REPLACE) RETAIN +
CURRENTDATA(NO) ISO(CS) ENCODING(EBCDIC) SQLRULES(DB2)
RUN  PROGRAM(DSNTEP4) PLAN(DSNTP4!!) +
LIB('DSN!!0.RUNLIB.LOAD') +
PARMS('/ALIGN(MID)')
END

//SYSIN DD *
SET CURRENT SQLID = 'SYSADM';
GRANT EXECUTE, BIND ON PLAN DSNTP4!!
TO PUBLIC;
SELECT EMPNO, FIRSTNME, MIDINIT, LASTNAME,
WORKDEPT, PHONENO, HIREDATE, JOB, EDLEVEL,
SEX, BIRTHDATE, SALARY, BONUS, COMM,
SALARY+BONUS+COMM AS TOTAL_SALARY
FROM EMP
ORDER BY TOTAL_SALARY;
SELECT * FROM DEPT;
SELECT * FROM ACT;
SELECT * FROM EMPPROJACT;
SELECT * FROM PROJ;
SELECT * FROM PROJACT;

Related reference
“Sample applications in TSO” on page 1027
A set of Db2 sample applications run in the TSO environment.

DSNTEJ1L
THIS JCL CREATES THE DSNTEP2 LOAD MODULE FROM THE SHIPPED OBJECT DECK, DSNTEP2L, AND
LINKS THE PACKAGE AND PLAN FOR THIS VERSION OF DSNTEP2.

polator='DSNTEJ1L)
# Licensed Materials - Property of IBM
# 5650-DB2
# (C) COPYRIGHT 1982, 2016 IBM Corp. All Rights Reserved.
# STATUS = Version 12
# FUNCTION = THIS JCL CREATES THE DSNTEP2 LOAD MODULE FROM THE
# SHIPPED OBJECT DECK, DSNTEP2L, AND LINKS THE
# PACKAGE AND PLAN FOR THIS VERSION OF DSNTEP2.
# = THIS JCL ALSO CREATES THE DSNTEP4 LOAD MODULE FROM
# THE SHIPPED OBJECT DECK, DSNTEP4L, AND LINKS THE
# PACKAGE AND PLAN FOR THIS VERSION OF DSNTEP4.
# THIS JOB IS RUN AFTER DSNTEJ1.
# NOTE: IF YOU RUN THIS JOB, YOU DO NOT NEED TO RUN THE SAMPLE
# JOB DSNTEJ3P EXCEPT TO PREPARE CUSTOMIZED VERSIONS OF
```
THE DSNTEP2 AND DSNTEP4 SOURCE CODE (YOU NEED A PL/I COMPLER TO RUN DSNTEJ1P SUCCESSFULLY).

CHANGE ACTIVITY = 08/18/2014 Single-phase migration s21938_inst1 s21938

STEP 1 : CREATE DSNTEP2 LOADMOD FROM DSNTEP2L OBJECT DECK

PH01LS01 EXEC PGM=IEWL,PARM='XREF'
SYSLIB DD DISP=SHR,DSN=CEE.V1R1M1.SCEELKED
SDSNSAMP DD DISP=SHR,DSN=DSN!0.SDNSLOAD
SYSLMOD DD DISP=SHR,DSN=DSN!0.SDNSAMP(DSNTEP2L)
SYSUDUMP DD SYSOUT=*
SYSUT1 DD UNIT=SYSDA,SPACE=(1024,(50,50))
SYSLIN DD *
INCLUDE SDSNSAMP(DSNTEP2L)
INCLUDE SYSLIB(DSNELI)
NAME DSNTEP2(R)

STEP 2 : BIND AND RUN PROGRAM DSNTEP2, TO PRINT THE TABLES

PH01LS02 EXEC PGM=IKJEFT01,DYNAMNBR=20,COND=(4,LT)
DBRMLIB DD DISP=SHR,DSN=DSN!0.SDNSAMP
SYSTSPRT DD SYSOUT=*
SYSUDUMP DD SYSOUT=*
SYSUT1 DD UNIT=SYSDA,SPACE=(1024,(50,50))
SYSLIN DD *
DSN SYSTEM(DSN)
BIND PACKAGE (DSNTEP2) MEMBER(DSN@EP2L) APPLCOMPAT(V1R1) + CURRENTDATA(NO) ACT(REP) ISO(CS) ENCODING(EBCDIC)
BIND PLAN(DSNTEP!!) PKLIST(DSNTEP2.* ) + ACTION(REPLACE) RETAIN + CURRENTDATA(NO) ISO(CS) ENCODING(EBCDIC) SQLRULES(DB2)
RUN PROGRAM(DSNTEP2) PLAN(DSNTEP!!) + LIB('DSN!0.RUNLIB.LOAD') + PARMS('ALIGN(MID)')

SYSIN DD *
SET CURRENT SQLID = 'SYSADM';
GRANT EXECUTE, BIND ON PLAN DSNTEP!! TO PUBLIC;
SELECT EMPNO, FIRSTNME, MIDINIT, LASTNAME, WORKDEPT, PHONENO, HIREDATE, JOB, EDLEVEL, SEX, BIRTHDATE, SALARY, BONUS, COMM, SALARY+BONUS+COMM AS TOTAL_SALARY
FROM EMP
ORDER BY TOTAL_SALARY;
SELECT * FROM DEPT;
SELECT * FROM ACT;
SELECT * FROM EMPPROJACT;
SELECT * FROM PROJ;
SELECT * FROM PROJACT;

STEP 3 : CREATE DSNTEP4 LOADMOD FROM DSNTEP4L OBJECT DECK

PH01LS03 EXEC PGM=IEWL,COND=(4,LT),PARM='XREF'
SYSLIB DD DISP=SHR,DSN=CEE.V1R1M1.SCEELKED
DD DISP=SHR,DSN=DSN!0.SDNSLOAD
SDSNSAMP DD DISP=SHR,DSN=DSN!0.SDNSAMP(DSNTEP4L)
SYSLMOD DD DISP=SHR,DSN=DSN!0.RUNLIB.LOAD(DSNTEP4)
SYSUDUMP DD SYSOUT=*
SYSUT1 DD UNIT=SYSDA,SPACE=(1024,(50,50))
SYSLIN DD *
INCLUDE SDSNSAMP(DSNTEP4L)
INCLUDE SYSLIB(DSNELI)
NAME DSNTEP4(R)

STEP 4 : BIND AND RUN PROGRAM DSNTEP4, TO PRINT THE TABLES

PH01LS04 EXEC PGM=IKJEFT01,DYNAMNBR=20,COND=(4,LT)
DBRMLIB DD DISP=SHR,DSN=DSN!0.SDNSAMP
SYSTSPRT DD SYSOUT=*

1304 Db2 12 for z/OS: Application Programming and SQL Guide
//SYSPRINT DD SYSOUT=*
//SYSDUMP DD SYSOUT=*
//SYSTSIN DD *

DSN SYSTEM(DSN)
BIND PACKAGE (DSNTEP4) MEMBER(DSN@EP4L) APPLCOMPAT(V!R1) +
CURRENTDATA(NO) ACT(REP) ISO(CS) ENCODING(EBCDIC)
BIND PLAN(DSNTP4!!) PKLIST(DSNTEP4.*) +
ACTION(REPLACE) RETAIN +
CURRENTDATA(NO) ISO(CS) ENCODING(EBCDIC) SQLRULES(DB2)
RUN PROGRAM(DSNTEP4) PLAN(DSNTP4!!) +
LIB('DSN!!0.RUNLIB.LOAD') +
PARMS('/ALIGN(MID)')
END

//SYSIN DD *
SET CURRENT SQLID = 'SYSADM';
GRANT EXECUTE, BIND ON PLAN DSNTP4!!
TO PUBLIC;
SELECT EMPNO, FIRSTNME, MIDINIT, LASTNAME,
WORKDEPT, PHONENO, HIREDATE, JOB, EDLEVEL,
SEX, BIRTHDATE, SALARY, BONUS, COMM,
SALARY+BONUS+COMM AS TOTAL_SALARY
FROM EMP
ORDER BY TOTAL_SALARY;
SELECT * FROM DEPT;
SELECT * FROM ACT;
SELECT * FROM EMPPROJACT;
SELECT * FROM PROJ;
SELECT * FROM PROJACT;

Related reference
“Sample applications in TSO” on page 1027
A set of Db2 sample applications run in the TSO environment.

DSNTEJ6P

THIS JCL EXECUTES THE PHASE 6 STORED PROCEDURE SAMPLE APPLICATION.

//*****************************************************************************/
/* NAME = DSNTEJ6P */
/* DESCRIPTIVE NAME = Db2 SAMPLE APPLICATION */
/* PHASE 6 */
/* LICENSED MATERIALS - PROPERTY OF IBM */
/* 5650-D62 */
/* (C) COPYRIGHT 1982, 2016 IBM CORP. ALL RIGHTS RESERVED. */
/* STATUS = VERSION 12 */
/* FUNCTION = THIS JCL EXECUTES THE PHASE 6 STORED PROCEDURE */
/* SAMPLE APPLICATION. */
/* DEPENDENCIES: */
/* (1) RUN SAMPLE JOB DSNTEJ6S AT THE SERVER SITE BEFORE RUNNING THIS */
/* JOB; DSNTEJ6S PREPARES THE SAMPLE STORED PROC W/O RESULT SET */
/* (2) RUN THIS JOB AT THE CLIENT SITE */
/* CHANGE ACTIVITY = */
/* 08/18/2014 Single-phase migration s21938_inst1 s21938 */
//*****************************************************************************/

JOBLIB DD DISP=SHR,DSN=DSN!0.SDSNEXIT
// DD DISP=SHR,DSN=DSN!0.SDSNLOAD
// DD DISP=SHR,DSN=CEE.V!R!M!.SCERUN
// DD DISP=SHR,DSN=DSN!0.RUNLIB.LOAD

*******************************************************************************/
//STEP 1: PRE-COMPILE, COMPILe, AND LINK-EDIT THE CALLING PROGRAM
*******************************************************************************/

//PH60PS01 EXEC DSNHPLI,MEM=DSN8EPI,
// PARM.PC='HOST(PLI),CCSID(37),STDSQL(NO),CONNECT(2)'
//PPLI.SYSIN DD DSN=DSN!!0.DBRMLIB.LOAD(DSN!8EPI),
// DISP=SHR
//PC.DBRMLIB DD DSN=DSN!!0.DBRMLIB.LOAD(DSN!8EPI),
// DISP=SHR

Chapter 12. Sample data and applications supplied with Db2 for z/OS 1305
PC.SYSLIB DD DSN=DSN!!0.SRCLIB.DATA,
// DISP=SHR
LKED.SYSLMOD DD DSN=DSN!!0.RUNLIB.LOAD(DSN8EP1),
// DISP=SHR
LKED.SYSIN DD *
INCLUDE SYSLIB(DSNELI)
INCLUDE SYSLIB(DSNTIAR)

/**/ STEP 2: BIND THE CALLING PROGRAM PACKAGE
/**/ PH06PS02 EXEC PGM=IKJEFT01,DYNAMNBR=20,COND=(4,LT)
//DBRMLIB DD DSN=DSN!!0.DBRMLIB.DATA,
// DISP=SHR
//SYSTSPRT DD SYSOUT=* //SYSPRINT DD SYSOUT=* //SYSDUMP DD SYSOUT=* //SYSSIN DD *
SET CURRENT SQLID = 'SYSADM';
GRANT BIND, EXECUTE ON PLAN DSN8EP1
TO PUBLIC;
//SYSSIN DD *
DSN SYSTEM(DSN)
BIND PACKAGE(DSN8EP!!) MEMBER(DSN8EP1) APPLCOMPAT(V!!R1) +
ACT(REP) ISO(CS) CURRENTDATA(YES) ENCODING(EBCDIC)
BIND PACKAGE(SAMPLOC.DSN8EP!!) MEMBER(DSN8EP1) APPLCOMPAT(V!!R1) +
MEMBER(DSN8EP1) ACT(REP) ISO(CS) CURRENTDATA(YES) ENCODING(EBCDIC)
BIND PLAN(DSN8EP1) -
PKLIST(DSN8EP!!:DSN8EP1, SAMPLOC.DSN8EP!!:DSN8EP1) -
ACTION(REPLACE) RETAIN +
ISO(CS) CURRENTDATA(YES) ENCODING(EBCDIC)
RUN PROGRAM(DSNTIAD) PLAN(DSN8EP1!!) -
LIB('DSN!!0.RUNLIB.LOAD')

/**/ STEP 3: EXECUTE THE STORED PROCEDURE
/**/ PH06PS03 EXEC PGM=IKJEFT01,DYNAMNBR=20,COND=(4,LT)
//SYSSPRT DD SYSOUT=* //SYSPRINT DD SYSOUT=* //SYSSDUMP DD SYSOUT=* //SYSSIN DD *
DSN SYSTEM(DSN)
RUN PROGRAM(DSN8EP1) PLAN(DSN8EP1) PARMS('/SAMPLOC')
END
//SYSSIN DD *
-DISPLAY ARCHIVE;
-DISPLAY THREAD(*) DETAIL;
/**/

Related reference
“Sample applications in TSO” on page 1027
A set of Db2 sample applications run in the TSO environment.

** DSNTEJ6S **
THIS JCL PREPARES THE SAMPLE STORED PROCEDURE.

/**/ NAME = DSNTEJ6S
/**/ DESCRIPTIVE NAME = DB2 SAMPLE APPLICATION
/**/ PHASE 6
/**/ LICENSED MATERIALS - PROPERTY OF IBM
/**/ 5650-DB2
/**/ (C) COPYRIGHT 1982, 2016 IBM CORP. ALL RIGHTS RESERVED.
/**/ STATUS = VERSION 12
/**/ FUNCTION = THIS JCL PREPARES THE SAMPLE STORED PROCEDURE.
/**/ DEPENDENCIES:
/**/ (1) RUN THIS JOB AT THE SERVER SITE BEFORE RUNNING SAMPLE JOB
/**/ DSNTEJ6P AT THE CLIENT SITE
/**/ CHANGE ACTIVITY =
/**/ 08/18/2014 Single-phase migration s21938_inst1 s21938
Chapter 12. Sample data and applications supplied with Db2 for z/OS
Related reference
“Sample applications in TSO” on page 1027
A set of Db2 sample applications run in the TSO environment.

**DSNTEJ6D**

THIS JCL PREPARES AND EXECUTES A SAMPLE APPLICATION PROGRAM, DSN8ED1, THAT DEMONSTRATES HOW TO CALL A Db2 STORED PROCEDURE THAT RETURNS A RESULT SET.

```plaintext
*** NAME = DSNTEJ6D
*** DESRIPTIVE NAME = DB2 SAMPLE APPLICATION
*** JOB; DSNTEJ6T PREPARES THE SAMPLE STORED PROC W/ RESULT SET
*** C LANGUAGE
*** LICENSED MATERIALS - PROPERTY OF IBM
*** 5650-DB2
*** (C) COPYRIGHT 1982, 2016 IBM CORP. ALL RIGHTS RESERVED.
*** STATUS = VERSION 12
*** FUNCTION = THIS JCL PREPARES AND EXECUTES A SAMPLE APPLICATION PROGRAM, DSN8ED1, THAT DEMONSTRATES HOW TO CALL A Db2 STORED PROCEDURE THAT RETURNS A RESULT SET.
*** DSN8ED1 ACCEPTS A Db2 COMMAND FROM STANDARD INPUT (SYSIN) AND PASSES IT AS A PARAMETER TO THE STORED PROCEDURE WHICH RUNS ON A REMOTE Db2 SUBSYSTEM (SEE DSNTEJ6T FOR DETAILS). THE STORED PROCEDURE PLACES THE RESPONSES IN A RESULT SET AND DSN8ED1 EXTRACTS THEM AND PRINTS THEM TO STANDARD OUTPUT (SYSPRINT).
*** DEPENDENCIES:
*** (1) RUN SAMPLE JOB DSNTEJ6T AT THE SERVER SITE BEFORE RUNNING THIS JOB; DSNTEJ6T PREPARES THE SAMPLE STORED PROC W/ RESULT SET
*** (2) RUN THIS JOB AT THE CLIENT SITE
*** CHANGE ACTIVITY =
*** 08/18/2014 Single-phase migration s21938_inst1 s21938
***
*** JOBLIB DD DSN=DSN!0.SDSNEXIT,DISP=SHR
***   DD DSN=DSN!0.SDSNLOAD,DISP=SHR
***   DD DSN=CEE.V!R!M!.SCEERUN,DISP=SHR
***   DD DSN=DSN!0.RUNLIB.LOAD,DISP=SHR
***
*** EXEC PH06DS01 EXEC DSNHC,MEM=DSN8ED1,
***   parm.pc=('HOST(C),CCSID(1047),MARGINS(1,72),STDSQL(NO)',
***   'SOURCE,XREF'),
***   parm.c='SOURCE LIST MAR(1,72) NORENT OPTFILE(DD:CCOPTS)',
***   parm.lked='AMODE=31,RMODE=ANY,MAP,NORENT'
***   DD DSN=DSN!0.BRMLIB.DATA(DSN8ED1),
***   disp=shr
***   DD DSN=DSN!0.SRCLIB.DATA,
***   disp=shr
***   DD DSN=DSN!0.SDSSAMP(DSN8ED1),
***   disp=shr
***   DD DSN=DSN!0.RUNLIB.LOAD(DSN8ED1),
***   disp=shr
***   DD DSN=DSN!0.SYSLIB DATA,
***   disp=shr
***   DD DSN=DSN!0.SYSIN DATA,
***   disp=shr
***   DD *
```
INCLUDE SYSLIB(DSNELI)
INCLUDE SYSLIB(DSNTIAR)

ORDER 2: BIND THE CALLING PROGRAM PACKAGE
ORDER 3: EXECUTE THE STORED PROCEDURE

Related reference
“Sample applications in TSO” on page 1027
A set of Db2 sample applications run in the TSO environment.

DSNTEJ6T

THIS JCL PREPARES AND EXECUTES A SAMPLE APPLICATION PROGRAM, DSN8ED2, THAT
DEMONSTRATES A Db2 STORED PROCEDURE THAT RETURNS A RESULT SET.

ORDER 1: BIND THE CALLING PROGRAM PACKAGE
ORDER 2: BIND THE CALLING PROGRAM PACKAGE
TABLE SO THEY CAN BE RETURNED AS A RESULT SET TO THE
CLIENT.

DEPENDENCIES:
(1) RUN THIS JOB AT THE SERVER SITE BEFORE RUNNING SAMPLE JOB
DSNTEJ6D AT THE CLIENT SITE

CHANGE ACTIVITY =
08/18/2014 Single-phase migration s21938_inst1 s21938

********************************************************************
//JOBLIB DD DSN=DSN!0.SDNSEXIT,DISP=SHR
// DD DSN=DSN!0.SDNSLOAD,DISP=SHR
// DD DSN=CEE.V!R!M!.SCEErun,DISP=SHR
// DD DSN=DSN!0.RUNLIB.LOAD,DISP=SHR

********************************************************************
STEP 1: DROP OBJECTS CREATED BY ANY PREVIOUS RUN OF DSNTEJ6T:
- SAMPLE STORED PROCEDURE DSN8.DSN8ED2
- GLOBAL TEMPORARY TABLE DSN8.DSN8ED2_RS_TBL

PH06TS01 EXEC PGM=IKJEFT01,DYNAMNBR=20
SYSTSPRT DD SYSOUT=
SYSTSIN DD *
DSN SYSTEM(DSN)
RUN PROGRAM(DSNTIAD) PLAN(DSNTIA!!) -
LIB('DSN!0.RUNLIB.LOAD') -
PARM('RC0')
//SYSPRINT DD SYSOUT=
//SYSUDUMP DD SYSOUT=
//SYSIN DD *
SET CURRENT SQLID = 'SYSADM';

DROP PROCEDURE DSN8.DSN8ED2 RESTRICT;
COMMIT;

DROP TABLE DSN8.DSN8ED2_RS_TBL;
COMMIT;

********************************************************************
STEP 2: CREATE SAMPLE STORED PROCEDURE DSN8.DSN8ED2
AND GLOBAL TEMPORARY TABLE DSN8.DSN8ED2_RS_TBL

PH06TS02 EXEC PGM=IKJEFT01,DYNAMNBR=20,
COND=(4,LT)
//SYSTSPRT DD SYSOUT=
//SYSTSIN DD *
DSN SYSTEM(DSN)
RUN PROGRAM(DSNTIAD) PLAN(DSNTIA!!) -
LIB('DSN!0.RUNLIB.LOAD')
//SYSPRINT DD SYSOUT=
//SYSUDUMP DD SYSOUT=
//SYSIN DD *
SET CURRENT SQLID = 'SYSADM';

CREATE PROCEDURE DSN8.DSN8ED2(
  IN VARCHAR(4096)    CCSID EBCDIC,
  OUT INTEGER,
  OUT INTEGER,
  OUT INTEGER,
  OUT VARCHAR(880)     CCSID EBCDIC )
LANGUAGE C
DETERMINISTIC
MODIFIES SQL DATA
EXTERNAL NAME DSN8ED2
PARAMETER STYLE GENERAL WITH NULLS
COLLID DSN8ED!!
WLM ENVIRONMENT WLMENV
ASUTIME LIMIT 50
STAY RESIDENT NO
PROGRAM TYPE MAIN
SECURITY DB2
NO DBINFO
RESULT SET 1
COMMIT ON RETURN NO;

CREATE GLOBAL TEMPORARY TABLE DSN8.DSN8ED2_RS_TBL
( RS_SEQUENCE   INTEGER      NOT NULL,
  RS_DATA       CHAR( 80 )   NOT NULL )
CCSID EBCDIC;
Related reference
“Sample applications in TSO” on page 1027
A set of Db2 sample applications run in the TSO environment.

**DSNTEJ61**
This JCL creates a sample application, DSN8EC1, that demonstrates a Db2 stored procedure for IMS ODBA.
/*
DIS: Retrieve an entry
- UPD: Update an entry
(3) LAST_NAME: Operand for, or result of, COMMAND
(4) FIRST_NAME:
(5) EXTENSION:
(6) ZIP-CODE:
- Output only:
(7) AIBRETRN: Return code from IMS AIB
(8) AIBREASRN: Reason code from IMS AIB
(9) ERROR-CALL: DL/I command executed
* *
Dependencies:
(1) Run this job at the server site before running sample job
DSNTEJ62 at the client site
(2) The server site must have an IMS subsystem running IMS/ESA V6
or a subsequent release
(3) This IMS subsystem must have the following IMS IVP parts
available
(4) Specify the id for this IMS subsystem in DB2 sample job
DSNTEJ62, step PH062S03
(5) The server site must also have a WLM environment started by
a proc that references the IMS reslib in both the STEPLIB DD
and the DFSRESLIB DD. See the DB2 Installation Guide for more
information.
(6) Before running this job, verify that this WLM environment is
the one specified in the CREATE PROCEDURE statement in step
PH061S01.
* *
Change Activity =
08/18/2014 Single-phase migration
s21938_inst1 s21938
* *
******************************************************************************
JOBLIB   DD DSN=DSN!!0.SDSNEXIT,DISP=SHR
// DSN=DSN!!0.SDSNLOAD,DISP=SHR
// DSN=CEE.V!R!M!.SCEERUN,DISP=SHR
// DSN=DSN!!0.RUNLIB.LOAD,DISP=SHR
******************************************************************************
******************************************************************************
//STEP 1: Drop the sample ODBA stored procedure, DSN8.DSN8EC1
******************************************************************************
PH061S01 EXEC PG=IKJEFT01,DYNAMNBR=20
//SYSTSPT DD SYSOUT=
//SYSTSIN DD *
DSN SYSTEM(DSN)
RUN PROGRAM(DSNTIAD) PLAN(DSNTIA!!) -
LIB('DSN!!0.RUNLIB.LOAD') -
PARM('RC0')
//SYSPRINT DD SYSOUT=
//SYSDUMP DD SYSOUT=
//SYSIN DD *
SET CURRENT SQLID = 'SYSADM';
DROP PROCEDURE DSN8.DSN8EC1 RESTRICT;
* *
******************************************************************************
//STEP 2: Create the sample ODBA stored procedure, DSN8.DSN8EC1
******************************************************************************
PH061S02 EXEC PG=IKJEFT01,DYNAMNBR=20
//SYSTSPT DD SYSOUT=
//SYSTSIN DD *
DSN SYSTEM(DSN)
RUN PROGRAM(DSNTIAD) PLAN(DSNTIA!!) -
LIB('DSN!!0.RUNLIB.LOAD') -
//SYSPRINT DD SYSOUT=
//SYSDUMP DD SYSOUT=
//SYSIN DD *
SET CURRENT SQLID = 'SYSADM';
CREATE PROCEDURE
DSN8.DSN8EC1(
IN CHAR(8) CCSID EBCDIC,
INOUT CHAR(8) CCSID EBCDIC,
INOUT CHAR(10) CCSID EBCDIC,
INOUT CHAR(10) CCSID EBCDIC,
INOUT CHAR(10) CCSID EBCDIC,
OUT INT,
OUT INT,
OUT CHAR(4)     CCSID EBCDIC )
FENCED
RESULT SETS 0
EXTERNAL NAME DSNBEC1
LANGUAGE COBOL
PARAMETER STYLE GENERAL
NOT DETERMINISTIC
NO SQL
NO DBINFO
NO COLLID
WLM ENVIRONMENT WLMENV
AS UTIME LIMIT 50
STAY RESIDENT NO
PROGRAM TYPE MAIN
SECURITY DB2
RUN OPTIONS 'TRAP(OFF),RPTOPTS(OFF),TERMTHDAC((QUIET),NONOVR)'
COMMIT ON RETURN NO;

/*
 *********************************************************************/
//*  Step 3: Pre-compile, compile, and link-edit the stored procedure

//**PH061S03 EXEC DSNHICOB, MEM=DSN8EC1,
//**     COND=(4,LT),
//**     PARM.PC=('HOST(IBMCOB)',APOST,APOSTSQL,SOURCE,
//**          NOXREF,'SQL(DB2)','DEC(31)'),
//**     PARM.COB=(NOSEQUENCE,QUOTE,RENT,'PGMNAME(LONGUPPER)')
//**PC.DBRMLIB   DD DSN=DSN!!0.DBRMLIB.DATA(DSN8EC1),
//**                DISP=SHR
//**PC.SYSLIB    DD DSN=DSN!!0.SRCLIB.DATA,
//**                DISP=SHR
//**PC.SYSIN     DD DSN=DSN!!0.SDSNSAMP(DSN8EC1),
//**                DISP=SHR
//**LKED.SYSLMOD DD DSN=DSN!!0.RUNLIB.LOAD(DSN8EC1),
//**                DISP=SHR
//**LKED.SYSIN   DD *
//** INCLUDE SYSLIB(DSNRLI)
NAME DSN8EC1(R)


/*
 *********************************************************************/
//*  Step 4: Bind the stored procedure package
//** Note: This step is commented out for sample stored
//** procedure DSN8EC1 because it contains no SQL
//** statements. If your stored procedure contains
//** SQL statements, you must bind it as a package.

//**PH061S04 EXEC PGM=IKJEFT01, DYNAMNBR=20, COND=(4,LT)
////**DBRMLIB   DD DSN=DSN!!0.DBRMLIB.DATA,
//**                DISP=SHR
//****SYSTSPRT DD SYSOUT=*  
//****SYSPRINT DD SYSOUT=*  
//****SYSUDUMP DD SYSOUT=*  
//****SYSTSNIN DD *  
//**     ** DSN SYSTEM(DSN)
//**     ** BIND PACKAGE(DSN8EC1!) MEMBER(DSNBEC1) APPLCOMPAT(V1!R1) +
//**          ** ACT(REP) ISO(CS) CURRENTDATA(YES) ENCODING(EBCDIC)
/**

Related reference
“Sample applications in TSO” on page 1027
A set of Db2 sample applications run in the TSO environment.

DSNTEJ62
This JCL prepares and executes a sample application program, DSN8EC2, that demonstrates how to call a
Db2 stored procedure for IMS ODBA.

//*******************************************************************************
//** Name = DSNTEJ62
//** Descriptive Name = DB2 Sample Application
//**     Phase 6
//**     Sample Client: Stored procedure for IMS ODBA
//**     Cobol Language
//**
//** LICENSED MATERIALS - PROPERTY OF IBM
//** 5650-DB2
//** (C) COPYRIGHT 1982, 2016 IBM Corp. All Rights Reserved.

Chapter 12. Sample data and applications supplied with Db2 for z/OS 1313
STATUS = Version 12

Function = This JCL prepares and executes a sample application program, DSN8EC2, that demonstrates how to call a DB2 stored procedure for IMS ODBA. The results are directed to the SYSOUT DD.

DSN8EC2 accepts a runtime parameter in step PH062S03 that specifies both the DB2 server location name where the stored procedure is registered and the id of the IMS subsystem where the ODBA activity is to occur. You must modify this job to provide the IMS subsystem id. See step PH062S03 for details.

Dependencies:
(1) Run sample job DSNTEJ61 at the server site before running this job; DSNTEJ61 prepares the sample stored proc for IMS ODBA
(2) Modify this job as directed in step PH062S03
(3) Run this job at the client site

Change activity = 08/18/2014 Single-phase migration s21938_inst1 s21938

*---------------------------------------------------------------*
*/

JOBLIB DD DSN=DSN!!0.SDSNEXIT,DISP=SHR
DD DSN=DSN!!0.SDSNLOAD,DISP=SHR
DD DSN=CEE.V!R!!M!.SCEERUN,DISP=SHR
DD DSN=DSN!!0.RUNLIB.LOAD,DISP=SHR
*/

Step 1: Pre-compile, compile, and link-edit the client program

PH062S01 EXEC DSNHICOB,MEM=DSNBEC2,
COND=(4,LT),
PARM.PC=('HOST(IBMCOB)',APOST,APOSTSQL,SOURCE,
NOXREF, 'SQL(DB2)', 'DEC(31)'),
PARM.COB=(NOSEQUENCE,QUOTE,RENT,'PGMNAME(LONGUPPER)'),
PARM.LKED='AMODE=31,RMODE=ANY,MAP'
PC.DBRMLIB DD DSN=DSN!!0.DBRMLIB.DATA(DSN8EC2),
DISP=SHR
PC.SYSLIB DD DSN=DSN!!0.SRCLIB.DATA,
DISP=SHR
PC.SYSIN DD DSN=DSN!!0.SDSNSAMP(DSN8EC2),
DISP=SHR
LKED.SYSLMOD DD DSN=DSN!!0.RUNLIB.LOAD(DSN8EC2),
DISP=SHR
LKED.SYSIN DD *
INCLUDE SYSLIB(DSNELI)
INCLUDE SYSLIB(DSNITAI)
*/

Step 2: Bind the client program package and plan

PH062S02 EXEC PGM=IKJEFT01,DYNAMNBR=20,COND=(4,LT)
DBRMLIB DD DSN=DSN!!0.DBRMLIB.DATA,
DISP=SHR
SYSTSPRT DD SYSOUT=*
SYSPRINT DD SYSOUT=*
SYSUDUMP DD SYSOUT=*
SYSUT1 DD *
DSN SYSTEM(DSN)
BIND PACKAGE(DSN8EC2!!) MEMBER(DSN8EC2) APPLCOMPAT(V!!R1) +
ACT(REQ) ISO(CS) CURRENTDATA(YES) ENCODING(EBCDIC)
BIND PACKAGE(SAMPLOC.DSN8EC2!!) APPLCOMPAT(V!!R1) +
MEMBER(DSN8EC2) ACT(REQ) ISO(CS) CURRENTDATA(YES) ENCODING(EBCDIC)
BIND PLAN(DSN8EC2) -
PKLIST(DSN8EC2!!).DSNBEC2, -
SAMPLOC.DSN8EC2!!).DSNBEC2) -
ACTION(REPLACE) RETAIN +
ISO(CS) CURRENTDATA(YES) ENCODING(EBCDIC)
RUN PROGRAM(DSN8EII) PLAN(DSN8EII) -
LIB('DSN!!0.RUNLIB.LOAD')
*/

Step 3: Invoke the client for the IMS ODBA stored procedure

DSN SYSTEM(DSN)
BIND PACKAGE(DSN8EC2) MEMBER(DSN8EC2) APPLCOMPAT(V!!R1) +
ACT(REP) ISO(CS) CURRENTDATA(YES) ENCODING(EBCDIC)
BIND PACKAGE(SAMPLOC.DSN8EC2!!) APPLCOMPAT(V!!R1) +
MEMBER(DSN8EC2) ACT(REP) ISO(CS) CURRENTDATA(YES) ENCODING(EBCDIC)
BIND PLAN(DSN8EC2) -
PKLIST(DSN8EC2!!).DSNBEC2, -
SAMPLOC.DSN8EC2!!).DSNBEC2) -
ACTION(REPLACE) RETAIN +
ISO(CS) CURRENTDATA(YES) ENCODING(EBCDIC)
RUN PROGRAM(DSN8EII) PLAN(DSN8EII) -
LIB('DSN!!0.RUNLIB.LOAD')
*/
single argument that specifies the DB2 server location name -and- the IMS subsystem id, in that order and separated by a single blank character.

Example: PARMS('SANTA_TERESA_LAB IMSP')

Verify that the PARMS keyword below specifies the name of the DB2 server location name where you ran DSNTEJ61.

Change the string ?IMSID? to the id of the IMS subsystem where you want the ODBA-directed activity to occur. This subsystem must reside on the same server as the DB2 server and must be running IMS/ESA V6 or a subsequent release.

Ph062s03 exec pgm=ikjeft01,dynamnbr=20,cond=(4,lt)

Related reference
“Sample applications in TSO” on page 1027
A set of Db2 sample applications run in the TSO environment.

DSNTEJ63
This JCL prepares DSN8ES1, a sample SQL procedure that accepts a department number and returns salary and bonus data for employees in that department from the Db2 sample data base.

Ph063s01 step
Ph063s02 step
Ph063s03 step

Dependencies =
(1) This job requires the DB2-provided JCL procedure DSNHSQL
(2) Run this job prior to running the client job DSNTEJ64

Notes =
08/18/2014 Single-phase migration
s21938_inst1 s21938
/**
 * JOBLIB DD DSN=DSN!!0.SDSNEXIT,DISP=SHR
 * DD DSN=DSN!!0.SDSNLOAD,DISP=SHR
 * DD DSN=CEE.VIR!M!.SCEERUN,DISP=SHR
 * DD DSN=DSN!!0.RUNLIB.LOAD,DISP=SHR
 */

/**
 * **********************************************************
 * ** STEP 1: Drop any pre-existing entries for stored proc DSN8ES1
 * ** and the global temporary table for its result set
 * **********************************************************
 * PH063S01 EXEC PGM=IKJEFT01,DYNAMNBR=20
 * /SYSIN DD *
 * DSN SYSTEM(DSN)
 * RUN PROGRAM(DSNTIAD) PLAN(DSN8ES1) -
 * LIB('DSN!!0.RUNLIB.LOAD') -
 * PARM('RC0')
 */

Set CURRENT SQLID = 'SYSADM';
DROP PROCEDURE DSN8.DSN8ES1 RESTRICT;
COMMIT;
DROP TABLE DSN8.DSN8ES1_RS_TBL;
COMMIT;

/**
 * **********************************************************
 * ** Step 2: Pre-compile, compile, and link-edit the stored procedure
 * **********************************************************
 * PH063S02 EXEC DSNHSQL,MEM=DSN8ES1,
 * COND=(4,LT),
 * PARM.PC=('HOST(SQL),SOURCE,XREF,MAR(1,72),CCSID(37),
 * 'STDSQL(NO)'),
 * PARM.PCC=('HOST(C),SOURCE,XREF,MAR(1,80),CCSID(37),
 * 'TWOPASS,STDSQL(NO)'),
 * PARM.C='SOURCE LIST MARGINS(1,80) NOSEQ LO RENT
 * LOCALE("SAA") OPTFILE(DD:CCOPTS)',
 * PARM.LKED='AMODE=31, RMODE=ANY, MAP, RENT'
 * /PC.SYSLIB DD DSN=DSN!!0.SRCLIB.DATA,
 * DISP=SHR
 * /PC.SYSIN DD DSN=DSN!!0.NEW.SDSNSAMP(DSN8ES1),
 * DISP=SHR
 * /PC.SYSLIB DD DSN=DSN!!0.SRCLIB.DATA,
 * DISP=SHR
 * /PC.DBRMLIB DD DSN=DSN!!0.DBRMLIB.DATA(DSN8ES1),
 * DISP=SHR
 * /PCC.SYSLIB DD DSN=DSN!!0.SRCLIB.DATA,
 * DISP=SHR
 * /LKED.SYSLMOD DD DSN=DSN!!0.RUNLIB.LOAD(DSN8ES1),
 * DISP=SHR
 * /LKED.SYSIN DD *
 * INCLUDE SYSLIB(DSNRLI)
 * NAME DSN8ES1(R)
 */

/**
 * **********************************************************
 * ** Step 3: Create the global temp table for DSN8ES1's result set
 * ** Register DSN8ES1 in SYSIBM.SYSROUTINES
 * ** Bind the package for DSN8ES1
 * **********************************************************
 * PH063S03 EXEC PGM=IKJEFT01,DYNAMNBR=20,
 * COND=(4,LT)
 * /DBRMLIB DD DSN=DSN!!0.DBRMLIB.DATA(DSN8ES1),
 * DISP=SHR
 * /SYSIN DD *
 * DSNTIAD DD *
 * EXEC PGM=IKJEFT01,DYNAMNBR=20
 * COND=(4,LT)
 * /DBRMLIB DD DSN=DSN!!0.DBRMLIB.DATA(DSN8ES1),
 * DISP=SHR
 * /SYSIN DD *
 */
Related reference
“Sample applications in TSO” on page 1027
A set of Db2 sample applications run in the TSO environment.

**DSNTEJ64**
This JCL prepares and executes DSN8ED3, a sample caller for the sample SQL procedure DSN8ES1.

```c
//******************************************************************************
//  Name = DSNTEJ64
//  Description = DB2 Sample Application
//  Phase 6
//  Sample Caller for sample SQL Procedure DSN8ES1
//  C Language
//  LICENSED MATERIALS - PROPERTY OF IBM
//  5650-DB2
//  (C) COPYRIGHT 1982, 2016 IBM CORP. ALL RIGHTS RESERVED.
//  Status = VERSION 12
//  Function = This JCL prepares and executes DSN8ED3, a sample caller for the sample SQL procedure DSN8ES1. DSN8ED3 passes a sample department number to DSN8ES1, then processes what is returned:
//  - Parameters containing:
//    - The total earnings (salaries and bonuses) of employees in that department
//    - The number of employees who got a bonus
//    - A result set containing a row of data (serial no, first and last name, salary, and bonus) for each employee who got a bonus
//  Pseudocode =
//  PH064S01 Step Prepare DSN8ED3 load module from DSN8ED3 src
//  PH064S02 Step Bind DSN8ED3 package in collection DSN8ED!!
//  PH064S03 Step Run DSN8ED3 to call stored procedure DSN8ES1
//  Dependencies =
//  (1) Run sample job DSNTEJ63 prior to running this job
//  (2) This job requires the DB2-provided JCL procedure DSNHC
//  Notes =
//  10/16/2013 Don’t use prelinker by default PI13612 DM1812
//  08/18/2014 Single-phase migration s21938_inst1 s21938
//******************************************************************************
// JOBLIB
//  DD DSN=DSN8!!:DSN8EXIT,DISP=SHR
//  DD DSN=DSN8!!:DSN8LOAD,DISP=SHR
//  DD DSN=CEE.V1M.SCEERUN,DISP=SHR
//  DD DSN=DSN8!!:RUNLIB.LOAD,DISP=SHR
//******************************************************************************
// Step 1: Pre-compile, compile, and link-edit DSN8ED3
// PH064S01 EXEC DSNHC,MEM=DSN8ED3,
// PARM.PC=('HOST(C),CCSID(1047),MARGINS(1,72),STDSQL(NO),')
```

Chapter 12. Sample data and applications supplied with Db2 for z/OS 1317
Related reference

“Sample applications in TSO” on page 1027
A set of Db2 sample applications run in the TSO environment.

DSNTEJ65

This JCL does the following.

 exec pgm=ikjeft01,dynamnbr=20,cond=(4,lt)
 bind plan(dsn8ed3) - pklist(dsn8ed3, - samploc.dsn8ed3, - samploc.dsn8ed3, - samploc.dsn8ed3) - action(replace) retain + iso(cs) currentdata(yes) encoding(ebcdic)
 run program(dsntiadi) plan(dsntia!!) - lib('dsn!!0.runlib.load')
FUNCTION =
This JCL does the following:
(1) Prepares and binds DSN8ED4, a sample caller for DSNTPSMP, the DB2 Stored Procedures Processor.
(2) Invokes DSN8ED4 to prequalify that the server has DSNTPSMP at the proper interface level supported by the this client
(3) Invokes DSN8ED4 to pass a sample SQL Procedure, DSN8.DSN8ES2, to DSNTPSMP for preparation
(4) Prepares, binds, and executes DSN8ED5, a sample caller for DSN8.DSN8ES2

PSEUOCODE =
PH065S01 Step Prepare DSN8ED4 (sample caller of DSNTPSMP)
PH065S02 Step Bind DSN8ED4
PH065S03 Step Call DSN8ED4 to request DSNTPSMP QUERYLEVEL
PH065S04 Step Call DSN8ED4 to pass DSN8.DSN8ES2 to DSNTPSMP
PH065S05 Step Prepare DSN8ED5 (sample caller of DSN8.DSN8ES2)
PH065S06 Step Bind DSN8ED5
PH065S07 Step Call DSN8ED5 to call DSN8.DSN8ES2

DEPENDENCIES =
(1) Sample program requires DSNTPSMP (the DB2 SQL Procedures Processor)

NOTES =

CHANGE ACTIVITY =
10/16/2013 Don't use prelinker by default PI13612 DM1812
08/18/2014 Single-phase migration s21938_inst1 s21938

******************************************************************************
//JOBLIB   DD  DISP=SHR,DSN=DSN!!0.SDSNEXIT
//         DD  DISP=SHR,DSN=DSN!!0.SDSNLOAD
//         DD  DISP=SHR,DSN=CEE.V!R!M!.SCEERUN
//         DD  DISP=SHR,DSN=DSN!!0.RUNLIB.LOAD
******************************************************************************
// Step 1: Prepare DSN8ED4, caller for DSNTPSMP
******************************************************************************
//PH065S01 EXEC DSNHC,MEM=DSN8ED4,
//         PARM.PC=('HOST(C),CCSID(1047),MARGINS(1,72),STDSQL(NO)',
//             SOURCE,XREF),
//         PARM.C='SOURCE LIST MAR(1,72) LO RENT OPTFILE(DD:CCOPTS)',
//         PARM.LKED='AMODE=31,RMODE=ANY,MAP,NORENT,UPCASE'
//PC.DBRMLIB   DD DSN=DSN!!0.DBRMLIB.DATA(DSN8ED4),
//             DISP=SHR
//PC.SYSLIB    DD DSN=DSN!!0.SRCLIB.DATA,
//             DISP=SHR
//PC.SYSIN     DD DSN=DSN!!0.SDSNSAMP(DSN8ED4),
//             DISP=SHR
//LKED.SYSLMOD DD DSN=DSN!!0.RUNLIB.LOAD(DSN8ED4),
//             DISP=SHR
//LKED.SYSIN   DD *
//INCLUDE SYSLIB(DSNELI)
//INCLUDE SYSLIB(DSNIAR)
******************************************************************************
// Step 2: Bind DSN8ED4's PLAN
******************************************************************************
//PH065S02 EXEC PGM=IKJEFT01,DYNAMNBR=20,COND=(4,LT)
//DBRMLIB DD DISP=SHR,DSN=DSN!!0.DBRMLIB.DATA
//SYSTSRT DD SYSOUT=*
//SYSPRINT DD SYSOUT=*
//SYSDUMP DD SYSOUT=*
//SYSSC DD +
//SET CURRENT SGD = 'SYSADM';
//GRANT BIND, EXECUTE ON PLAN DSN8ED4 TO PUBLIC;
//SYSSC DD +
//DSN SYSTEM(DSN)
//BIND PACKAGE(DSN8ED4)) MEMBER(DSN8ED4) APPLCOMPAT(V(!R1) +
//ACT(REP) ISO(CS) CURRENTDATA(YES) ENCODING(EBCDIC)
//BIND PACKAGE(SAMPLOC.DSN8ED4) MEMBER(DSN8ED4) -
//APPLCOMPAT(V(!R1) +
//ACT(REP) ISO(CS) CURRENTDATA(YES) ENCODING(EBCDIC)
//BIND PLAN(DSN8ED4) -
//PKLIST(DSN8ED4) -
//SAMPLOC.DSN8ED4 -
//SAMPLOC.DSN8ED4 -
//SAMPLOC.DSN8ED4 -
//SAMPLOC.DSN8ED4 -
ACTION(REPLACE) RETAIN +
ISO(CS) CURRENTDATA(YES) ENCODING(EBCDIC)
RUN PROGRAM(DSNTIAD) PLAN(DSNTIA!!) -
LIB('DSN!!0.RUNLIB.LOAD')
END

//****************************************************************************
//*  STEP 3: Invoke DSN8ED4 to pass a QUERYLEVEL request to DSNTPSMP.
//*          This is a prequalification that the server has a DSNTPSMP
//*          at the correct Interface level for our DSN8ED4 client.
//*         Parms: (* used as place holders)
//*          (1) QUERYLEVEL
//*          (2) * 
//*          (3) * 
//*          (4) (optional) name of server where DSNTPSMP is to be run
//*          Note: DSN8ED4 requires all the same definitions be present
//*          as on a BUILD request, even though only the function
//*          request QUERYLEVEL is passed to DSNTPSMP.
//*****************************************************************************

PH065S03 EXEC PGM=IKJEFT01,DYNAMNBR=20,COND=(4,LT)
//SYSTSIN DD DSN SYSTEM(DSN)
//SYSTSPRT DD SYSOUT=*
//SYSPRINT DD SYSOUT=*
//SYSUDUMP DD SYSOUT=*
//SYSTSIN DD *
DSN SYSTEM(DSN)
RUN PROGRAM(DSN8ED4) PLAN(DSN8ED4) -
PARMS('QUERYLEVEL ** SAMPLOC')
END

//PCOPTS DD *
//COPTS DD *
//PLKDOPTS DD *
//BINDOPTS DD *
//SOLIN DD *
//REPORT01 DD SYSOUT=*,DCB=(RECFM=FBA)
//REPORT02 DD SYSOUT=*
//REPORT03 DD SYSOUT=*
//REPORT04 DD SYSOUT=*
//REPORT05 DD SYSOUT=*
//REPORT06 DD SYSOUT=*

//****************************************************************************
//*  STEP 4:  Invoke DSN8ED4 to pass sample SQL Procedure DSN8.DSN8ES2
//*           to DSNTPSMP
//*          Parms:
//*           (1) operation to be performed by DSNTPSMP
//*           (2) schema.name of SQL proc to be prepared by DSNTPSMP
//*           (3) SQL authid to be used when calling DSNTPSMP
//*           (4) (optional) name of server where DSNTPSMP is to be run
//*           Note: Options passed in the PCOPTS, COPTS, PLKDOPTS, and
//*                 BINDOPTS DDs can span more than one input record.
//*                 Do not use continuation characters (+ or -) to
//*                 continue BIND options onto the next BINDOPTS record
//****************************************************************************

PH065S04 EXEC PGM=IKJEFT01,DYNAMNBR=20,COND=(4,LT)
//SYSTSIN DD DSN SYSTEM(DSN)
//SYSTSPRT DD SYSOUT=*
//SYSPRINT DD SYSOUT=*
//SYSUDUMP DD SYSOUT=*
//SYSTSIN DD *
DSN SYSTEM(DSN)
RUN PROGRAM(DSN8ED4) PLAN(DSN8ED4) -
PARMS('REBUILD DSN8.DSN8ES2 AUTHID SAMPLOC')
END

//PCOPTS DD *
//COPTS DD *
//PLKDOPTS DD *
//BINDOPTS DD *
//SOLIN DD *
//SOURCE,XREF,MAP(1,80),STDSQL(NO)
//SOURCE LIST MAR(1,80) NOSEQ LO RENT
//AMODE=31,RMODE=ANY,MAP,RENT
//QUALIFIER(DSN8!!0) ACT(REP) ISO(CS) CURRENTDATA(YES) ENCODING(EBCDIC)
//SQLIN DD DSN=DSN!!0.NEW.SDSNSAMP(DSN8ES2),
//       DISP=SHR
//REPORT01 DD SYSOUT=*,DCB=(RECFM=FBA)
//REPORT02 DD SYSOUT=*
//REPORT03 DD SYSOUT=*
//REPORT04 DD SYSOUT=*
//REPORT05 DD SYSOUT=*

1320 Db2 12 for z/OS: Application Programming and SQL Guide
//REPORT06 DD SYSOUT=* 
//
*************************************************************************************
// Step 5: Prepare DSN8ED5, sample caller of DSN8.DSN8ES2
*************************************************************************************
//PH065S05 EXEC DSNHC,MEM=DSN8ED5,COND=(4,LT),
//      PARM.PC=('HOST(C),CCSID(1047),MARGINS(1,72),STDSQL(NO),' 
//              SOURCE,XREF),
//      PARM.C='SOURCE LIST MAR(1,72) LO RENT OPTFILE(DD:CCOPTS)'
//      PARM.LKED='AMODE=31,RMODE=ANY,MAP,NORENT,UPCASE'
//PC.DBRMLIB DD DSN=DSN!!0.DBRMLIB.DATA(DSN8ED5),
//      DISP=SHR
//PC.SYSLIB DD DSN=DSN!!0.SRCLIB.DATA,
//      DISP=SHR
//PC.SYSIN DD DSN=DSN!!0.SDSNSAMP(DSN8ED5),
//      DISP=SHR
//LKED.SYSLMOD DD DSN=DSN!!0.RUNLIB.LOAD(DSN8ED5),
//      DISP=SHR
//LKED.SYSIN DD *
//INCLUDE SYSLIB(DSNELI)
//INCLUDE SYSLIB(DSNTIAR)
//
******************************************************************************
// Step 6: Bind DSN8ED5's PLAN
******************************************************************************
//PH065S06 EXEC PGM=IKJEFT01,DYNAMNBR=20,COND=(4,LT)
//DBRMLIB DD DISP=SHR,DSN=DSN!!0.DBRMLIB.DATA
//SYSTSPRT DD SYSOUT=* 
//SYSPRINT DD SYSOUT=* 
//SYSUDUMP DD SYSOUT=* 
//SYSTSIN DD *
//DSN SYSTEM(DSN)
//SET CURRENT SQLID = 'SYSADM';
//GRANT BIND, EXECUTE ON PLAN DSN8ED5
//TO PUBLIC;
//SYSTSIN DD *
//DSN SYSTEM(DSN)
//BIND PACKAGE(DSN8ED!!) MEMBER(DSN8ED5) APPLCOMPAT(V!!R1) +
//ACT(REP) ISO(CS) CURRENTDATA(YES) ENCODING(EBCDIC)
//BIND PACKAGE(SAMPLOC.DSN8ED!!) MEMBER(DSN8ED5) -
//APPLCOMPAT(V!!R1) +
//ACT(REP) ISO(CS) CURRENTDATA(YES) ENCODING(EBCDIC)
//BIND PLAN(DSN8ED5) -
//PKLIST(DSN8ED!!,DSN8ED5, -
//SAMPLOC.DSN8ED!!..DSN8ED5, -
//SAMPLOC.DSN8ED!!..*) -
//ACTION(REPLACE) RETAIN +
//ISO(CS) CURRENTDATA(YES) ENCODING(EBCDIC)
//RUN PROGRAM(DSNTIAD) PLAN(DSNTIA!!) -
//LIB('DSN!!0.RUNLIB.LOAD')
//END
//
******************************************************************************
// Step 7: Invoke DSN8ED5 to call sample SQL Procedure DSN8.DSN8ES2
******************************************************************************
//PH065S07 EXEC PGM=IKJEFT01,DYNAMNBR=20,COND=(4,LT)
//SYSTSPRT DD SYSOUT=* 
//SYSPRINT DD SYSOUT=* 
//SYSDUMP DD SYSOUT=* 
//SYSTSIN DD *
//DSN SYSTEM(DSN)
//RUN PROGRAM(DSN8ED5) PLAN(DSN8ED5) -
//PARMS('1500.00 SAMPLOC')
//END

Related reference
“Sample applications in TSO” on page 1027
A set of Db2 sample applications run in the TSO environment.

DSNTEJ6W
This JCL does the following.

*** DB2 Sample Application
*** Phase 6
*** Sample caller for Stored Procedure WLM_REFRESH
*** IBM C/C++ for z/OS
***
**Function**

* Prepares and executes DSN8ED6, a sample caller of the WLM_REFRESH stored procedure. WLM_REFRESH refreshes a WLM environment specified as an input parameter using an authorization ID also specified as an input parameter. The authorization ID must have READ access on a resource profile called !DSN!.WLM_REFRESH.!WLMENV!
* Use job DSNTIJRA job step DSNTWR to create and permit access to this resource profile.
* Optional: Prepares DSNTWR and DSNTWRE, external modules for WLM_REFRESH. These modules are provided in SDSNLOAD so preparing them is required only if you maintain a customized copy of DSNTWRS, the sample source code for DSNTWR, or DSNTWRE, the sample source code for DSNTWRE.

**Pseudocode**

**PH06WS00 Step** Optional: Prepare DSNTWRE, a program for getting the DB2 Environment Info Block (EIB)
  -> Uncomment and run this step if you want to override the DB2-supplied DSNTWRE module
**PH06WS01 Step** Optional: Prepare DSNTWR, the external module for SYSPROC.WLM_REFRESH
  -> Uncomment and run this step if you want to override the DB2-supplied DSNTWR module
**PH06WS02 Step** Optional: Bind the package for DSNTWR
  -> Uncomment and run this step only if you also uncomment and run the step PH06WS01
**PH06WS03 Step** Prepare DSN8ED6
**PH06WS04 Step** Bind the plan and package for DSN8ED6
**PH06WS05 Step** Invoke DSN8ED6

**Dependencies**

* (1) This job requires the DB2-provided JCL procedures DSNHASM and DSNHC
* (2) Run this job only after running jobs DSNTIJTM and DSNTIJRT
* (3) The DSN8ED6 program receives parameters that contain the name of the WLM environment to be refreshed and the authorization ID to be used for the request. The authorization ID must have READ access on a resource profile called !DSN!.WLM_REFRESH.!WLMENV!
* Use job DSNTIJRA job step DSNTWR to create and permit access to this resource profile.

**Notes**

* Change Activity =
  * 10/16/2013 Don't use prelinker by default PI13612 DM1812
  * 08/18/2014 Single-phase migration s21938_instl s21938

---

```
// JOBLIB   DD  DISP=SHR,DSN=DSN!!0.SDSNEXIT
// DD  DISP=SHR,DSN=DSN!!0.SDSNLOAD
// DD  DISP=SHR,DSN=C.EE.V!R!M!.SCEERUN
//
// /* Step 0 (Optional): Prepare DSNTWRE, a program that gets the DB2 group attach name
// /*
// /*
// /*PH06WS00 EXEC DSNHASM,COND=(4,LT),
// * MEM=DSNTWRE,
// /*
// * PARM.PC='HOST(ASM),STDSQL(NO)',
// * PARM.ASM='RENT,OBJECT,NODECK',
// * PARM.LKED='LIST,XREF,AMODE=31,RMODE=ANY,RENT'
// /*
// /*PC.DBRMLIB DD DSN=DSN!!0.DBRMLIB.DATA(DSNTWRE),
// /* DISP=SHR
// /*PC.SYSLIB DD DSN=DSN!!0.SDSNMACS,
// /* DISP=SHR
// /*
// /*
```
Chapter 12. Sample data and applications supplied with Db2 for z/OS 1323
Related reference
“Sample applications in TSO” on page 1027
A set of Db2 sample applications run in the TSO environment.

**DSNTEJ6Z**

This JCL prepares and executes DSN8ED7, a sample caller of ADMIN_INFO_SYSPARM, a Db2-provided stored procedure that returns the current settings of your Db2 subsystem parameters.

```sql
// SYSIN DD *
SET CURRENT SQLID = 'SYSADM';
GRANT BIND, EXECUTE ON PLAN DSN8ED6
TO PUBLIC;
//*
// SYSTSIN DD *
DSN SYSTEM(DSN)
RUN PROGRAM(DSN8ED6) PLAN(DSN8ED6) -
LIB('DSN!!0.RUNLIB.LOAD') -
PARMS('!WLMENV! !DSN! !ID!')
END
```
Pseudocode =

PH06ZS01 Step Prepare DSN8ED7
PH06ZS02 Step Bind the plan and package for DSN8ED7
PH06ZS03 Step Invoke DSN8ED7

Dependencies =
- This job requires the DB2-provided JCL procedure DSNHC

Notes =

Change Activity =
10/16/2013 Don't use prelinker by default PI13612 DM1812
08/18/2014 Single-phase migration s21938_inst1 s21938

Dependencies =
- This job requires the DB2-provided JCL procedure DSNHC

Notes =

Change Activity =
10/16/2013 Don't use prelinker by default PI13612 DM1812
08/18/2014 Single-phase migration s21938_inst1 s21938

Pseudocode =

PH06ZS01 Step Prepare DSN8ED7
PH06ZS02 Step Bind the plan and package for DSN8ED7
PH06ZS03 Step Invoke DSN8ED7

Dependencies =
- This job requires the DB2-provided JCL procedure DSNHC

Notes =

Change Activity =
10/16/2013 Don't use prelinker by default PI13612 DM1812
08/18/2014 Single-phase migration s21938_inst1 s21938

Related reference
“Sample applications in TSO” on page 1027
A set of Db2 sample applications run in the TSO environment.

**DSNTEJ66**

This JCL does the following.

```plaintext
//*****************************************************************************
//* Name = DSNTEJ66
//*//* Descriptive Name = DB2 Sample Application - Native SQL Procedure
//*                     Phase 6
//*//* Licensed Materials - Property of IBM
//* 5650-DB2
//* (C) COPYRIGHT 2006, 2016 IBM Corp. All Rights Reserved.
//*//* STATUS = Version 12
//*//* Function = This JCL does the following:
//*             - Creates a sample native SQL procedure called
//*               DSN8.DSN8ES3 that generates and returns (by result
//*               set) a CREATE PROCEDURE statement for a given stored
//*               procedure.
//*             - Prepares and executes a sample caller of DSN8ES3
//*               called DSN8ED9.
//*             - Shows how to use ALTER PROCEDURE ... ADD VERSION to
//*               create a version V2 of DSN8ES3 that does the same
//*               thing as the original version but also adds a
//*               terminating semicolon at the end of the generated
//*               CREATE PROCEDURE statement
//*             - Shows how to ALTER ACTIVATE version V2 to make it
//*               the active version of DSN8ES3
//*             - Shows how to DEPLOY DSN8ES3 at a remote site
//*//* Restrictions =
//* As part of the setup to DEPLOY DSN8ES3, the DSNTEP2 application
//* needs to be able to connect to the remote site.
//*//* Notice =
//* Pseudocode =
//* PH066S01 Step  Drop objects created by prior runs of this job
//* PH066S02 Step  Create the global temporary table for
//*                  DSN8.DSN8ES3
//* PH066S03 Step  Prepare DSN8ES3 as a native SQL procedure
//*                  -> Also generates a package called
//*                  DSN8.DSN8ES3
//* PH066S04 Step  Prepare DSN8ED9, sample caller for the DSN8ES3
//*                  SQL proc
//* PH066S05 Step  Bind the plan for DSN8ED9
//* PH066S06 Step  Execute DSN8ED9 to request a CREATE PROC
//*                  statement for the stored procedure
//*                  SYSPROC.DSNUTILS
//* PH066S07 Step  Create a work copy of the DSN8ES3 source code
//* PH066S08 Step  Use TSO edit to modify the work copy into an
//*                  ALTER PROCEDURE that will make a trivial
//*                  change to DSN8ES3 as VERSION V2
//*                  -> The generated CREATE PROC statement will be
//*                  terminated by a semicolon
//* PH066S09 Step  Save the work copy as DSN8ES3 in
//*                  DSN!!0.NEW.SDSNSAMP
//* PH066S10 Step  Process the ALTER PROCEDURE DSN8ES3 to ADD
//*                  VERSION V2
//*                  -> Also generates a package called
//*                  DSN8.DSN8ES3 (VERSION V2)
//* PH066S11 Step  Activate V2 as the current version of DSN8ES3
//* PH066S12 Step  Execute DSN8ED9 to request a CREATE PROC
//*                  statement for SYSPROC.DSNUTILS
//*                  -> When using DSN8ES3 V2, it's terminated by a
//*                  semicolon
//* PH066S13 Step  Setup to DEPLOY DSN8ES3: Create a global
//*                  temporary table on the remote server
//*                  -> To rerun this step, uncomment the DROP
//*                  and COMMIT statements
//* PH066S14 Step  DEPLOY DSN8ES3 on the remote server
//* PH066S15 Step  Bind the plan for DSN8ED9 on the remote server
//* PH066S16 Step  Execute DSN8ED9 to request a CREATE PROC
//*                  statement for SYSPROC.DSNUTILS at the remote
```
Chapter 12. Sample data and applications supplied with Db2 for z/OS 1327
// PC.SYSLIB DD DSN=DSN!!0.SRCLIB.DATA,
//   DISP=SHR
// PC.SYSIN DD DSN=DSN!!0.DSNSAMP(DSN8ED9),
//   DISP=SHR
// LKED.SYSLMOD DD DSN=DSN!!0.RUNLIB.LOAD(DSN8ED9),
//   DISP=SHR
// LKED.SYSIN DD *
// INCLUDE SYSLIB(DSNELI)
// INCLUDE SYSLIB(DSNTIAR)
// ** Step 5: Bind the plan for DSN8ED9
// **
// PH066505 EXEC PGM=IKJEFT01, DYNAMNBR=20, COND=(4,LT)
// DBRMLIB 1 DD DSN=DSN!!0.DBRMLIB.DATA,
//   DISP=SHR
// SYSTSPRT DD SYSOUT=*
// SYSPRINT DD SYSOUT=*
// SYSUDUMP DD SYSOUT=*
// SYSTSIN DD *
// SET CURRENT SQLID = 'SYSADM';
// GRANT BIND, EXECUTE ON PLAN DSN8ED9
// TO PUBLIC;
// SYSTSIN DD *
// DSN SYSTEM(DSN)
// BIND PACKAGE(DSN8ED9) MEMBER(DSN8ED9) APPLCOMPAT(V!!R1) +
// ACT(REP) ISO(CS) CURRENTDATA(YES) ENCODING(EBCDIC)
// BIND PLAN(DSN8ED9) +
// PKLIST(DSN8ED9) +
// ACTION(REPLACE) RETAIN +
// ISO(CS) CURRENTDATA(YES) ENCODING(EBCDIC)
// RUN PROGRAM(DSNTIAD) PLAN(DSNTIA!!) +
// LIB('DSN!!0.RUNLIB.LOAD')
// **
// ** Step 6: Execute DSN8ED9 to request a CREATE PROC statement
// ** for the stored procedure named SYSPROC.DSNUTILS
// **
// PH066506 EXEC PGM=IKJEFT01, DYNAMNBR=20, COND=(4,LT)
// SYSTSPRT DD SYSOUT=*
// SYSPRINT DD SYSOUT=*
// SYSUDUMP DD SYSOUT=*
// SYSTSIN DD *
// DSN SYSTEM(DSN)
// RUN PROGRAM(DSN8ED9) PLAN(DSN8ED9) +
// LIB('DSN!!0.RUNLIB.LOAD') +
// PARM('SYSPROC DSNUTILS')
// END
// **
// ** Step 7: Create a work copy of the DSN8ES3 source code
// **
// PH066507 EXEC PGM=IEBGENER, COND=(4,LT)
// SYSTSPRT DD SYSOUT=*
// SYSPRINT DD SYSOUT=*
// SYSTSIN DD *
// DSN SYSTEM(DSN)
// EDIT 'DSN!!0.DSN8.DSN8ES3.WORKCOPY' +
// DATA OLD NONUM NORECOVER ASIS
// FIND /CREATE PROCEDURE DSN8.DSN8ES3/
// CHANGE * 1 /CREATE PROCEDURE/ALTER PROCEDURE /
// INSERT ADD VERSION V2
// FIND /PARAMETER CCSID EBCDIC/
// CHANGE * 1 /PARAMETER/ADD TERMINATING SEMICOLON /
// INSERT SET LINE = ';' ;
// INSERT SET RETURN_POINT = 'DONE';
// INSERT GOTO INSERTLINE;
LIST 1 9999
END SAVE

//** Step 9: Save in DSN!!0.NEW.SDSNSAMP
//**
//** PH066S09 EXEC PGML=IEBGENER,COND=(4,LT)
//** SYSIN DD DUMMY
//** SYSPRINT DD SYSPUT=
//** SYSUT1 DD DISP=(OLD,DELETE),
//** DSN=DSN!!0.DSN8.DSN8ES3.WORKCOPY
//** SYSUT2 DD DISP=SHR,
//** DSN=DSN!!0.NEW.SDSNSAMP(DSN8ES3)
//**
//** Step 10: Process the ALTER PROCEDURE DSN8ES3 to ADD VERSION V2
//** -> Also generates a package called DSN8.DSN8ES3 (V2)
//**
//** PH066S10 EXEC PGML=IKJEF101,DYNAMNBR=20,COND=(4,LT)
//** SYSSTPRT DD SYSPUT=
//** SYSPRINT DD SYSPUT=
//** SYSUDUMP DD SYSPUT=
//** SYSSTIN DD *
//** DSN SYSTEM(DSN)
//** RUN PROGRAM(DSNTSEP2) PLAN(DSNTSEP!!) +
//** LIB('DSN!!0.RUNLIB.LOAD') PARM('*/SQLTERM(%)')
END
//** SYSIN DD *
//** SET CURRENT SQLID = 'SYSADM'
//** DD DISP=SHR,
//** DSN=DSN!!0.NEW.SDSNSAMP(DSN8ES3)
//** DD *
//**
//** Step 11: Activate V2 as the current version of DSN8ES3
//**
//** PH066S11 EXEC PGML=IKJEF101,DYNAMNBR=20,COND=(4,LT)
//** SYSSTPRT DD SYSPUT=
//** SYSPRINT DD SYSPUT=
//** SYSUDUMP DD SYSPUT=
//** SYSSTIN DD *
//** DSN SYSTEM(DSN)
//** RUN PROGRAM(DSNTIA) PLAN(DSNTIA!!) +
//** LIB('DSN!!0.RUNLIB.LOAD')
END
//** SYSIN DD *
//** SET CURRENT SQLID = 'SYSADM';
//** ALTER PROCEDURE DSN8.DSN8ES3 ACTIVATE VERSION V2;
//**
//** Step 12: Execute DSN8ED9 to request a CREATE PROC statement
//** for SYSASPROC.DSNUTILU
//** -> When using DSN8ES3 V2, it's terminated by a semicolon
//**
//** PH066S12 EXEC PGML=IKJEF101,DYNAMNBR=20,COND=(4,LT)
//** SYSSTPRT DD SYSPUT=
//** SYSPRINT DD SYSPUT=
//** SYSUDUMP DD SYSPUT=
//** SYSSTIN DD *
//** DSN SYSTEM(DSN)
//** REBIND PACKAGE(DSN8ED9!!.DSN8ED9) APPLCOMPAT(V!!R1) +
//** PLANMGMT(OFF)
//** RUN PROGRAM(DSNTSEP9) PLAN(DSNTSEP9) +
//** LIB('DSN!!0.RUNLIB.LOAD') +
//** PARM('*/SYSPROC DSNUTILU')
END
//**
//** Step 13: Setup to DEPLOY DSN8ES3 - Create a global temporary
//** table on the remote server
//**
//** PH066S13 EXEC PGML=IKJEF101,DYNAMNBR=20,COND=(4,LT)
//** SYSSTPRT DD SYSPUT=
//** SYSPRINT DD SYSPUT=
//** SYSUDUMP DD SYSPUT=
//** SYSSTIN DD *
//** DSN SYSTEM(DSN)
//** RUN PROGRAM(DSNTSEP2) +
//** PLAN(DSNTSEP!!) +
//** LIB('DSN!!0.RUNLIB.LOAD')
//** SYSIN DD *(2)
//** SET CURRENT SQLID = 'SYSADM';
//** CONNECT TO SAMPLOC;
//** DROP TABLE DSN8.DSN8ES3_RS_TBL;
//** COMMIT;

Chapter 12. Sample data and applications supplied with Db2 for z/OS 1329
CREATE GLOBAL TEMPORARY TABLE DSN8.DSN8ES3_RS_TBL
    ( RS_SEQUENCE INTEGER NOT NULL,
      RS_LINE CHAR(80) NOT NULL )
CCSID EBCDIC;

//* Step 14: DEPLOY DSN8ES3 on the remote server
//*
//PH066S14 EXEC PGM=IKJEFT01,DYNAMNBR=20,COND=(4,LT)
//SYSTSIN DD SYSOUT=* 
//SYSPRINT DD SYSOUT=* 
//SYSDUMP DD SYSOUT=* 
//DSN SYSTEM(DSN)
BIND PACKAGE(SAMPLOC.DSN8) APPLCOMPAT(VRR1) +
DEPLOY(DSN8.DSN8ES3) +
COPYVER(V2) +
ACTION(REP)

//* Step 15: Bind the plan for DSN8ED9 on the remote server
//*
//PH066S15 EXEC PGM=IKJEFT01,DYNAMNBR=20,COND=(4,LT)
//DBRMLIB DD DSN=DSN!!0.DBRMLIB.DATA, 
//DISP=SHR 
//SYSTSIN DD SYSOUT=* 
//SYSPRINT DD SYSOUT=* 
//SYSDUMP DD SYSOUT=* 
//DSN SYSTEM(DSN)
BIND PACKAGE(SAMPLOC.DSN8ED!!) MEMBER(DSN8ED9) +
APPLCOMPAT(VRR1) +
ACT(REP) ISO(CS) CURRENTDATA(YES) ENCODING(EBCDIC)
BIND PLAN(DSN8ED9) +
PKLIST(DSN8ED!!.DSN8ED9, * , SAMPLOC.DSN8ED!!..DSN8ED9, +
SAMPLOC.DSN8ES!!..* ) +
ACTION(REPLACE) RETAIN +
ISO(CS) CURRENTDATA(YES) ENCODING(EBCDIC)
RUN  PROGRAM(DSN8TIAD) PLAN(DSN8IA!!) +
LIB('DSN!!0.RUNLIB.LOAD')

//* Step 16: Execute DSN8ED9 to request a CREATE PROC statement for SYSPROC.DSNUTILS at the remote site
//* 
//PH066S16 EXEC PGM=IKJEFT01,DYNAMNBR=20,COND=(4,LT)
//DSN SYSTEM(DSN)
RUN PROGRAM(DSN8ED9) PLAN(DSN8ED9) +
LIB('DSN!!0.RUNLIB.LOAD') +
PARMS('/SYSPROC DSNUTILS SAMPLOC')
END

Related reference
“Sample applications in TSO” on page 1027
A set of Db2 sample applications run in the TSO environment.

DSNTEJ2U
THIS JCL PREPARES THE FOLLOWING Db2 USER-DEFINED FUNCTIONS (UDF'S) AND A DRIVER PROGRAM TO INVOKE THEM.

ростной заполненность

1330 Db2 12 for z/OS: Application Programming and SQL Guide
// Status = Version 12

// Function = This JCL prepares the following DB2 user-defined functions (UDF's) and a driver program to invoke them.

// Notes = Ensure that line number sequencing is set 'on' if this job is submitted from an ISPF edit session

// This job is run after phase 1.

// Change activity =

10/16/2013 Don't use prelinker by default PI13612 DM1812
08/18/2014 Single-phase migration s21938_inst1 s21938

//*********************************************************************

// JOBLIB DD DSN=DSN!!0.SDSNEXIT,DISP=SHR
// DD DSN=DSN!!0.SDSNLOAD,DISP=SHR
// DD DSN=CEE.V!R!M!.SCEERUN,DISP=SHR
// DD DSN=DSN!!0.RUNLIB.LOAD,DISP=SHR

// Step 1: Drop any existing DB2 sample UDF's

PH02US01 EXEC PGM=IKJEFT01,DYNAMNBR=20
SYSTSPRT DD SYSOUT=* 
SYSTSIN DD *
DSN SYSTEM(DSN) 
RUN PROGRAM(DSNTIAD) - 
PLAN(DSNTIA!!) - 
LIB('DSN!!0.RUNLIB.LOAD') - 
PARM('RC0') 
SYSPRINT DD SYSOUT=* 
SYSUDUMP DD SYSOUT=* 
SYSIN DD *

SET CURRENT SQLID = 'SYSADM';

DROP SPECIFIC FUNCTION DSN8.DSN8DUCDDVV RESTRICT;
DROP SPECIFIC FUNCTION DSN8.DSN8DUCDVVV RESTRICT;
DROP SPECIFIC FUNCTION DSN8.DSN8DUADV RESTRICT;
DROP SPECIFIC FUNCTION DSN8.DSN8DUCTTVV RESTRICT;
DROP SPECIFIC FUNCTION DSN8.DSN8DUCTVVV RESTRICT;
DROP SPECIFIC FUNCTION DSN8.DSN8DUATV RESTRICT;
DROP SPECIFIC FUNCTION DSN8.DSN8DUCYFV RESTRICT;
DROP SPECIFIC FUNCTION DSN8.DSN8DUCYFVV RESTRICT;
DROP SPECIFIC FUNCTION DSN8.DSN8EUNDN RESTRICT;
DROP SPECIFIC FUNCTION DSN8.DSN8EUDNV RESTRICT;
DROP SPECIFIC FUNCTION DSN8.DSN8EUMND RESTRICT;
DROP SPECIFIC FUNCTION DSN8.DSN8EUMNV RESTRICT;
DROP SPECIFIC FUNCTION DSN8.DSN8DUTINV RESTRICT;
DROP SPECIFIC FUNCTION DSN8.DSN8DUTINVV RESTRICT;
DROP SPECIFIC FUNCTION DSN8.DSN8DUTISV RESTRICT;
DROP SPECIFIC FUNCTION DSN8.DSN8DUTISVVV RESTRICT;
DROP SPECIFIC FUNCTION DSN8.DSN8DUTILV RESTRICT;
DROP SPECIFIC FUNCTION DSN8.DSN8DUTILVVV RESTRICT;
DROP SPECIFIC FUNCTION DSN8.DSN8DUWFV RESTRICT;

// Step 2: Define sample UDF's to DB2

PH02US02 EXEC PGM=IKJEFT01,DYNAMNBR=20,COND=(4,LT)
SYSTSPRT DD SYSOUT=* 
SYSTSIN DD *
DSN SYSTEM(DSN) 
RUN PROGRAM(DSNTIAD) - 
PLAN(DSNTIA!!) - 
LIB('DSN!!0.RUNLIB.LOAD') 
PARM('RC0') 
SYSPRINT DD SYSOUT=* 
SYSUDUMP DD SYSOUT=* 
SYSIN DD *

SET CURRENT SQLID = 'SYSADM';
CREATE FUNCTION DSN8.ALTDATE(
    VARCHAR(13) CCSID EBCDIC )
RETURNS
    VARCHAR(17) CCSID EBCDIC
SPECIFIC DSN8.DSN8DUADV
LANGUAGE C
DETERMINISTIC
NO SQL
EXTERNAL NAME DSN8DUAD
PARAMETER STYLE DB2SQL
NULL CALL
NO EXTERNAL ACTION
NO SCRATCHPAD
NO FINAL CALL
ALLOW PARALLEL
NO COLLID
ASUTIME LIMIT 10
STAY RESIDENT NO
PROGRAM TYPE SUB
WLM ENVIRONMENT WLMENV
SECURITY DB2
NO DBINFO;

CREATE FUNCTION DSN8.ALTDATE(
    VARCHAR(17) CCSID EBCDIC,
    VARCHAR(13) CCSID EBCDIC,
    VARCHAR(13) CCSID EBCDIC )
RETURNS
    VARCHAR(17) CCSID EBCDIC
SPECIFIC DSN8.DSN8DUCDVVV
LANGUAGE C
DETERMINISTIC
NO SQL
EXTERNAL NAME DSN8DUCD
PARAMETER STYLE DB2SQL
NULL CALL
NO EXTERNAL ACTION
NO SCRATCHPAD
NO FINAL CALL
ALLOW PARALLEL
NO COLLID
ASUTIME LIMIT 10
STAY RESIDENT NO
PROGRAM TYPE SUB
WLM ENVIRONMENT WLMENV
SECURITY DB2
NO DBINFO;

CREATE FUNCTION DSN8.ALTDATE(
    DATE,
    VARCHAR(13) CCSID EBCDIC,
    VARCHAR(13) CCSID EBCDIC )
RETURNS
    VARCHAR(17) CCSID EBCDIC
SPECIFIC DSN8.DSN8DUCDDVV
SOURCE SPECIFIC DSN8.DSN8DUCDVVV;

CREATE FUNCTION DSN8. ALTTIME(
    VARCHAR(14) CCSID EBCDIC )
RETURNS
    VARCHAR(11) CCSID EBCDIC
SPECIFIC DSN8.DSN8DUATV
LANGUAGE C
DETERMINISTIC
NO SQL
EXTERNAL NAME DSN8DUAT
PARAMETER STYLE DB2SQL
NULL CALL
NO EXTERNAL ACTION
NO SCRATCHPAD
NO FINAL CALL
ALLOW PARALLEL
NO COLLID
ASUTIME LIMIT 10
STAY RESIDENT NO
PROGRAM TYPE SUB
WLM ENVIRONMENT WLMENV
CREATE FUNCTION DSN8.ALTTIME(
    VARCHAR(11) CCSID EBCDIC,
    VARCHAR(14) CCSID EBCDIC,
    VARCHAR(14) CCSID EBCDIC)
RETURNS
    VARCHAR(11) CCSID EBCDIC
SPECIFIC DSN8.DSN8DUCTVVV
LANGUAGE C
DETERMINISTIC
NO SQL
EXTERNAL NAME DSN8DUCT
PARAMETER STYLE DB2SQL
NULL CALL
NO EXTERNAL ACTION
NO SCRATCHPAD
NO FINAL CALL
ALLOW PARALLEL
NO COLLID
ASUTIME LIMIT 10
STAY RESIDENT NO
PROGRAM TYPE SUB
WLM ENVIRONMENT WLMENV
SECURITY DB2
NO DBINFO;

CREATE FUNCTION DSN8.ALTTIME(
    TIME,
    VARCHAR(14) CCSID EBCDIC,
    VARCHAR(14) CCSID EBCDIC)
RETURNS
    VARCHAR(11) CCSID EBCDIC
SPECIFIC DSN8.DSN8DUCTTVV
SOURCE SPECIFIC DSN8.DSN8DUCTVVV;

CREATE FUNCTION DSN8.CURRENCY(
    FLOAT,
    VARCHAR(2)  CCSID EBCDIC )
RETURNS
    VARCHAR(19) CCSID EBCDIC
SPECIFIC DSN8.DSN8DUCYFV
LANGUAGE C
DETERMINISTIC
NO SQL
EXTERNAL NAME DSN8DUCY
PARAMETER STYLE DB2SQL
NULL CALL
NO EXTERNAL ACTION
NO SCRATCHPAD
NO FINAL CALL
ALLOW PARALLEL
NO COLLID
ASUTIME LIMIT 10
STAY RESIDENT NO
PROGRAM TYPE MAIN
WLM ENVIRONMENT WLMENV
SECURITY DB2
NO DBINFO;

CREATE FUNCTION DSN8.CURRENCY(
    FLOAT,
    VARCHAR(2)  CCSID EBCDIC,
    VARCHAR(5)  CCSID EBCDIC)
RETURNS
    VARCHAR(19) CCSID EBCDIC
SPECIFIC DSN8.DSN8DUCYFVV
LANGUAGE C
DETERMINISTIC
NO SQL
EXTERNAL NAME DSN8DUCY
PARAMETER STYLE DB2SQL
NULL CALL
NO EXTERNAL ACTION
NO SCRATCHPAD
NO FINAL CALL
ALLOW PARALLEL
CREATE FUNCTION DSN8.DAYNAME(
    VARCHAR(10) CCSID EBCDIC )
RETURNS
    VARCHAR(9) CCSID EBCDIC
SPECIFIC DSN8.DSN8EUDNV
LANGUAGE C
DETERMINISTIC
NO SQL
EXTERNAL NAME DSN8EUDN
PARAMETER STYLE DB2SQL
NULL CALL
NO EXTERNAL ACTION
NO SCRAMBLE
NO FINAL CALL
ALLOW PARALLEL
NO COLLID
ASUTIME LIMIT 10
STAY RESIDENT NO
PROGRAM TYPE SUB
WLM ENVIRONMENT WLMENV
SECURITY DB2
NO DBINFO;

CREATE FUNCTION DSN8.DAYNAME(
    DATE )
RETURNS
    VARCHAR(9) CCSID EBCDIC
SPECIFIC DSN8.DSN8EUDND
SOURCE SPECIFIC DSN8.DSN8EUDNV;

CREATE FUNCTION DSN8.MONTHNAME(
    VARCHAR(10) CCSID EBCDIC )
RETURNS
    VARCHAR(9) CCSID EBCDIC
SPECIFIC DSN8.DSN8EUMNV
LANGUAGE C
DETERMINISTIC
NO SQL
EXTERNAL NAME DSN8EUMN
PARAMETER STYLE DB2SQL
NULL CALL
NO EXTERNAL ACTION
NO SCRAMBLE
NO FINAL CALL
ALLOW PARALLEL
NO COLLID
ASUTIME LIMIT 10
STAY RESIDENT NO
PROGRAM TYPE SUB
WLM ENVIRONMENT WLMENV
SECURITY DB2
NO DBINFO;

CREATE FUNCTION DSN8.MONTHNAME(
    DATE )
RETURNS
    VARCHAR(9) CCSID EBCDIC
SPECIFIC DSN8.DSN8EUMND
SOURCE SPECIFIC DSN8.DSN8EUMNV;

CREATE FUNCTION DSN8.TABLE_NAME(
    VARCHAR(18) CCSID EBCDIC )
RETURNS
    VARCHAR(18) CCSID EBCDIC
SPECIFIC DSN8.DSN8DUTINV
LANGUAGE C
DETERMINISTIC
READS SQL DATA
EXTERNAL NAME DSN8DUTI
CREATE FUNCTION DSN8.TABLE_NAME(
    VARCHAR(18) CCSID EBCDIC,
    VARCHAR(8)  CCSID EBCDIC )
RETURNS
    VARCHAR(18) CCSID EBCDIC
SPECIFIC DSN8.DSN8DUTINVV
LANGUAGE C
DETERMINISTIC
READS SQL DATA
EXTERNAL NAME DSN8DUTI
PARAMETER STYLE DB2SQL
NULL CALL
NO EXTERNAL ACTION
NO SCRATCHPAD
NO FINAL CALL
ALLOW PARALLEL
COLLID DSN8DU!!
ASUTIME LIMIT 10
STAY RESIDENT NO
PROGRAM TYPE MAIN
WLM ENVIRONMENT WLMENV
SECURITY DB2
NO DBINFO;

CREATE FUNCTION DSN8.TABLE_NAME(
    VARCHAR(18) CCSID EBCDIC,
    VARCHAR(8)  CCSID EBCDIC,
    VARCHAR(16) CCSID EBCDIC )
RETURNS
    VARCHAR(18) CCSID EBCDIC
SPECIFIC DSN8.DSN8DUTINVV
LANGUAGE C
DETERMINISTIC
READS SQL DATA
EXTERNAL NAME DSN8DUTI
PARAMETER STYLE DB2SQL
NULL CALL
NO EXTERNAL ACTION
NO SCRATCHPAD
NO FINAL CALL
ALLOW PARALLEL
COLLID DSN8DU!!
ASUTIME LIMIT 10
STAY RESIDENT NO
PROGRAM TYPE MAIN
WLM ENVIRONMENT WLMENV
SECURITY DB2
NO DBINFO;

CREATE FUNCTION DSN8.TABLE_SCHEMA(
    VARCHAR(18) CCSID EBCDIC )
RETURNS
    VARCHAR(8)  CCSID EBCDIC
SPECIFIC DSN8.DSN8DUTISV
LANGUAGE C
DETERMINISTIC
READS SQL DATA
EXTERNAL NAME DSN8DUTI
PARAMETER STYLE DB2SQL
NULL CALL
NO EXTERNAL ACTION
NO SCRATCHPAD
NO FINAL CALL
ALLOW PARALLEL
COLLID DSN8DU!!
ASUTIME LIMIT 10
STAY RESIDENT NO
PROGRAM TYPE MAIN
WLM ENVIRONMENT WLMENV
SECURITY DB2
NO DBINFO;

CREATE FUNCTION
DSN8.TABLE_SCHEMA(
    VARCHAR(18) CCSID EBCDIC,
    VARCHAR(8) CCSID EBCDIC )
RETURNS
    VARCHAR(8) CCSID EBCDIC
SPECIFIC DSN8.DSN8DUTISVV
LANGUAGE C
DETERMINISTIC
READS SQL DATA
EXTERNAL NAME DSN8DUTI
PARAMETER STYLE DB2SQL
NULL CALL
NO EXTERNAL ACTION
NO SCRATCHPAD
NO FINAL CALL
ALLOW PARALLEL
COLLID DSN8DU!!
ASUTIME LIMIT 10
STAY RESIDENT NO
PROGRAM TYPE MAIN
WLM ENVIRONMENT WLMENV
SECURITY DB2
NO DBINFO;

CREATE FUNCTION
DSN8.TABLE_SCHEMA(
    VARCHAR(18) CCSID EBCDIC,
    VARCHAR(8) CCSID EBCDIC,
    VARCHAR(16) CCSID EBCDIC )
RETURNS
    VARCHAR(8) CCSID EBCDIC
SPECIFIC DSN8.DSN8DUTISVVV
LANGUAGE C
DETERMINISTIC
READS SQL DATA
EXTERNAL NAME DSN8DUTI
PARAMETER STYLE DB2SQL
NULL CALL
NO EXTERNAL ACTION
NO SCRATCHPAD
NO FINAL CALL
ALLOW PARALLEL
COLLID DSN8DU!!
ASUTIME LIMIT 10
STAY RESIDENT NO
PROGRAM TYPE MAIN
WLM ENVIRONMENT WLMENV
SECURITY DB2
NO DBINFO;

CREATE FUNCTION
DSN8.TABLE_LOCATION(
    VARCHAR(18) CCSID EBCDIC )
RETURNS
    VARCHAR(16) CCSID EBCDIC
SPECIFIC DSN8.DSN8DUTILV
LANGUAGE C
DETERMINISTIC
READS SQL DATA
EXTERNAL NAME DSN8DUTI
PARAMETER STYLE DB2SQL
NULL CALL
NO EXTERNAL ACTION
NO SCRATCHPAD
NO FINAL CALL
ALLOW PARALLEL
COLLID DSN8DU!!
ASUTIME LIMIT 10
STAY RESIDENT NO
PROGRAM TYPE MAIN
WLM ENVIRONMENT WLMENV
SECURITY DB2
NO DBINFO;
CREATE FUNCTION DSN8.TABLE_LOCATION(
    VARCHAR(18) CCSID EBCDIC,
    VARCHAR(8)  CCSID EBCDIC )
RETURNS
    VARCHAR(16) CCSID EBCDIC
SPECIFIC DSN8.DSN8DUTILV
LANGUAGE C
DETERMINISTIC
READS SQL DATA
EXTERNAL NAME DSN8DUTI
PARAMETER STYLE DB2SQL
NULL CALL
NO EXTERNAL ACTION
NO SCRATCHPAD
NO FINAL CALL
ALLOW PARALLEL
COLLID DSN8DU!!
ASUTIME LIMIT 10
STAY RESIDENT NO
PROGRAM TYPE MAIN
WLM ENVIRONMENT WLMENV
SECURITY DB2
NO DBINFO;

CREATE FUNCTION DSN8.TABLE_LOCATION(
    VARCHAR(18) CCSID EBCDIC,
    VARCHAR(8)  CCSID EBCDIC,
    VARCHAR(16) CCSID EBCDIC )
RETURNS
    VARCHAR(16) CCSID EBCDIC
SPECIFIC DSN8.DSN8DUTILV
LANGUAGE C
DETERMINISTIC
READS SQL DATA
EXTERNAL NAME DSN8DUTI
PARAMETER STYLE DB2SQL
NULL CALL
NO EXTERNAL ACTION
NO SCRATCHPAD
NO FINAL CALL
ALLOW PARALLEL
COLLID DSN8DU!!
ASUTIME LIMIT 10
STAY RESIDENT NO
PROGRAM TYPE MAIN
WLM ENVIRONMENT WLMENV
SECURITY DB2
NO DBINFO;

CREATE FUNCTION DSN8.WEATHER(
    VARCHAR(44) CCSID EBCDIC )
RETURNS
    TABLE(
        CITY     VARCHAR(30) CCSID EBCDIC,
        TEMP_IN_F INTEGER,
        HUMIDITY INTEGER,
        WIND     VARCHAR(5)  CCSID EBCDIC,
        WIND_VELOCITY INTEGER,
        BAROMETER FLOAT,
        FORECAST VARCHAR(25) CCSID EBCDIC )
SPECIFIC DSN8.DSN8DUWF
LANGUAGE C
DETERMINISTIC
NO SQL
EXTERNAL NAME DSN8DUWF
PARAMETER STYLE DB2SQL
NULL CALL
NO EXTERNAL ACTION
SCRATCHPAD
FINAL CALL
DISALLOW PARALLEL
NO COLLID
ASUTIME LIMIT 10
STAY RESIDENT NO
PROGRAM TYPE SUB
WLM ENVIRONMENT WLMENV
SECURITY DB2
NO DBINFO;
GRANT EXECUTE ON SPECIFIC FUNCTION DSN8.DSN8DUADV,
  DSN8.DSN8DUCDVVV,
  DSN8.DSN8DUCDDVV,
  DSN8.DSN8EUADV,
  DSN8.DSN8DUCYFV,
  DSN8.DSN8DUCYFVV,
  DSN8.DSN8EUADV,
  DSN8.DSN8DUCFVV,
  DSN8.DSN8DUCFDV,
  DSN8.DSN8EUADV,
  DSN8.DSN8DUCFDVV,
  DSN8.DSN8DUCNDDV,
  DSN8.DSN8DUCNDDVV,
  DSN8.DSN8DUCNDDV,
  DSN8.DSN8DUCNDDVV,
  DSN8.DSN8DUCNDDV,
  DSN8.DSN8DUCNDDVV,
  DSN8.DSN8DUCNDDV,
  DSN8.DSN8DUCNDDVV,
  DSN8.DSN8DUCNDDV,
  DSN8.DSN8DUCNDDVV,
  DSN8.DSN8DUCNDDV,
  DSN8.DSN8DUCNDDVV,
  DSN8.DSN8DUCNDDV,
  DSN8.DSN8DUCNDDVV,
  DSN8.DSN8DUCNDDV,
  DSN8.DSN8DUCNDDVV,
  DSN8.DSN8DUCNDDV,
  DSN8.DSN8DUCNDDVV,
  DSN8.DSN8DUCNDDV,
  DSN8.DSN8DUCNDDVV,
  DSN8.DSN8DUCNDDV,
  DSN8.DSN8DUCNDDVV,
  DSN8.DSN8DUCNDDV,
  DSN8.DSN8DUCNDDVV,
  DSN8.DSN8DUCNDDV,
  DSN8.DSN8DUCNDDVV,
  DSN8.DSN8DUCNDDV,
  DSN8.DSN8DUCNDDVV,
  DSN8.DSN8DUCNDDV,
  DSN8.DSN8DUCNDDVV,
  DSN8.DSN8DUCNDDV,
  DSN8.DSN8DUCNDDVV,
  DSN8.DSN8DUCNDDV,
  DSN8.DSN8DUCNDDVV,
  DSN8.DSN8DUCNDDV,
  DSN8.DSN8DUCNDDVV,
  DSN8.DSN8DUCNDDV,
  DSN8.DSN8DUCNDDVV,
  DSN8.DSN8DUCNDDV,
  DSN8.DSN8DUCNDDVV,
  DSN8.DSN8DUCNDDV,
  DSN8.DSN8DUCNDDVV,
  DSN8.DSN8DUCNDDV,
  DSN8.DSN8DUCNDDVV,
  DSN8.DSN8DUCNDDV,
  DSN8.DSN8DUCNDDVV,
  DSN8.DSN8DUCNDDV,
  DSN8.DSN8DUCNDDVV,
  DSN8.DSN8DUCNDDV,
  DSN8.DSN8DUCNDDVV,
  DSN8.DSN8DUCNDDV,
  DSN8.DSN8DUCNDDVV,
  DSN8.DSN8DUCNDDV,
  DSN8.DSN8DUCNDDVV,
  DSN8.DSN8DUCNDDV,
  DSN8.DSN8DUCNDDVV,
  DSN8.DSN8DUCNDDV,
  DSN8.DSN8DUCNDDVV,
  DSN8.DSN8DUCNDDV,
  DSN8.DSN8DUCNDDVV,
  DSN8.DSN8DUCNDDV,
  DSN8.DSN8DUCNDDVV,
  DSN8.DSN8DUCNDDV,
  DSN8.DSN8DUCNDDVV,
  DSN8.DSN8DUCNDDV,
  DSN8.DSN8DUCNDDVV,
  DSN8.DSN8DUCNDDV,
  DSN8.DSN8DUCNDDVV,
  DSN8.DSN8DUCNDDV,
  DSN8.DSN8DUCNDDVV,
  DSN8.DSN8DUCNDDV,
  DSN8.DSN8DUCNDDVV,
  DSN8.DSN8DUCNDDV,
  DSN8.DSN8DUCNDDVV,
  DSN8.DSN8DUCNDDV,
  DSN8.DSN8DUCNDDVV,
  DSN8.DSN8DUCNDDV,
  DSN8.DSN8DUCNDDVV,
  DSN8.DSN8DUCNDDV,
  DSN8.DSN8DUCNDDVV,
  DSN8.DSN8DUCNDDV,
  DSN8.DSN8DUCNDDVV,
  DSN8.DSN8DUCNDDV,
  DSN8.DSN8DUCNDDVV,
  DSN8.DSN8DUCNDDV,
  DSN8.DSN8DUCNDDVV,
  DSN8.DSN8DUCNDDV,
  DSN8.DSN8DUCNDDVV,
  DSN8.DSN8DUCNDDV,
  DSN8.DSN8DUCNDDVV,
  DSN8.DSN8DUCNDDV,
  DSN8.DSN8DUCNDDVV,
  DSN8.DSN8DUCNDDV,
  DSN8.DSN8DUCNDDVV,
  DSN8.DSN8DUCNDDV,
  DSN8.DSN8DUCNDDVV,
  DSN8.DSN8DUCNDDV,
  DSN8.DSN8DUCNDDVV,
  DSN8.DSN8DUCNDDV,
Chapter 12. Sample data and applications supplied with Db2 for z/OS 1339
NAME DSN8EUMN(R)

/**
 * STEP 10: PREPARE EXTERNAL FOR TABLE_NAME, TABLE_SCHEMA,
 * AND TABLE_LOCATION UDF'S
 */
/**
 * PH02US10 EXEC DSNHC, MEM=DSN8DUTI, COND=(4, LT),
 * PARM.PC=('HOST(C), CCSID(1047), MARGINS(1,72), STDSQL(NO)',
 * 'SOURCE,XREF'),
 * PARM.C='SOURCE RENT XREF MARGINS(1,72) OPTFILE(DD:CCOPTS)'),
 * PARM.LKED='MAP, RENT, REUS, AMODE=31, RMODE=ANY'
 */
// PC.DBRMLIB DD DSN=DSN!!0.DBRMLIB.DATA(DSN8DUTI),
// DISP=SHR
// PC.SYSLIB DD DSN=DSN!!0.SRCLIB.DATA,
// DISP=SHR
// PC.SYSIN DD DSN=DSN!!0.SDSNSAMP(DSN8DUTI),
// DISP=SHR
// LKED.SYSLMOD DD DSN=DSN!!0.RUNLIB.LOAD(DSN8DUTI),
// DISP=SHR
// LKED.IGNORE DD *
// LKED.SYSLIN DD *
// INCLUDE SYSLIB(DSNRLI)
NAME DSN8DUTI(R)

/**
 * STEP 11: BIND PACKAGE FOR TABLE_NAME, TABLE_SCHEMA, AND
 * TABLE_LOCATION UDF'S
 */
/**
 * PH02US11 EXEC PGM=IKJEFT01, COND=(4, LT)
 */
// DBRMLIB DD DSN=DSN!!0.DBRMLIB.DATA, DISP=SHR
// SYSTSPRT DD SYSOUT=*
// SYSPRINT DD SYSOUT=*
// SYSDUMP DD SYSOUT=*
// SYSEXCEPTION DD SYSOUT=*
// SYSCURRENT DD SYSOUT=*
// SYSTSPRT DD *
// SYSTSPRT DD *
// DSNSYSTEM(DSN)
// BIND PACKAGE (DSN8DUTI) MEMBER(DSN8DUTI) APPLCOMPAT(V!!R1) +
// ACT(REP) ISO(CS) CURRENTDATA(YES) ENCODING(EBCDIC)
END

/**
 * STEP 12: EXERCISE THE SAMPLE UDF'S
 */
/**
 * PH02US12 EXEC PGM=IKJEFT01, COND=(4, LT), DYNAMNBR=20
 */
// SYSTSPRT DD SYSOUT=*
// SYSPRINT DD SYSOUT=*
// SYSDUMP DD SYSOUT=*
// SYSEXCEPTION DD SYSOUT=*
// SYSCURRENT DD SYSOUT=*
// SYSTSPRT DD *
// DSNSYSTEM(DSN)
// RUN PROGRAM(DSNTEP2) PLAN(DSNTEP!!) -
// LIB('DSN!!0.RUNLIB.LOAD') PARMS('/ALIGN(MID)')
END
// SYSCURRENT DD *
// SET CURRENT SQLID = 'SYSADM';
// DD DSN=DSN!!0.SDSNSAMP(DSNTESU),
// DISP=SHR

/**
 * STEP 13: PREPARE EXTERNAL FOR WEATHER UDF TABLE FUNCTION
 */
/**
 * PH02US13 EXEC DSNHC, MEM=DSN8DUWF, COND=(4, LT),
 * PARM.PC=('HOST(C), CCSID(1047), MARGINS(1,72), STDSQL(NO)',
 * 'SOURCE,XREF'),
 * PARM.C='SOURCE RENT XREF MARGINS(1,72) OPTFILE(DD:CCOPTS)'),
 * PARM.LKED='MAP, RENT, REUS, AMODE=31, RMODE=ANY'
 */
// PC.DBRMLIB DD DSN=DSN!!0.DBRMLIB.DATA(DSN8DUWF),
// DISP=SHR
// PC.SYSLIB DD DSN=DSN!!0.SRCLIB.DATA,
// DISP=SHR
// PC.SYSIN DD DSN=DSN!!0.SDSNSAMP(DSN8DUWF),
// DISP=SHR
// LKED.SYSLMOD DD DSN=DSN!!0.RUNLIB.LOAD(DSN8DUWF),
// DISP=SHR
// LKED.IGNORE DD *
// LKED.SYSLIN DD *
// INCLUDE SYSLIB(DSNRLI)
NAME DSN8DUWF(R)

/**
 * STEP 14: PREPARE CLIENT FOR WEATHER UDF TABLE FUNCTION
 */
/**
 * PH02US14 EXEC DSNHC, MEM=DSN8DUCW, COND=(4, LT),
 * PARM.PC=('HOST(C), CCSID(1047), MARGINS(1,72), STDSQL(NO)',
 * 'SOURCE,XREF'),
 */
Related reference
“Sample applications in TSO” on page 1027
A set of Db2 sample applications run in the TSO environment.

**DSNTEJ71**
PREPARES AND RUNS THE FOLLOWING PROGRAMS IN SUPPORT OF THE Db2 LOB SAMPLE C APPLICATION.
**FUNCTION** = PREPARES AND RUNS THE FOLLOWING PROGRAMS IN SUPPORT OF THE DB2 LOB SAMPLE C APPLICATION:

- **DSN8DLPL**: POPULATES THE PSEG AND BMP IMAGE COLUMNS IN THE DSN8!!0.EMP_PHOTO_RESUME SAMPLE LOB TABLE. THE INPUT DATA IS READ FROM A TSO DATA SET. THIS PROGRAM DEMONSTRATES HOW TO POPULATE LOB COLUMNS WITH MORE THAN 32 KB OF DATA.

- **DSN8DLTC**: VALIDATES THE CONTENTS OF THE LOB COLUMNS IN THE DSN8!!0.EMP_PHOTO_RESUME TABLE. THIS IS DONE BY COMPARING THE DATA IN THE TABLE TO THE SOURCE DATA SETS.

**CHANGE ACTIVITY** =

08/18/2014 Single-phase migration s21938_inst1 s21938

*******************************

**JOBLIB DD** DSN=DSN!!0.SDSNEXIT,DISP=SHR
** DD DSN=DSN!!0.SDSNLOAD,DISP=SHR
** DD DSN=CEE.V!R!M!.SCEERUN,DISP=SHR
** DD DSN=DSN!!0.RUNLIB.LOAD,DISP=SHR
**
** STEP 1: PREPARE LOADER FOR EMPLOYEE PHOTO IMAGES
**
**/PH071S01 EXEC DSNHC,MEM=DSN8DLPL,COND=(4,LT),
// PARM.PC=('HOST(C),CCSID(1847),MARGINS(1,72),STDSQL(NO)',
// SOURCE,XREF),
// PARM.C='SOURCE RENT XREF MARGINS(1,72) OPTFILE(DD:CCOPTS)',
// PARM.LKED='MAP,RENT,REUS,AMODE=31,RMODE=ANY'
// PC.DBRMLIB DD DSN=DSN!!0.DBRMLIB.DATA(DSN8DLPL),
// DISP=SHR
// PC.SYSLIB DD DSN=DSN!!0.SRCLIB.DATA,
// DISP=SHR
// PC.SYSIN DD DSN=DSN!!0.SDSNSAMP(DSN8DLPL),
// DISP=SHR
// LKED.SYSLMOD DD DSN=DSN!!0.RUNLIB.LOAD(DSN8DLPL),
// DISP=SHR
// LKED.SYSIN DD *
//INCLUDE SYSLIB(DSNELI)
NAME DSN8DLPL(R)
**
** STEP 2: PREPARE SAMPLE LOB TABLE VALIDATOR
**
**/PH071S02 EXEC DSNHC,MEM=DSN8DLTC,COND=(4,LT),
// PARM.PC=('HOST(C),CCSID(1847),MARGINS(1,72),STDSQL(NO)',
// SOURCE,XREF),
// PARM.C='SOURCE RENT XREF MARGINS(1,72) OPTFILE(DD:CCOPTS)',
// PARM.LKED='MAP,RENT,REUS,AMODE=31,RMODE=ANY'
// PC.DBRMLIB DD DSN=DSN!!0.DBRMLIB.DATA(DSN8DLTC),
// DISP=SHR
// PC.SYSLIB DD DSN=DSN!!0.SRCLIB.DATA,
// DISP=SHR
// PC.SYSIN DD DSN=DSN!!0.SDSNSAMP(DSN8DLTC),
// DISP=SHR
// LKED.SYSLMOD DD DSN=DSN!!0.RUNLIB.LOAD(DSN8DLTC),
// DISP=SHR
// LKED.SYSIN DD *
//INCLUDE SYSLIB(DSNELI)
NAME DSN8DLTC(R)
**
** STEP 3: BIND PACKAGES AND PLANS FOR DSN8DLPL AND DSN8DLTC
**
**/PH071S03 EXEC PGM=IKJEFT01,COND=(4,LT)
//DBRMLIB DD DSN=DSN!!0.DBRMLIB.DATA,DISP=SHR
//SYSTSPT DD SYSOUT=*
//SYSPRINT DD SYSOUT=*
//CDEUDUMP DD SYSOUT=*
//SYSDUMP DD SYSOUT=*
//REPORT DD SYSOUT=*
//SYSIN DD *
//SET CURRENT SQLID = 'SYSADM';
//GRANT BIND, EXECUTE ON PLAN DSNBLC!!, DSNBLL!! TO PUBLIC;
//SYSTSIN DD *
//DSN SYSTEM(DSN)
//BIND PACKAGE (DSNBLL!!) MEMBER(DSN8DLPL) APPLCOMPAT(V1R1) + QUALIFIER(DSNB!!0) - ACT(REP) ISO(CS) CURRENTDATA(YES) ENCODING(EBCDIC)
//BIND PLAN(DSNBLL!!) PKLIST(DSNBLL!!.* ) - ACTION(REPLACE) RETAIN +
Related reference
“Sample applications in TSO” on page 1027
A set of Db2 sample applications run in the TSO environment.

**DSNTEJ73**

PREPARES AND RUNS THE FOLLOWING PROGRAMS IN SUPPORT OF THE Db2 LOB SAMPLE C APPLICATION.

---

Chapter 12. Sample data and applications supplied with Db2 for z/OS 1343
** LICENSED MATERIALS - PROPERTY OF IBM
** 5650-DB2
** (C) COPYRIGHT 1982, 2016 IBM CORP. ALL RIGHTS RESERVED.
**
** STATUS = VERSION 12
**
** FUNCTION = PREPARES AND RUNS THE FOLLOWING PROGRAMS IN SUPPORT
** OF THE DB2 LOB SAMPLE C APPLICATION:
** - DSN8SDM: CAF CONNECTION MANAGER (C LANGUAGE), USED
** TO INVOKED THE DB2 SAMPLE APPLICATIONS MENU
** UNDER ISPF AND TO MANAGE INVOKATION OF THE
** DB2 SAMPLE APPLICATION PROGRAMS, INCLUDING
** THE LOB SAMPLE RESUME AND PHOTO IMAGE
** VIEWERS.
** - DSN8DLRV: EXTRACTS A SPECIFIED EMPLOYEE'S RESUME IN
** CLOB FORMAT FROM DSN8!!0.EMP_PHOTO_RESUME.
** DB2 LOB LOCATOR FUNCTIONS ARE USED TO PARSE
** DATA FROM CLOB FORMAT INTO ISPF FIELDS AND
** THE RESULT IS DISPLAYED TO THE USER.
**
** CHANGE ACTIVITY =
** 08/18/2014 Single-phase migration s21938_inst1 s21938
**
** *********************************************************************
** JOBLIB DD DSN=DSN!!0.SDSNEXIT,DISP=SHR
** DD DSN=DSN!!0.SDSNLOAD,DISP=SHR
** DD CEE.V!R!M!.SCEERUN,DISP=SHR
** DD DSN=DSN!!0.RUNLIB.LOAD,DISP=SHR
**
** ** STEP 1: PREPARE SAMPLE CALL ATTACH CONTROLLER
**
** PH073S01 EXEC DSNHC,MEM=DSN8SDM,COND=(4,LT),
** PARM.PC=('HOST(C),CCSID(1047),MARGINS(1,72),STDSQL(NO)',
** SOURCE,XREF),
** PARM.C='SOURCE RENT XREF MARGINS(1,72) OPTFILE(DD:CCOPTS)',
** PARM.LKED='MAP,RENT,REUS,AMODE=31,RMODE=ANY'
**
** PC.DBRMLIB DD DSN=DSN!!0.DBRMLIB.DATA(DSN8SDM),
** DISP=SHR
** PC.SYSLIB DD DSN=DSN!!0.SRCLIB.DATA,
** DISP=SHR
** PC.SYSIN DD DSN=DSN!!0.SDSNSAMP(DSN8SDM),
** DISP=SHR
**
** LKED.SYSLMOD DD DSN=DSN!!0.RUNLIB.LOAD(DSN8SDM),
** DISP=SHR
** LKED.SYSIN DD *
** INCLUDE SYSLIB(DSNALI)
** NAME DSN8SDM(R)
**
** ** STEP 2: PREPARE EMPLOYEE RESUME VIEWER (ISPF)
**
** PH073S02 EXEC DSNHC,MEM=DSN8DLRV,COND=(4,LT),
** PARM.PC=('HOST(C),CCSID(1047),MARGINS(1,72),STDSQL(NO)',
** SOURCE,XREF),
** PARM.C='SOURCE RENT XREF MARGINS(1,72) OPTFILE(DD:CCOPTS)',
** PARM.LKED='MAP,RENT,REUS,AMODE=31,RMODE=ANY'
**
** PC.DBRMLIB DD DSN=DSN!!0.DBRMLIB.DATA(DSN8DLRV),
** DISP=SHR
** PC.SYSLIB DD DSN=DSN!!0.SRCLIB.DATA,
** DISP=SHR
** PC.SYSIN DD DSN=DSN!!0.SDSNSAMP(DSN8DLRV),
** DISP=SHR
**
** LKED.SYSLMOD DD DSN=DSN!!0.RUNLIB.LOAD(DSN8DLRV),
** DISP=SHR
** LKED.SYSIN DD *
** INCLUDE SYSLIB(DSNALI)
** NAME DSN8DLRV(R)
**
** ** STEP 3: BIND PACKAGE AND PLAN FOR THE RESUME VIEWER
**
** PH073S03 EXEC PGM=IKJEFT01,COND=(4,LT)
** DBRMLIB DD DSN=DSN!!0.DBRMLIB.DATA,DISP=SHR
** SYSSTSPRT DD SYSOUT=* SYSTEM(DSN)
** SYSPRINT DD SYSOUT=* SYSTEM(DSN)
** CEEDEMP DD SYSOUT=* SYSTEM(DSN)
** SYSSDUMP DD SYSOUT=* SYSTEM(DSN)
** REPORT DD SYSOUT=* SYSTEM(DSN)
** SYSTSEC DD *
** BIND PACKAGE (DSN8LR!!) APPLCOMPAT(V!!R1) + 
DSNTEJ75

PREPARES AND RUNS THE FOLLOWING PROGRAM IN SUPPORT OF THE Db2 LOB SAMPLE C APPLICATION.

```sql
//************************************************************
// ** NAME = DSNTEJ75
// ** DESCRIPTIVE NAME = DB2 SAMPLE APPLICATION
// ** PHASE 7
// ** SAMPLE APPLICATIONS: VIEW, MANIPULATE BLOB DATA
// ** C LANGUAGE
// ** LICENSED MATERIALS - PROPERTY OF IBM
// ** 5650-DB2
// ** (C) COPYRIGHT 1982, 2016 IBM CORP. ALL RIGHTS RESERVED.
// ** STATUS = VERSION 12
// ** FUNCTION = PREPARES AND RUNS THE FOLLOWING PROGRAM IN SUPPORT
// ** OF THE DB2 LOB SAMPLE C APPLICATION:
// ** - DSN8DLPV: EXTRACTS A SPECIFIED EMPLOYEE'S PSEG PHOTO
// ** IMAGE IN BLOB FORMAT FROM THE SAMPLE TABLE
// ** DSN8!!0.EMP_PHOTO_RESUME. THE DATA IS
// ** HANDED OFF TO GDDM FOR CONVERSION FOR CON-
// ** VERSION AND DISPLAY.
// ** CHANGE ACTIVITY =
// ** 08/18/2014 Single-phase migration s21938_inst1 s21938
// ** JOBLIB DD DSN=DSN!!0.SDSNEXIT,DISP=SHR
// ** DD DSN=DSN!!0.SDSNLOAD,DISP=SHR
// ** DD DSN=CEE.V!R!M!.SCEERUN,DISP=SHR
// ** DD DSN=DSN!!0.RUNLIB.LOAD,DISP=SHR
// ** STEP 1: PREPARE EMPLOYEE PHOTO VIEWER (GDDM)
// ** EXEC DSNHC,MEM=DSN8DLPV,COND=(4,LT),
// ** PARM.PC=('HOST(C),CCSID(1047),MARGINS(1,72),STDSQL(NO)',
// ** SOURCE,XREF),
// ** PARM.C='SOURCE RENT XREF MARGINS(1,72) OPTFILE/DD:CCOPTS',
// ** PARM.LKED='MAP,RENT,REUS,AMODE=31,RMODE=ANY'
// ** PC.DBRMLIB DD DSN=DSN!!0.DBRMLIB.DATA(DSN8DLPV),
// ** DISP=SHR
// ** PC.SYSLIB DD DSN=DSN!!0.SRCLIB.DATA,
// ** DISP=SHR
// ** PC.SYSIN DD DSN=DSN!!0.SDONSAMP(DSN8DLPV),
// ** DISP=SHR
// ** LKED.SYSLMOD DD DSN=DSN!!0.RUNLIB.LOAD(DSN8DLPV),
// ** DISP=SHR
// ** LKED.SYSLMOD DD DSN=DSN!!0.RUNLIB.LOAD(DSN8DLPV),
// ** DISP=SHR
// ** INCLUDE SYSLIB(ADMASRT)
// ** INCLUDE SYSLIB(DSN8DLRVR)
// ** INCLUDE SYSLIB(DSN8DLRVR)
// ** NAME DSN8DLPV(R)
// ** STEP 2: BIND PACKAGE AND PLAN FOR THE PHOTO VIEWER
```

Related reference

“Sample applications in TSO” on page 1027

A set of Db2 sample applications run in the TSO environment.
Related reference  
“Sample applications in TSO” on page 1027  
A set of Db2 sample applications run in the TSO environment.

Sample applications in IMS  
A set of Db2 sample applications run in the IMS environment.

Table 183. Sample Db2 applications for IMS

<table>
<thead>
<tr>
<th>Application</th>
<th>Program name</th>
<th>JCL member name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Organization</td>
<td>DSN8IC0</td>
<td>DSNTEJ4C</td>
<td>IMS COBOL Organization</td>
</tr>
<tr>
<td></td>
<td>DSN8IC1</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>DSN8IC2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Organization</td>
<td>DSN8IP0</td>
<td>DSNTEJ4P</td>
<td>IMS PL/I Organization</td>
</tr>
<tr>
<td></td>
<td>DSN8IP1</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>DSN8IP2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Project</td>
<td>DSN8IP6</td>
<td>DSNTEJ4P</td>
<td>IMS PL/I Project Application</td>
</tr>
<tr>
<td></td>
<td>DSN8IP7</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>DSN8IP8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Phone</td>
<td>DSN8IP3</td>
<td>DSNTEJ4P</td>
<td>IMS PL/I Phone Application. This program lists employee telephone numbers and updates them if requested.</td>
</tr>
</tbody>
</table>

Related reference  
Data sets that the precompiler uses
When you invoke the precompiler you need to provide data sets that contain input for the precompiler, such as the host programming statements and SQL statements. You also need to provide data sets where the precompiler can store its output, such as the modified source code and diagnostics messages.

### DSN8IC0

**THIS MODULE RECEIVES AN INPUT MESSAGE AND DEFORMATS IT, CALLS DSN8IC1, FORMATS OUTPUT MESSAGE AND SENDS IT.**

```
IDENTIFICATION DIVISION.

PROGRAM-ID. DSN8IC0.

****** DSN8IC0 - IMS SUBSYSTEM INTERFACE MODULE - COBOL *******

* MODULE NAME = DSN8IC0

* DESCRIPTIVE NAME = DB2 SAMPLE APPLICATION

* SUBSYSTEM INTERFACE MODULE

* IMS

* COBOL

* ORGANIZATION APPLICATION

* LICENSED MATERIALS - PROPERTY OF IBM

*5615-DB2

*(C) COPYRIGHT 1996, 2013 IBM CORP. ALL RIGHTS RESERVED.

* STATUS = VERSION 11

* FUNCTION = THIS MODULE RECEIVES AN INPUT MESSAGE AND

* DEFORMATS IT, CALLS DSN8IC1,

* FORMATS OUTPUT MESSAGE AND SENDS IT.

* NOTES = NONE

* MODULE TYPE = FROM IMS

* LINKAGE = FROM IMS

* INPUT = PARAMETERS EXPLICITLY PASSED TO THIS FUNCTION:

* SYMBOICL LABEL/NAME = DSN8IC1G

* DESCRIPTION = IMS/VS MFS GENERAL MENU

* SYMBOICL LABEL/NAME = DSN8IC1D

* DESCRIPTION = IMS/VS MFS DETAIL MENU

* OUTPUT = PARAMETERS EXPLICITLY RETURNED:

* SYMBOICL LABEL/NAME = DSN8ICG0

* DESCRIPTION = IMS/VS MFS GENERAL MENU

* SYMBOICL LABEL/NAME = DSN8ICDO

* DESCRIPTION = IMS/VS MFS DETAIL MENU

* EXIT-NORMAL = RETURN CODE 0 NORMAL COMPLETION

* EXIT-ERROR =

* RETURN CODE = NONE

* ABEND CODES = NONE

* ERROR-MESSAGES =

* DSN8064E - INVALID DL/I STC-CODE ON GU MSG

* DSN8065E - INVALID DL/I STC-CODE ON ISRT MSG

* EXTERNAL REFERENCES =

* ROUTINES/SERVICES = MODULE DSN8IC1

* MODULE CBLTDLI

* MODULE DSN8MC6

* DATA-AREAS =

* DSN8MCCA - PARAMETER TO BE PASSED TO DSN8IC1* 00640000
```

Chapter 12. Sample data and applications supplied with Db2 for z/OS  1347
DECLARATION FOR OUTPUT: MODNAME DSN8ICGO/DSN8ICDO

DATA DIVISION.
01 COMMAREA.
  EXEC SQL INCLUDE DSN8MCCA END-EXEC.
  DECLARATION FOR INPUT/OUTPUT DATA BETWEEN THE
  SUBSYSTEM DEPENDENT MODULE IMS/DLI AND SQL1 AND SQL2.

ENVIRONMENT DIVISION.

WORKING-STORAGE SECTION.
DECLARATION FOR PASSING INPUT/OUTPUT DATA BETWEEN THE
SUBSYSTEM DEPENDENT MODULE IMS/DLI AND SQL1 AND SQL2.

DATA DIVISION.

IN-MESSAGE - MFS INPUT
OUT-MESSAGE - MFS OUTPUT
CONTROL-BLOCKS =

CHANGE-ACTIVITY = 05/18/2012: SWITCH ARITHMETICS FROM COMP TO COMP-5 PM66408

PSEUDOCODE

PROEDURE

PROCEDURE.

************
ENVIRONMENT DIVISION.
------------------------
WORKING-STORAGE SECTION.
DECLARATION FOR PASSING INPUT/OUTPUT DATA BETWEEN THE
SUBSYSTEM DEPENDENT MODULE IMS/DLI AND SQL1 AND SQL2.

DATA DIVISION.

IN-MESSAGE.
02 LL PIC S9(3) COMP-5.
02 Z1 PIC X.
02 Z2 PIC X.
02 TC-CODE PIC X(7).
02 IN-PUT. PIC X(668).
03 MAJSYS PIC X.
03 ACTION PIC X.
03 OBJFILD PIC X(2).
03 SRCCH PIC X(2).
03 PFKIN PIC X.
03 DATAIN PIC X(60).
03 TRANDATA PIC X(48) OCCURS 15.
02 IN-PUT0 REDEFINES IN-PUT PIC X(668).

DECLARATION FOR OUTPUT: MODNAME DSN8ICGO/DSN8ICDO

OUT-MESSAGE.
02 LL PIC S9(3) COMP-5.
02 ZZ PIC S9(3) COMP-5 VALUE +0.
02 OUTPUTAREA.
 03 MAJSYS PIC X.
 03 ACTION PIC X.
 03 OBJFJD PIC X(2).
 03 SRC PIC X(2).
 03 DATAOUT PIC X(60).
 03 HTITLE PIC X(50).
 03 DESC2 PIC X(50).
 03 DESC3 PIC X(50).
 03 DESC4 PIC X(50).
 03 MSG.
   05 STC PIC X(4).
   05 MSGTEXT PIC X(75).
   03 PFTEXT PIC X(79).
   03 OUTPUT PIC X(69).
   03 LINE0 PIC X(79).
* 02 OUTPUTAREA O REDEFINES OUTPUTAREA PIC X(169).
*******************************************************************************
* FIELDS SENT TO MESSAGE ROUTINE                                          *
*******************************************************************************
 01 MSGCODE PIC X(84).
 01 OUTMSG PIC X(69).
*******************************************************************************
* DECLARATION FOR PGM-LOGIC                                                 *
*******************************************************************************
01 OUTMSG.
 77 MODNAME PIC X(8).
*******************************************************************************
* LINKAGE SECTION                                                          *
*******************************************************************************
 77 USERID PIC X(8).
 77 SEQNUM PIC X(4).
 77 CTIME PIC X(4).
 77 CDATE PIC X(4).
 77 STC CODE PIC X(2).
 77 FILLER PIC X(2).
 77 IOLTERM PIC X(8).
 77 ROLL-FKT PIC X(4) VALUE 'ROLL'.
 77 CHNG-FKT PIC X(4) VALUE 'CHNG'.
 77 ISRT-FKT PIC X(4) VALUE 'ISRT'.
 77 GU-FKT PIC X(4) VALUE 'GU'.
*******************************************************************************
* DECLARATION FOR IO / ALTPCB                                               *
*******************************************************************************
01 IOPCB.
 02 IOLTERM PIC X(8).
 02 FILLER PIC X(2).
 02 STC-TERM PIC X(2).
 02 DATE PIC X(4).
 02 TIME PIC X(4).
 02 SEQNUM PIC X(4).
 02 MOD-NAME PIC X(8).
 02 USERID PIC X(8).
* 01 ALTPCB.
 02 ALTTERM PIC X(8).
 02 FILLER PIC X(2).
 02 STC-TERM PIC X(2).
 02 USERID PIC X(8).
**********************************************************************
PROCEDURE DIVISION.
*---------------------------------------------------------------*
* ENTRY 'DLITCBL' USING IOPCB ALTPCB.                           *
* ALLOCATE COBOL WORK AREA /INITIALIZATIONS                     *
* CSTART.                                                        *
* MOVE SPACES TO COMMAREA.                                      *
* MOVE SPACES TO IN-MESSAGE.                                    *
* MOVE 'DSN0ICG0' TO MAJSYS.                                     *
* MOVE 'DSN0IC80' TO MAJOR IN DSN8-MODULE-NAME.                  *
* MOVE 'O' TO MAJSYS IN OUTAREA.                                 *
* MOVE 'O' TO EXITCODE.                                         *
* MOVE +1613 TO LL IN OUT-MESSAGE.                              *
* CALL DL1 GU INPUT MESSAGE                                      *
* PRINT ERROR MESSAGE IF STATUS CODE NOT OK                     *
* **CALL DL1 GU                                                   *

Chapter 12. Sample data and applications supplied with Db2 for z/OS 1349
CALL 'CBLTDLI' USING GU-FKT IOPCB IN-MESSAGE.

 IF STC-CODE IN IOPCB NOT = ' ' **ERROR?
 THEN MOVE '064E' TO MSGCODE
 CALL 'DSN8MCG' USING MAJOR MSGCODE OUTMSG
 MOVE OUTMSG TO MSGTEXT IN OUTPUTAREA
 GO TO CSEND.

****************************************************************
* CLEARED AND UNFORMATTED SCREEN? *
****************************************************************

 IF Z2 = LOW-VALUE
 THEN MOVE '00' TO PKF1N IN INAREA.
 MOVE IOLTERM IN IOPCB TO TRMID IN CONVID.
 MOVE USERID IN IOPCB TO USERID IN CONVID.

 **MOVE INPUT MESSAGE FIELDS TO INAREA FIELDS**
 MOVE IN-PUT0 TO INAREA0.
 MOVE '0' TO MAJSYS IN INAREA.

 CALL 'DSN8IC1' USING COMMAREA.

 **MOVE OUTAREA FIELDS TO OUTPUT MESSAGE FIELDS**
 MOVE OUTAREA0 TO OUTPUTAREA0.

 IF LASTSCR = 'DSN8002'
 THEN MOVE 'DSN8IC00' TO MODNAME
 ELSE MOVE 'DSN8ICGO' TO MODNAME.

****************************************************************
* CALL DL ISRT OUTPUT MESSAGE *
* PRINT ERROR MESSAGE IF STATUS CODE NOT OK *
****************************************************************

 CALL 'CBLTDLI' USING ISRT-FKT IOPCB OUT-MESSAGE MODNAME.

 IF STC-CODE IN IOPCB = ' ' THEN GO TO CEND.

 **STATUS CODE NOT OK**
 **PRINT ERROR MESSAGE**
 MOVE '065E' TO MSGCODE.
 CALL 'DSN8MCG' USING MAJOR MSGCODE OUTMSG.
 MOVE OUTMSG TO MSGTEXT IN OUTPUTAREA.
 MOVE STC-CODE IN IOPCB TO STC IN OUTPUTAREA.

 CALL 'CBLTDLI' USING CHNG-FKT ALTPCB IOLTERM.

 IF STC-CODE IN ALTPCB NOT = ' ' **ERROR?
 GO TO CSEND1.

 CALL 'CBLTDLI' USING ISRT-FKT IOPCB OUT-MESSAGE MODNAME.

 **PERFORM ROLLBACK**
 CALL 'CBLTDLI' USING ROLL-FKT.

 **RETURN**

 Related reference
 “Sample applications in IMS” on page 1346
A set of Db2 sample applications run in the IMS environment.

**DSN8IC1**

THIS MODULE RETRIEVES THE ROW CONTAINING INFORMATION ON THE CURRENT CONVERSATION, VALIDATES SELECTION CRITERIA, AND ISSUES MESSAGES TO COMPLETE THE ACTION, OBJECT, AND SEARCH CRITERIA.

```
IDENTIFICATION DIVISION.
*------------------------
PROGRAM-ID. DSN8IC1.

*********** DSN8IC1 - SQL 1 MAINLINE FOR IMS - COBOL ************
* MODULE NAME = DSN8IC1
* DESCRIPTIVE NAME = DB2 SAMPLE APPLICATION
* SQL 1 MAINLINE
* IMS
* COBOL

*COPYRIGHT = 5615-DB2 (C) COPYRIGHT IBM CORP 1982, 2013
*REFER TO COPYRIGHT INSTRUCTIONS FORM NUMBER G120-2083

* STATUS = VERSION 11
*
* FUNCTION = THIS MODULE RETRIEVES THE ROW CONTAINING
* INFORMATION ON THE CURRENT CONVERSATION,
* VALIDATES SELECTION CRITERIA, AND ISSUES
* MESSAGES TO COMPLETE THE ACTION, OBJECT,
* AND SEARCH CRITERIA.

* NOTES = NONE
*
* MODULE TYPE =
* PROCESSOR   = DB2 PRECOMPILER, COBOL COMPILER
* MODULE SIZE = SEE LINKEDIT
* ATTRIBUTES  = REUSABLE
*
* ENTRY POINT = DSN8IC1
* PURPOSE = SEE FUNCTION
* LINKAGE = CALLED BY DSN8IC0
*
* INPUT = PARAMETERS EXPLICITLY PASSED TO THIS FUNCTION:
* *
* SYMBOLIC LABEL/NAME = COMMPTR
* DESCRIPTION = POINTER TO COMMAREA
*
* SYMBOLIC LABEL/NAME = INAREA
* DESCRIPTION = USER INPUT
*
* SYMBOLIC LABEL/NAME = PFKN
* DESCRIPTION = 00/01/02/03/07/08/10/11
*
* OUTPUT = PARAMETERS EXPLICITLY RETURNED:
* *
* SYMBOLIC LABEL/NAME = OUTAREA
* DESCRIPTION = GENERAL MENU OR SECONDARY
*   SELECTION MENU
*
* SYMBOLIC LABEL/NAME = LASTSCR
* DESCRIPTION = DSN8001/DSN8002
*
* EXIT-NORMAL = DSN8IC0
* EXIT-ERROR = DSN8IC0
* RETURN CODE = NONE
* ABEND CODES = NONE
* ERROR-MESSAGES = NONE
* EXTERNAL REFERENCES =
* ROUTINES/SERVICES =
* DSN8IC2
* DSN8MCG
* DSNTIAR
```
data-areas =
  dsn8mcca - cobol structure for commarea
  dsn8mcc2 - commarea part 2
  dsn8mccs - vcona table dcl & pcnota dclgen
  dsn8mcov - poptval table dcl & poptval dclgen
  dsn8mcmv - validation cursors
  dsn8mcxx - sql error handling module
  dsn8mc1 - sql1 common module for ims & cics
  dsn8mc3 - dsn8mc5 - validation modules called by dsn8mc1
control-blocks =
  sqlca - sql communication area
  tables = none
  change-activity = none
*pseudocode*
  procedure
    include declarations.
    include dsn8mc1.
    cc1exit: (referenced by dsn8mc1)
      return.
    cc1call: (referenced by dsn8mc1)
      call 'dsn8ic2' using commarea.
      go to mc1save. (label in dsn8mc1)
    include validation modules.
end.

environment division.
-----------------------------------
data division.
---------------------
working-storage section.
**********************************************************************
* * declare field sent to message routine*
* * declare conversation status*
* * declare message text*
* * declare option validation*
* * declare common area and common area part 2*
**********************************************************************
01 msgcode pic x(04).
01 outmsg pic x(69).
exec sql include dsn8mccs end-exec.
exec sql include dsn8mccs end-exec.
exec sql include sqlca end-exec.
exec sql include dsn8mcc2 end-exec.

linkage section.
01 commarea.
exec sql include dsn8mcca end-exec.

procedure division using commarea.
-------------------------------
* * sql error handling*
* execution whenever sqlerror go to db-error end-exec
exec sql when ever sqlwarning go to db-error end-exec.
* move 'dsn8ic1' to major in dsn8-module-name.
**********************************************************************
* find valid options for action, object, search criterion*
* retrieve conversation, validate, call sql2*
**********************************************************************
exec sql include dsn8mcov end-exec.
exec sql include dsn8mc1 end-exec.  **include sql1 main
* CC1-EXIT.
**RETURN
GOBACK.

**********************************************************************
* VALIDATE ACTION, OBJECT, SEARCH CRITERIA
* HANDLE ERRORS
**********************************************************************

CC1-CALL.
* CALL 'DSN8IC2' USING COMMAREA.
GO TO MC1-SAVE.

EXEC SQL INCLUDE DSN8MC3 END-EXEC.
EXEC SQL INCLUDE DSN8MC4 END-EXEC.
EXEC SQL INCLUDE DSN8MC5 END-EXEC.
EXEC SQL INCLUDE DSN8MCXX END-EXEC.
GOBACK.

Related reference
“Sample applications in IMS” on page 1346
A set of Db2 sample applications run in the IMS environment.

**DSN8IC2**

ROUTER FOR SECONDARY SELECTION AND/OR DETAIL PROCESSING CALLS SECONDARY SELECTION MODULES DSN8MCA DSN8MCM CALLS DETAIL MODULES DSN8MCD DSN8MCE DSN8MCF DSN8MCT DSN8MCV DSN8MCW DSN8MCX DSN8MCZ CALLED BY DSN8IC1 (SQL1).
* SUB PROCEDURES. ALL SQL CURSOR DEFINITIONS AND ALL SECONDARY SELECTION AND DETAIL MODULES ARE 'INCLUDED'.

AND DETAIL MODULES. BETWEEN SQL 0, SQL 1, SQL 2, AND THE SECONDARY SELECTION POINTS TO A COMMUNICATION CONTROL AREA USED TO COMMUNICATE MODULE. PASSED PARAMETERS CONSIST ONLY OF A POINTER WHICH FIELDS (MAJOR SYSTEM, ACTION, OBJECT, SEARCH CRITERIA). IF TO SQL 1. SQL 1 PERFORMS VALIDATION ON THE SYSTEM DEPENDENT INPUT SCREEN, FORMATTED THE INPUT, AND PASSED CONTROL TO SQL 1. SQL 1 PERFORMS VALIDATION ON THE SYSTEM DEPENDENT FIELDS (MAJOR SYSTEM, ACTION, OBJECT, SEARCH CRITERIA). IF ALL SYSTEM FIELDS ARE VALID, SQL 1 PASSED CONTROL TO THIS MODULE. PASSED PARAMETERS CONSIST ONLY OF A POINTER WHICH POINTS TO A COMMUNICATION CONTROL AREA USED TO COMMUNICATE BETWEEN SQL 0, SQL 1, SQL 2, AND THE SECONDARY SELECTION AND DETAIL MODULES.

WHAT HAS HAPPENED SO FAR?.............. THE SUBSYSTEM DATA-AREAS =

ROUTINES/SERVICES = 10 MODULES LISTED ABOVE

EXTERNAL REFERENCES =

ROUTINES/SERVICES = 10 MODULES LISTED ABOVE

ERROR-MESSAGES =

EXTERNAL REFERENCES =

ERROR-MESSAGES =

CONTROL-BLOCKS =

SQLCA - SQL COMMUNICATION AREA

OUTPUT = POINTER TO COMMAREA (COMMUNICATION AREA)

SYMBOLIC LABEL/NAME = COMMAREA

DESCRIPTION = COMMUNICATION AREA PASSED BETWEEN MODULES

EXIT-NORMAL =

EXIT-ERROR = IF SQLERROR OR SQLWARNING, SQL WHENEVER AND PROGRAM WILL GO TO THE LABEL DB-ERROR.

RETURN CODE = NONE

ABEND CODES = NONE

ERROR-MESSAGES =

EXTERNAL REFERENCES =

ROUTINES/SERVICES = 10 MODULES LISTED ABOVE

DATA-AREAS =

CONTROL-BLOCKS =

THIS MODULE DETERMINES WHICH SECONDARY SELECTION AND/OR DETAIL MODULE(S) ARE TO BE CALLED FOR THE IMS/COBOL ENVIRONMENT

WHAT HAS HAPPENED SO FAR?.............. THE SUBSYSTEM DEPENDENT MODULE (IMS.CICS) (SQL 0) HAS READ THE INPUT SCREEN, FORMATTED THE INPUT, AND PASSED CONTROL TO SQL 1. SQL 1 PERFORMS VALIDATION ON THE SYSTEM DEPENDENT FIELDS (MAJOR SYSTEM, ACTION, OBJECT, SEARCH CRITERIA). IF ALL SYSTEM FIELDS ARE VALID, SQL 1 PASSED CONTROL TO THIS MODULE. PASSED PARAMETERS CONSIST ONLY OF A POINTER WHICH POINTS TO A COMMUNICATION CONTROL AREA USED TO COMMUNICATE BETWEEN SQL 0, SQL 1, SQL 2, AND THE SECONDARY SELECTION AND DETAIL MODULES.

WHAT IS INCLUDED IN THIS MODULE?..............

ALL SECONDARY SELECTION AND DETAIL MODULES ARE 'INCLUDED'.

ALL VARIABLES KNOWN IN THIS PROCEDURE ARE KNOWN IN THE SUB PROCEDURES. ALL SQL CURSOR DEFINITIONS AND SQL 'INCLUDES' ARE DONE IN THIS PROCEDURE. ALL CURSOR HOST
Chapter 12. Sample data and applications supplied with Db2 for z/OS  1355
EXEC SQL INCLUDE DSNMCC2 END-EXEC.
EXEC SQL INCLUDE DSNMCDP END-EXEC.
EXEC SQL INCLUDE DSNMCEM END-EXEC.
EXEC SQL INCLUDE DSNMCDM END-EXEC.
EXEC SQL INCLUDE DSNMCAD END-EXEC.
EXEC SQL INCLUDE DSNMCA2 END-EXEC.
EXEC SQL INCLUDE DSNMCOV END-EXEC.
EXEC SQL INCLUDE DSNMCDT END-EXEC.
EXEC SQL INCLUDE DSNMCED END-EXEC.

01 CONSTRAINTS.
  03 PARM-LENGTH PIC S9(4) COMP-4.
  03 REF-CONSTRAINT PIC X(08).
  03 FILLER PIC X(62).
  01 MGRNO-CONSTRAINT PIC X(08) VALUE 'RDE '.

LINKAGE SECTION.
01 COMMAREA.
  EXEC SQL INCLUDE DSNMCCA END-EXEC.

PROCEDURE DIVISION USING COMMAREA.

*------------------
*******************************************************
*   SQL ERROR CODE HANDLING                                *
*******************************************************
EXEC SQL WHENEVER SQLERROR   GO TO DB-ERROR END-EXEC
EXEC SQL WHENEVER SQLWARNING GO TO DB-ERROR END-EXEC.

EXEC SQL INCLUDE DSNMCAE END-EXEC.
EXEC SQL INCLUDE DSNMCAL END-EXEC.
EXEC SQL INCLUDE DSNMCDH END-EXEC.
EXEC SQL INCLUDE DSNMCDI END-EXEC.
EXEC SQL INCLUDE DSNMCDI END-EXEC.

*******************************************************
*   INITIALIZATIONS                                        *
*******************************************************
MOVE 'DSN8IC2' TO MAJOR.
MOVE SPACES TO MINOR.

IF NEWREQ OF COMPARM = 'Y' THEN GO TO IC2008.

*******************************************************
*   DETERMINES WHETHER NEW REQUEST OR NOT                  *
*******************************************************
IC2005.
IF PREV OF PCONVSTA = ' ' THEN
  MOVE 'Y' TO NEWREQ OF COMPARM.
IF NEWREQ OF COMPARM = 'N' AND PREV OF PCONVSTA = 'S'
  AND DATA01 NOT = ' ' AND DATAIN NOT = 'NEXT'
  THEN MOVE 'Y' TO NEWREQ OF COMPARM.
IF NEWREQ OF COMPARM NOT = 'Y' THEN GO TO IC2010.

*******************************************************
*   DETERMINE IF VALID SELECTION NUMBER GIVEN               *
*******************************************************
IC2010.
IF PREV OF PCONVSTA NOT = 'S'
  OR MAXSEL < 1
  OR DATAIN = 'NEXT'
  OR DAT2 = DATO2 THEN GO TO IC201.
IF DAT1 NUMERIC AND DAT2 = ' ' THEN
MOVE DAT1 TO DAT2
MOVE '0' TO DAT1.

* **DETAIL SELECTION GIVEN
  IF DATA2 NUMERIC
    AND DATA2 > '00' AND DATA2 NOT > MAXSEL THEN
    MOVE 'Y' TO NEWREQ OF COMPARM
    GO TO DETAIL0.
  END-IF.

* **INVALID SELECTION NO.
* **PRINT ERROR MESSAGE
  MOVE '072E' TO MSGCODE.
  CALL 'DSN8MCG' USING MAJOR MSGCODE OUTMSG.
  MOVE OUTMSG TO MSG OF OUTAREA.
  GO TO EXIT0.

*******************************************************************************
* DETERMINES WHETHER SECONDARY SELECTION OR DETAIL
*******************************************************************************

**SECONDARY SELECTION
  IF PREV OF PCONVSTA = 'S' THEN
    PERFORM SECSEL THRU END-SECSEL
    GO TO EXIT0.
  END-IF.

**DETAIL PROCESSOR
  IF PREV OF PCONVSTA = 'D' THEN GO TO DETAIL0.

**LOGIC ERROR
* **PRINT ERROR MESSAGE
  MOVE '066E' TO MSGCODE.
  CALL 'DSN8MCG' USING MAJOR MSGCODE OUTMSG.
  MOVE OUTMSG TO MSG OF OUTAREA.
  GO TO EXIT0.

*******************************************************************************
* DETERMINES WHETHER SECONDARY SELECTION OR DETAIL
*******************************************************************************

**ADMINISTRATIVE
  IF OBJFLD OF INAREA = 'DS' THEN
    PERFORM DSN8MCA THRU END-DSN8MCA
    ELSE
      IF OBJFLD OF INAREA = 'DE' THEN
        PERFORM DSN8MCA THRU END-DSN8MCA
        ELSE
          IF OBJFLD OF INAREA = 'EM' THEN
            PERFORM DSN8MCA THRU END-DSN8MCA
            ELSE
              MOVE DAT1 TO DAT2
              CALL 'DSN8MC' USING MAJOR MSGCODE OUTMSG.
              MOVE OUTMSG TO MSG OF OUTAREA.
              GO TO EXIT0.
            END-IF.
          END-ELSE.
        END-IF.
      END-ELSE.
    END-ELSE.
  END-IF.

**INDIVIDUAL DEPARTMENT
  **PROCESSING
    IF OBJFLD OF INAREA = 'DE' THEN
      PERFORM DSN8MCA THRU END-DSN8MCA
      ELSE
        IF OBJFLD OF INAREA = 'EM' THEN
          PERFORM DSN8MCA THRU END-DSN8MCA
          ELSE
            MOVE '062E' TO MSGCODE.
            CALL 'DSN8MCG' USING MAJOR MSGCODE OUTMSG.
            MOVE OUTMSG TO MSG OF OUTAREA.
            GO TO EXIT0.
          END-ELSE.
        END-IF.
      END-ELSE.
    END-ELSE.
  END-ELSE.

**INDIVIDUAL EMPLOYEE
  **PROCESSING
    IF OBJFLD OF INAREA = 'EM' THEN
      PERFORM DSN8MCA THRU END-DSN8MCA
      ELSE
        IF OBJFLD OF INAREA = 'DE' THEN
          PERFORM DSN8MCA THRU END-DSN8MCA
          ELSE
            PERFORM DSN8MCA THRU END-DSN8MCA
          END-ELSE.
        END-IF.
      END-ELSE.
    END-ELSE.
  END-ELSE.

**UNSELECTED FOR OBJECT
**PRINT ERROR MESSAGE
  MOVE '072E' TO MSGCODE.
  CALL 'DSN8MCG' USING MAJOR MSGCODE OUTMSG.
  MOVE OUTMSG TO MSG OF OUTAREA.
  GO TO EXIT0.

*******************************************************************************
* DETERMINES WHETHER SECONDARY SELECTION OR DETAIL
*******************************************************************************

**ADMINISTRATIVE
  IF OBJFLD OF INAREA = 'DS' THEN
    PERFORM DSN8MCA THRU END-DSN8MCA
    ELSE
      IF OBJFLD OF INAREA = 'DE' THEN
        PERFORM DSN8MCA THRU END-DSN8MCA
        ELSE
          IF OBJFLD OF INAREA = 'EM' THEN
            PERFORM DSN8MCA THRU END-DSN8MCA
            ELSE
              MOVE DAT1 TO DAT2
              CALL 'DSN8MC' USING MAJOR MSGCODE OUTMSG.
              MOVE OUTMSG TO MSG OF OUTAREA.
              GO TO EXIT0.
            END-ELSE.
          END-IF.
        END-ELSE.
      END-ELSE.
    END-ELSE.
  END-IF.

**INDIVIDUAL DEPARTMENT
  **PROCESSING
    IF OBJFLD OF INAREA = 'DE' THEN
      PERFORM DSN8MCA THRU END-DSN8MCA
      ELSE
        IF OBJFLD OF INAREA = 'EM' THEN
          PERFORM DSN8MCA THRU END-DSN8MCA
          ELSE
            MOVE DAT1 TO DAT2
            CALL 'DSN8MC' USING MAJOR MSGCODE OUTMSG.
            MOVE OUTMSG TO MSG OF OUTAREA.
            GO TO EXIT0.
          END-ELSE.
        END-IF.
      END-ELSE.
    END-ELSE.
  END-ELSE.

**INDIVIDUAL EMPLOYEE
  **PROCESSING
    IF OBJFLD OF INAREA = 'EM' THEN
      PERFORM DSN8MCA THRU END-DSN8MCA
      ELSE
        IF OBJFLD OF INAREA = 'DE' THEN
          PERFORM DSN8MCA THRU END-DSN8MCA
          ELSE
            PERFORM DSN8MCA THRU END-DSN8MCA
          END-ELSE.
        END-IF.
      END-ELSE.
    END-ELSE.
  END-ELSE.
THIS MODULE RECEIVES INPUT MESSAGE AND DEFORMATS IT, CALLS DSN8IP1, FORMATS OUTPUT MESSAGE AND SENDS IT.

**DSN8IP0**

**Related reference**

“Sample applications in IMS” on page 1346

A set of Db2 sample applications run in the IMS environment.
Chapter 12. Sample data and applications supplied with Db2 for z/OS 1359
OUTPUTAREA.MSG = OUTMSG;
EXITCODE = '0';
OUTAREA.MAJSYS = 'O';
DSN8_MODULE_NAME.MAJOR = 'DSN8IP0';
COMMAREA = '';
Related reference

“Sample applications in IMS” on page 1346
A set of Db2 sample applications run in the IMS environment.

DSN8IP1

PERFORM INCLUDES TO BRING IN SQL TABLE DCLS AND DCLGEN STRUCTURES AS WELL AS PARAMETER AREA.

DSN8IP1:PROC (COMMPTR);
/*****************************/
/* MODULE NAME = DSN8IP1 */
/* DESCRIPTIVE NAME = SAMPLE APPLICATION */
/* SQL 1 MAINLINE */
/* IMS */
/* PL/I */
/* COPYRIGHT = 5740-XYR (C) COPYRIGHT IBM CORP 1982, 1985 */
/* REFER TO COPYRIGHT INSTRUCTIONS FORM NUMBER G120-2083 */
/* STATUS = RELEASE 2, LEVEL 0 */
/* FUNCTION = PERFORM INCLUDES TO BRING IN SQL TABLE DCLS AND */
/* DCLGEN STRUCTURES AS WELL AS PARAMETER AREA. */
/* INCLUDE DSN8MP1. */
/* CALL DSN8IP2. */
*      INCLUDE VALIDATION MODULES.                                  *
*          GO TO MP1SAVE.  (LABEL IN DSN8MP1)                        *
*          CALL 'DSN8IP2'(COMMPTR).                                  *
*      CP1CALL:  ( REFERENCED BY DSN8MP1 )                           *
*                                                                    *
*          RETURN.                                                   *
*                                                                    *
*      INCLUDE ERROR HANDLER.                                        *
*      INCLUDE DSN8MP1.                                              *
*      INCLUDE DECLARATIONS.                                         *
*    PROCEDURE                                                       *
*                                                                    *
*   CHANGE-ACTIVITY = NONE                                           *
*                                                                    *
*   TABLES = NONE                                                    *
*                                                                    *
*   SQLCA               - SQL COMMUNICATION AREA               *
*   COMMON AREA.                                                   *
*   COMMON AREA.                                                   *
*   SYMBOLIC LABEL/NAME = PFKIN                                     *
*   DESCRIPTION = 00/01/02/03/08/10                                 *
*   SYMBOLIC LABEL/NAME = INAREA                                    *
*   DESCRIPTION = USER INPUT                                       *
*                                                                    *
*   OUTPUT = PARAMETERS EXPLICITLY RETURNED:                        *
*   COMMON AREA.                                                   *
*   COMMON AREA.                                                   *
*   SYMBOLIC LABEL/NAME = OUTAREA                                   *
*   DESCRIPTION = GENERAL MENU OR SECONDARY                         *
*                                                                        *
*   SYMBOLIC LABEL/NAME = LASTSCR                                   *
*   DESCRIPTION = DSN8001/DSN8002                                   *
*                                                                    *
*   EXIT-NORMAL = DSN8IP0                                           *
*   EXIT-ERROR = DSN8IP0                                            *
*   RETURN CODE = NONE                                              *
*   ABEND CODES = NONE                                              *
*   ERROR-MESSAGES = NONE                                           *
*   EXTERNAL REFERENCES =                                           *
*   ROUTINES/SERVICES =     NONE                                    *
*   DATA-AREAS =                                                    *
*   PLI STRUCTURE FOR COMMAREA                                      *
*   VCONA TABLE DCL AND PCONA DCLGEN                                *
*   VOPTVAL TABLE DCL & POPTVAL DCLGEN                              *
*   SQL1 COMMON MODULE FOR IMS AND CICS                             *
*   VALIDATION MODULES CALLED BY DSN8MP1                            *
*   SQL ERROR HANDLER                                              *
*   SQL COMMUNICATION AREA                                          *
*   SQLCA.                                                         *
*   SQL COMMUNICATION AREA                                          *
*   SQLCA.                                                         *
*   SQL COMMUNICATION AREA                                          *
*   PSEUDO CODE                                                     *
*   PROCEDURE                                                       *
*   INCLUDE DECLARATIONS.                                          *
*   INCLUDE DSN8MP1.                                                *
*   INCLUDE ERROR HANDLER.                                         *
*   CP1EXIT:  ( REFERENCED BY DSN8MP1 )                             *
*   RETURN.                                                        *
*   CP1CALL:  ( REFERENCED BY DSN8MP1 )                             *
*   CALL 'DSN8IP2'(COMMPTR).                                       *
*   GO TO MP1SAVE.  (LABEL IN DSN8MP1)                             *
*   INCLUDE VALIDATION MODULES.                                    *
DCL STRING BUILTIN;
DCL J FIXED BIN;
DCL SAVE_CONVID CHAR(16);
DCL (SENDBIT, ENDBIT, NEXTBIT, ON, OFF) BIT(1);
/* SQL RETURN CODE HANDLING */
EXEC SQL WHENEVER SQLERROR GO TO DB_ERROR;
EXEC SQL WHENEVER SQLWARNING GO TO DB_ERROR;

/*/ FIELDS PASSED TO MESSAGE ROUTINE */
DCL MODULE       CHAR(07);
DCL OUTMSG       CHAR(69);
DCL DSN8IP2 EXTERNAL ENTRY;
DCL DSN8MPG EXTERNAL ENTRY;
EXEC SQL INCLUDE DSN8MPCA;                 /* INCLUDE COMMAREA */
DSN8_MODULE_NAME.MAJOR = 'DSN8IP1';       /* INITIALIZE MODULE NAME*/
EXEC SQL INCLUDE DSN8MPCS;                 /* INCLUDE PCONA */
EXEC SQL INCLUDE DSN8MPOV;                 /* INCLUDE CURSOR */
EXEC SQL INCLUDE DSN8SQLCA;                /* SQL COMMON AREA */
EXEC SQL INCLUDE DSN8MP1;                  /* INCLUDE SQL1 MAIN*/
EXEC SQL INCLUDE DSN8MPXX;                 /* HANDLES ERRORS */
CP1EXIT :                                  /* STANDARD EXIT */
RETURN;
CP1CALL :
    CALL DSN8IP2 (COMMPTR);
    GO TO MP1SAVE;
EXEC SQL INCLUDE DSN8MP3;              /* INCLUDE ACTION VALIDATION*/
EXEC SQL INCLUDE DSN8MP4;              /* INCLUDE OBJECT VALIDATION*/
EXEC SQL INCLUDE DSN8MP5;              /* INCLUDE SEARCH CRITERIA*/
END;                                   /* VALIDATION */

Related reference
“Sample applications in IMS” on page 1346
A set of Db2 sample applications run in the IMS environment.

DSN8IP2
ROUTER FOR SECONDARY SELECTION AND/OR DETAIL PROCESSIN CALLS SECONDARY SELECTION MODULES DSN8MPA CALLS DETAIL MODULES DSN8MPD DSN8MPE DSN8MPF CALLED BY DSN8IP1 (SQL1).
Db2 12 for z/OS: Application Programming and SQL Guide

* NOTES = NONE

* MODULE TYPE = BLOCK OF PL/I CODE

* PROCESSOR = DB2 PRECOMPILER, PL/I OPTIMIZER

* MODULE SIZE = SEE LINKEDIT

* ATTRIBUTES = REUSABLE

* ENTRY POINT = DSN8IP2

* PURPOSE = SEE FUNCTION

* LINKAGE = CALL DSN8IP2(COMMPTR)

* INPUT =

  SYMBOLIC LABEL/NAME = COMMPTR
  DESCRIPTION = POINTER TO COMMUNICATION AREA

* OUTPUT =

  SYMBOLIC LABEL/NAME = COMMPTR
  DESCRIPTION = POINTER TO COMMUNICATION AREA

* EXIT-NORMAL =

* ERROR-MESSAGES =

  DSN8062E-INVALID SELECTION ON SECONDARY SCREEN
  DSN8066E-UNSUPPORTED PFK OR LOGIC ERROR
  DSN8072E-INVALID SELECTION ON SECONDARY SCREEN

* DATA-AREAS =

  DSN8MPA - SECONDARY SELECTION FOR ORGANIZATION
  DSN8MPAD - DECLARE ADMINISTRATIVE DETAIL
  DSN8MPAE - CURSOR EMPLOYEE LIST
  DSN8MPAL - CURSOR ADMINISTRATION LIST
  DSN8MPA2 - DECLARE ADMINISTRATIVE DETAIL
  DSN8MPCA - DECLARE SQL COMMON AREA
  DSN8MPD - DEPARTMENT STRUCTURE DETAIL
  DSN8MPDA - CURSOR ADMINISTRATION DETAIL
  DSN8MPDH - CURSOR FOR DISPLAY TEXT FROM
  DSN8MPDM - DECLARE DEPARTMENT MANAGER
  DSN8MPDP - DECLARE DEPARTMENT
  DSN8MPDT - DECLARE DISPLAY TEXT
  DSN8MPF - DECLARE EMPLOYEE DETAIL
  DSN8MPED - DECLARE EMPLOYEE-DEPARTMENT
  DSN8MPAL - DECLARE EMPLOYEE DETAIL

* CONTROL-BLOCKS =

  SQLCA - SQL COMMUNICATION AREA

* TABLES = NONE

* CHANGE-ACTIVITY = NONE

* PSEUDOCODE*

* THIS MODULE DETERMINES WHICH SECONDARY SELECTION AND/OR DETAIL MODULE(S) ARE TO BE CALLED FOR THE IMS/PL1 ENVIRONMENT.

* NOTES =

  WHAT HAS HAPPENED SO FAR? 
  THE SUBSYSTEM
  DEPENDENT MODULE (IMS,CICS) (SQL 0) HAS READ THE INPUT SCREEN, FORMATTED THE INPUT, AND PASSED CONTROL TO SQL 1. SQL 1 PERFORMS VALIDATION ON THE SYSTEM DEPENDENT FIELDS (MAJOR SYSTEM, ACTION, OBJECT, SEARCH CRITERIA). IF ALL SYSTEM FIELDS ARE VALID, SQL 1 PASSED CONTROL TO THIS
* MODULE. PASSED PARAMETERS CONSIST ONLY OF A POINTER WHICH * 01060000
  * POINTS TO A COMMUNICATION CONTROL AREA USED TO COMMUNICATE * 01070000
  * BETWEEN SQL 0, SQL 1, SQL 2, AND THE SECONDARY SELECTION * 01080000
  * AND DETAIL MODULES. * 01090000
  * * 01100000 01110000
  * WHAT IS INCLUDED IN THIS MODULE?.............. * 01120000
  * ALL SECONDARY SELECTION AND DETAIL MODULES ARE 'INCLUDED'. * 01130000
  * ALL VARIABLES KNOWN IN THIS PROCEDURE ARE KNOWN IN THE * 01140000
  * SUB PROCEDURES. ALL SQL CURSOR DEFINITIONS AND * 01150000
  * SQL 'INCLUDES' ARE DONE IN THIS PROCEDURE. ALL CURSOR HOST * 01160000
  * VARIABLES ARE DECLARED IN THIS PROCEDURE BECAUSE OF THE * 01170000
  * RESTRICTION THAT CURSOR HOST VARIABLES MUST BE DECLARED BEFORE* 01180000
  * THE CURSOR DEFINITION. * 01190000
  * * 01200000 01210000
  * PROCEDURE * 01220000
  * * 01230000 01240000
  * IF ANSWER TO DETAIL SCREEN & DETAIL PROCESSOR * 01250000
  * IS NOT WILLING TO ACCEPT AN ANSWER THEN * 01260000
  * NEW REQUEST* * 01270000
  * ELSE * 01280000
  * IF ANSWER TO A SECONDARY SELECTION THEN * 01290000
  * DETERMINE IF NEW REQUEST. * 01300000
  * CASE (NEW REQUEST) * 01310000
  * SUBCASE ('ACTION') * 01320000
  * DETAIL PROCESSOR * 01330000
  * RETURN TO SQL 1 * 01340000
  * ENDSUB * 01350000
  * SUBCASE ('DISPLAY','ERASE','UPDATE') * 01360000
  * CALL SECONDARY SELECTION * 01370000
  * IF # OF POSSIBLE CHOICES IS ^= 1 THEN * 01380000
  * RETURN TO SQL 1 * 01390000
  * ELSE * 01400000
  * CALL THE DETAIL PROCESSOR * 01410000
  * RETURN TO SQL 1. * 01420000
  * ENDSUB * 01430000
  * * 01440000 01450000
  * ENDCASE * 01460000
  * IF ANSWER TO SECONDARY SELECTION AND A SELECTION HAS * 01470000
  * ACTUALLY BEEN MADE THEN * 01480000
  * VALID SELECTION #? * 01490000
  * IF IT IS VALID THEN * 01500000
  * CALL DETAIL PROCESSOR * 01510000
  * RETURN TO SQL 1 * 01520000
  * ELSE * 01530000
  * PRINT ERROR MSG * 01540000
  * RETURN TO SQL 1. * 01550000
  * IF ANSWER TO SECONDARY SELECTION THEN * 01560000
  * CALL SECONDARY SELECTION * 01570000
  * RETURN TO SQL 1. * 01580000
  * IF ANSWER TO DETAIL THEN * 01590000
  * CALL DETAIL PROCESSOR * 01600000
  * RETURN TO SQL 1. * 01610000
  * END. * 01620000
  * * 01630000 01640000
  * EXAMPLE- A ROW IS SUCCESSFULLY ADDED, THE OPERATOR RECEIVES * 01650000
  * THE SUCCESSFULLY ADDED MESSAGE AND JUST HITS ENTER. * 01660000
  * DCL DSNMPG EXTERNAL ENTRY; * 01670000
  * DCL LENGTH BUILTIN; * 01680000
  * DCL LENGTH SQLA; * 01690000
  * /* INCLUDE DECLARES */ * 01700000
  * EXEC SQL INCLUDE DSNMPPCA; */COMMUNICATION AREA BETWEEN MODULES */ 01710000
  * EXEC SQL INCLUDE SQLA; */SQL COMMUNICATION AREA */ 01720000
  * /* ORGANIZATION */ 01730000
  * EXEC SQL INCLUDE DSNMPPOP; /* DCLGEN FOR DEPARTMENT */ 01740000
  * EXEC SQL INCLUDE DSNMPPEM; /* DCLGEN FOR EMPLOYEE */ 01750000
  * EXEC SQL INCLUDE DSNMPMPE; /* DCLGEN FOR EMPLOYEE-DEPARTMENT */ 01760000
  * EXEC SQL INCLUDE DSNMPMPO; /* DCLGEN FOR DEPARTMENT/DEPARTMENT */ 01770000
  * EXEC SQL INCLUDE DSNMPMD; /* DCLGEN FOR ADMINISTRATION DETAIL */ 01780000
  * EXEC SQL INCLUDE DSNMPMAD; /* DCLGEN FOR ADMINISTRATION DETAIL */ 01790000
  * EXEC SQL INCLUDE DSNMPMDT; /* DCLGEN FOR DISPLAY TEXT TABLE */ 01800000
  * /* CURSORS */
EXEC SQL INCLUDE DSN8MPAL; /* MAJSYS O - SEC SEL FOR DS AND DE */ 01879000
EXEC SQL INCLUDE DSN8MPAE; /* MAJSYS O - SEC SEL FOR EM */ 01880000
EXEC SQL INCLUDE DSN8MPDA; /* MAJSYS O - DETAIL FOR DS */ 01890000
EXEC SQL INCLUDE DSN8MPDH; /* PROG TABLES - DISPLAY HEADINGS */ 01900000

EXEC SQL INCLUDE DSN8MPAE; /* MAJSYS O - SEC SEL FOR EM */ 01880000
EXEC SQL INCLUDE DSN8MPDA; /* MAJSYS O - DETAIL FOR DS */ 01890000
EXEC SQL INCLUDE DSN8MPDH; /* PROG TABLES - DISPLAY HEADINGS */ 01900000

DCL MODULE CHAR (07) INIT ('DSN8IP2');
DCL OUTMSG CHAR (69);

/*********************************************************/
/*          **  FIELDS SENT TO MESSAGE ROUTINE              */
/*********************************************************/

DCL UNSPEC BUILTIN;
DCL VERIFY BUILTIN;

/*********************************************************/
/*  DETERMINES WHETHER NEW REQUEST OR NOT                  */
/*********************************************************/

IF PCONVSTA.PREV = ' ' THEN
   COMPARM.NEWREQ = 'Y';
END;

IF COMPARM.NEWREQ = 'N' & PCONVSTA.PREV = 'S' &
   SUBSTR(COMPARM.DATA,3,58) ^= ' ' &
   COMPARM.DATA ^= 'NEXT'
   THEN COMPARM.NEWREQ = 'Y';

/*********************************************************/
/*  IF NEW REQUEST AND ACTION IS 'ADD' THEN                */
/*       CALL DETAIL PROCESSOR                             */
/*  ELSE CALL SECONDARY SELECTION                          */
/*********************************************************/

IF COMPARM.NEWREQ='Y' THEN
   DO;
      IF COMPARM.ACTION = 'A' THEN
         CALL DETAIL; /* CALL DETAIL PROCESSOR */
         GO TO EXIT; /* RETURN */
      END;
   END;
   CALL SECSEL; /* CALL SECONDARY SELECTION */
   IF MAXSEL = 1 THEN /* IF NO. OF CHOICES = 1 */
      CALL DETAIL; /* CALL DETAIL PROCESSOR */
      GO TO EXIT; /* RETURN */
   END;
   CALL ECSEL; /* CALL SECONDARY SELECTION */
   IF MAXSEL < 1 THEN /* NO VALID CHOICES */
      IF PCONVSTA.MAXSEL <> 3 THEN GO TO IP201; /* TO SECONDARY SEL */
      IF PCONVSTA.PREV ^= 'S' THEN GO TO IP201; /* TO SECONDARY SEL */
      IF PCONVSTA.MAXSEL < 1 THEN GO TO IP201; /* NO VALID CHOICES */
      IF PCONVSTA.MAXSEL = 1 THEN /* IF NO. OF CHOICES = 1 */
         CALL DETAIL; /* CALL DETAIL PROCESSOR */
         GO TO EXIT; /* RETURN */
      END;
      CALL SECSEL; /* CALL SECONDARY SELECTION */
      IF MAXSEL = 1 THEN /* IF NO. OF CHOICES = 1 */
         CALL DETAIL; /* CALL DETAIL PROCESSOR */
         GO TO EXIT; /* RETURN */
      END;
      CALL ECSEL; /* CALL SECONDARY SELECTION */
      IF MAXSEL < 1 THEN /* NO VALID CHOICES */
         IF PCONVSTA.MAXSEL <> 3 THEN GO TO IP201; /* TO SECONDARY SEL */
         IF PCONVSTA.PREV ^= 'S' THEN GO TO IP201; /* TO SECONDARY SEL */
         IF PCONVSTA.MAXSEL < 1 THEN GO TO IP201; /* NO VALID CHOICES */
IF COMPARM.DATA = 'NEXT' THEN GO TO IP201; /* SCROLL REQUEST*/

IF SUBSTR(COMPARM.DATA,1,2) = SUBSTR(PCONVSTA.DATA,1,2)
THEN GO TO IP201; /* NO CHANGE ON INPUT SCREEN */

IF SUBSTR(COMPARM.DATA,2,1) = ' ' THEN /* SECOND CHAR BLANK */

IF VERIFY(SUBSTR(COMPARM.DATA,1,1),'123456789') = 0 THEN DO;

SUBSTR(COMPARM.DATA,2,1) = SUBSTR(COMPARM.DATA,1,1);
SUBSTR(COMPARM.DATA,1,1) = '0';
END;

IF VERIFY(SUBSTR(COMPARM.DATA,1,2),'0123456789') = 0 &
SUBSTR(COMPARM.DATA,1,2) > '00' THEN

IF DATAP <= PCONVSTA.MAXSEL THEN DO;

COMPARM.NEWREQ = 'Y'; /* TELL DETAIL PROCESSOR NEW REQ */
CALL DETAIL;          /* CALL DETAIL PROCESSOR         */
GO TO EXIT;           /* RETURN                        */
END;

/* INVALID SELECTION NO.       */
/* PRINT ERROR MESSAGE         */
CALL DSN8MPG (MODULE, '066E', OUTMSG);
PCONVSTA.MSG = OUTMSG;
GO TO EXIT;                      /* RETURN                      */

IF PCONVSTA.PREV = 'S' THEN
DO;
CALL SECSEL;                 /* CALL SECONDARY SELECTION   */
GO TO EXIT;                  /* RETURN                     */
END;

IF PCONVSTA.PREV = 'D' THEN
DO;
CALL DETAIL;                 /* CALL DETAIL PROCESSOR      */
GO TO EXIT;                  /* RETURN                     */
END;

/* LOGIC ERROR        */
CALL DSN8MPG (MODULE, '062E', OUTMSG);  /* PRINT ERROR MESSAGE*/
PCONVSTA.MSG = OUTMSG;
GO TO EXIT;

EXEC SQL INCLUDE DSN8MPXX;      /* HANDLES SQL ERRORS         */
GO TO EXIT;                     /* RETURN                     */

SECSEL: PROC; /* CALL APPROPRIATE SECONDARY SELECTION MODULE */

PCONVSTA.LASTSCR = 'DSN8001'; /* NOTE GENERAL SCREEN */
IF COMPARM.OBJFLD='DS' |           /* DEPARTMENT STRUCTURE*/
COMPARM.OBJFLD='DE' |           /* INDIVIDUAL DEPARTMENT*/
COMPARM.OBJFLD='EM' THEN        /* INDIVIDUAL EMPLOYEE  */
DO;

CALL DSN8MPA;                                                RETURN; 03180000
END;

/*MISSING SECONDARY SEL*/
CALL DSN8MPG (MODULE, '066E', OUTMSG);  /* PRINT ERROR MESSAGE*/
PCONVSTA.MSG = OUTMSG;
GO TO EXIT;

EXEC SQL INCLUDE DSN8MPXX;      /* HANDLES SQL ERRORS         */
GO TO EXIT;                     /* RETURN                     */

DETAIL: PROC; /* CALL APPROPRIATE DETAIL MODULE */

PCONVSTA.LASTSCR = 'DSN8002'; /* NOTE DETAIL SCREEN */
IF COMPARM.OBJFLD='DS' THEN        /* ADMINISTRATIVE */
DO;
/* DEPARTMENT STRUCTURE */

Chapter 12. Sample data and applications supplied with Db2 for z/OS 1367
Related reference

“Sample applications in IMS” on page 1346
A set of Db2 sample applications run in the IMS environment.

DSN8IP6

THIS MODULE RECEIVES INPUT MESSAGE AND DEFORMATS IT, CALLS DSN8IP7, FORMATS OUTPUT MESSAGE AND SENDS IT.

DSN8IP6: PROC(IOPCB_ADDR,ALTPCB_ADDR) OPTIONS (MAIN);

/*********************************************************************
*                                                                    *
*   MODULE NAME = DSN8IP6                                            *
*   DESCRIPTIVE NAME = DB2 SAMPLE APPLICATION                        *
*                      SUBSYSTEM INTERFACE MODULE                    *
*                      IMS                                           *
*                      PL/I                                          *
*                      PROJECT                                       *
*                                                                    *
*    COPYRIGHT = 5740-XYR (C) COPYRIGHT IBM CORP 1982, 1985          *
*    REFER TO COPYRIGHT INSTRUCTIONS FORM NUMBER G120-2083           *
*                                                                    *
*    STATUS = RELEASE 2, LEVEL 0                                     *
*                                                                    *
*   FUNCTION = THIS MODULE RECEIVES INPUT MESSAGE AND DEFORMATS IT,  *
*              CALLS DSN8IP7, FORMATS OUTPUT MESSAGE AND SENDS IT.   *
*                                                                    *
*   NOTES =  NONE                                                    *
*                                                                    *
*   MODULE TYPE = PL/I PROC OPTIONS(MAIN)                            *
*      PROCESSOR   = PL/I OPTIMIZER                                  *
*      MODULE SIZE = SEE LINKEDIT                                    *
*      ATTRIBUTES  = REUSABLE                                        *
*                                                                    *
*   ENTRY POINT =  DSN8IP6                                           *
*      PURPOSE = SEE FUNCTION                                        *
*      LINKAGE = FROM IMS                                            *
*                                                                    *
*   INPUT = PARAMETERS EXPLICITLY PASSED TO THIS FUNCTION:          *
*      COMMON AREA:                                                   *
*                                                                    *
*      SYMBOLIC LABEL/NAME = COMPARM.PFKIN                           *
*      DESCRIPTION = 00/01/02/03/08/10                               *
*      SYMBOLIC LABEL/NAME = COMPARM.INAREA                           *
*      DESCRIPTION = USER INPUT                                      *
*                                                                    *
*   OPTIONS = MAIN                                                  *
*                                                                    *
*   ORGANIZATION */                                                   *
*                                                                    *
EXEC SQL INCLUDE DSN8MPA; /* SEC SEL - ADMIN STRUCTURE */
EXEC SQL INCLUDE DSN8MPD; /* DETAIL  - ADMIN STRUCTURE */
EXEC SQL INCLUDE DSN8MPE; /* DETAIL  - DEPARTMENTS */
EXEC SQL INCLUDE DSN8MPF; /* DETAIL  - EMPLOYEES   */
END; /*       DSN8IP2          */
* INPUT-MESSAGE: *
  * SYMLOBIC LABEL/NAME = DSN8IPFI
  * DESCRIPTION = GENERAL MENU
  * SYMLOBIC LABEL/NAME = DSN8IPEI
  * DESCRIPTION = SECONDARY SELECTION MENU
  * OUTPUT = PARAMETERS EXPLICITLY RETURNED:
    * COMMON AREA:
      * SYMLOBIC LABEL/NAME = COMPARM.OUTAREA
        * DESCRIPTION = USER OUTPUT
      * SYMLOBIC LABEL/NAME = COMPARM_LASTSCR
        * DESCRIPTION = DSN8001/DSN8002
    * OUTPUT-MESSAGE:
      * SYMLOBIC LABEL/NAME = DSN8IPFO
        * DESCRIPTION = GENERAL MENU
      * SYMLOBIC LABEL/NAME = DSN8IPEO
        * DESCRIPTION = SECONDARY SELECTION MENU
      * EXIT-NORMAL =
      * EXIT-ERROR =
      * RETURN CODE = NONE
      * ABEND CODES = NONE
      * ERROR-MESSAGES =
        * DSN8064E - INVALID DL/I STC-CODE ON GU MSG
        * DSN8065E - INVALID DL/I STC-CODE ON ISRT MSG
      * EXTERNAL REFERENCES =
        * ROUTINES/SERVICES = MODULE DSN8IP7
          * MODULE PLITDLI
          * MODULE DSN8MPG
      * DATA-AREAS =
        * DSN8MPCA - PARAMETER TO BE PASSED TO DSN8CP7
          * CONTAINS TERMINAL INPUT AND OUTPUT AREAS.
      * IN_MESSAGE - MFS INPUT
      * OUT_MESSAGE - MFS OUTPUT
      * CONTROL-BLOCKS = NONE
      * TABLES = NONE
      * CHANGE-ACTIVITY = NONE
      * PROCEDURE
      * DECLARATIONS.
        * ALLOCATE PL/I WORK AREA FOR COMMAREA.
        * INITIALIZATION.
        * PUT MODULE NAME 'DSN8IP6' IN AREA USED BY ERROR-HANDLER.
        * PUT MODNAME 'DSN8IPFO' IN MODNAME FIELD.
      * STEP1.
        * CALL DLI GU INPUT MESSAGE.
        * IF STATUS CODE NOT OK THEN SEND ERROR MESSAGE AND STOP PROGRAM.
      * IF SCREEN CLEARED/UNFORMATTED , MOVE '00' TO PFKN.
        * MOVE INPUT MESSAGE FIELDS TO CORRESPONDING INAREA FIELDS IN COMPARM.
        * CALL DSN8IP7 (COMMAREA)
        * MOVE OUTAREA FIELDS IN PCONVSTA TO CORRESPONDING OUTPUT MESSAGE FIELDS.
        * IF LASTSCR 'DSN8001' MOVE 'DSN8IPFO' TO MODNAME FIELD
          ELSE MOVE 'DSN8IPEO' TO MODNAME FIELD.
        * CALL DLI ISRT OUTPUT MESSAGE.
        * IF STATUS CODE NOT OK THEN SEND ERROR MESSAGE AND STOP PROGRAM.
      * END.
/* * 012260000 */ 012300000
/*****************************************************************************/ 012400000
/* ** FIELDS SENT TO MESSAGE ROUTINE */ 012500000
/*****************************************************************************/ 012600000
DCL  MODULE  CHAR (07) INIT ('DSN8IP6'); 012800000
DCL  OUTMSG  CHAR (69); 012900000
1/*******************************************************************************/ 013000000
/* DECLARATION FOR INPUT: MIDNAME DSN8IPFI/DSN8IPEI */ 013100000
/*******************************************************************************/ 013200000
0DCL 1  IN_MESSAGE Static, 013300000
  2 LL  BIN Fixed (31), 013400000
  2 Z1  CHAR (3), 013500000
  2 Z2  CHAR (1), 013600000
  2 TC_CODE  CHAR (7), 013700000
  2 MESSAGE, 013800000
    3 INPUT, 013900000
      5 MAJSYS  CHAR (1), 014000000
      5 ACTION  CHAR (1), 014100000
      5 OBJFLD  CHAR (2), 014200000
      5 SEARCH  CHAR (2), 014300000
      5 DATA  CHAR (68), 014400000
      5 TRANDATA(15) CHAR (40); 014500000
  1/*******************************************************************************/ 014600000
/* DECLARATION FOR OUTPUT: MODNAME DSN8IPFO/DSN8IPEO */ 014700000
/*******************************************************************************/ 014800000
0DCL 1  OUT_MESSAGE Static, 014900000
  2 LL  BIN Fixed (31) INIT (1613), 015000000
  2 ZZ  BIN Fixed (15) INIT (0), 015100000
  2 OUTPUT, 015200000
    3 OUTPUTAREA, 015300000
      5 MAJSYS  CHAR (1), 015400000
      5 ACTION  CHAR (1), 015500000
      5 OBJFLD  CHAR (2), 015600000
      5 SEARCH  CHAR (2), 015700000
      5 DATA  CHAR (60), 015800000
      5 TRANDATA(15) CHAR (40); 015900000
  2 Z2  CHAR (7), 016000000
  2 Z1  CHAR (1), 016100000
  2 LL  BIN Fixed (31), 016200000
  2 ZZ  BIN Fixed (15) INIT (0), 016300000
    2 MESSAGE, 016400000
      2 TC_CODE  CHAR (7), 016500000
      2 Z2  CHAR (1), 016600000
      2 Z1  CHAR (1), 016700000
      2 LL  BIN Fixed (31), 016800000
      2 ZZ  BIN Fixed (15) INIT (0), 016900000
      2 OUTPUT, 017000000
        7 LINE (15) CHAR (79); 017100000
  7 LINE (15) CHAR (79); 017200000
  1/*******************************************************************************/ 017300000
/* DECLARATION FOR PGM-LOGIC */ 017400000
/*******************************************************************************/ 017500000
0DCL ONE  BIN Fixed (31) INIT (1) Static; 017600000
DCL THREE  BIN Fixed (31) INIT (3) Static; 017700000
DCL FOUR  BIN Fixed (31) INIT (4) Static; 017800000
0DCL GU_FKT  CHAR (4) INIT ('GU ') Static; 017900000
DCL ISRT_FKT  CHAR (4) INIT ('ISRT') Static; 018000000
DCL CHNG_FKT  CHAR (4) INIT ('CHNG') Static; 018100000
DCL ROLL_FKT  CHAR (4) INIT ('ROLL') Static; 018200000
0DCL MODNAME  CHAR (8) Static; 018300000
0DCL (ADDR,LOW) Builtin; 018400000
0DCL PLITDLI external Entry; 018500000
DCL DSN8IP7 external Entry; 018600000
0DCL (IOPCB_ADDR,ALTPCB_ADDR) Pointer; 018700000
0DCL DSN8MPG external Entry; 018800000
1*******************************************************************************/ 018900000
/* DECLAREATION FOR IO / ALTPCB Mask */ 019000000
1*******************************************************************************/ 019100000
0DCL 1 IOPCB  based (IOPCB_ADDR), 019200000
  2 IOLTERM  CHAR (8), 019300000
  2 FILLER  CHAR (2), 019400000
  2 STC_CODE  CHAR (2), 019500000
  2 CDATE  CHAR (4), 019600000
  2 CTIME  CHAR (4), 019700000
  2 SEQNUM  CHAR (4), 019800000
  2 MOD_NAME  CHAR (8), 019900000
  2 USERID  CHAR (8), 020000000
0DCL 1 ALTPCB  based (ALTPCB_ADDR), 020100000
  2 ALTTERM  CHAR (8), 020200000
  2 ALTTERM  CHAR (8), 020300000
ASSIGN COBOL WORK AREA /INITIALIZATIONS *
****************************************************************
/*         ALLOCATE COBOL WORK AREA /INITIALIZATIONS            */
****************************************************************

0 ALLOCATE COMMAREA SET(COMMPTR);
  COMMAREA = '';                      /* CLEAR COMMON AREA*/
  IN_MESSAGE = '';                      /* CLEAR INPUT FIELD*/
  MODNAME                = 'DSN8IPFO';  /* GET MODULE NAME */
  DSN8_MODULE_NAME.MAJOR = 'DSN8IP6';   /* GET MODULE NAME  */
  OUTAREA.MAJSYS = 'P';                 /* MAJOR SYSTEM - P */

02180000

02190000

02200000

CSEND:                                                                 02570000

CEND:                                         /* RETURN           */   02790000
END DSN8IP6;                                                        02800000

Related reference
“Sample applications in IMS” on page 1346
A set of Db2 sample applications run in the IMS environment.

**DSN8IP7**

**THIS MODULE PERFORMS THE INCLUDES TO BRING IN THE SQL TABLE DCLS AND DCLGEN STRUCTURES AS WELL AS THE PARAMETER AREA.**

```
DSN8IP7:PROC (COMMPTR) ;
/*******************************************************************************/
* * MODULE NAME = DSN8IP7
* * DESCRIPTIVE NAME = SAMPLE APPLICATION
* * SQL 1 MAINLINE
* * IMS
* * PL/I
* * COPYRIGHT = 5740-XYR (C) COPYRIGHT IBM CORP 1982, 1985
* * REFER TO COPYRIGHT INSTRUCTIONS FORM NUMBER G120-2083
* * STATUS = RELEASE 2, LEVEL 0
* * FUNCTION = THIS MODULE PERFORMS THE INCLUDES TO BRING IN THE
* * SQL TABLE DCLS AND DCLGEN STRUCTURES AS WELL AS
* * THE PARAMETER AREA.
* * INCLUDE DSN8MP1.
* * CALL DSN8IP8.
* * RETURN TO DSN8IP6.
* * NOTES = NONE
* * MODULE TYPE = PL/I PROC(COMMPTR).
* * PROCESSOR = DB2 PRECOMPILER, PL/I OPTIMIZER
* * MODULE SIZE = SEE LINKEDIT
* * ATTRIBUTES = REUSABLE
* * ENTRY POINT = DSN8IP7
* * PURPOSE = SEE FUNCTION
* * LINKAGE = CALLED BY DSN8IP6
* * INPUT = PARAMETERS EXPLICITLY PASSED TO THIS FUNCTION:
* *   SYMBOLIC LABEL/NAME = COMMPTR
* *   DESCRIPTION = POINTER TO COMMAREA
* *   COMMON AREA.
* *   SYMBOLIC LABEL/NAME = PFKIN
* *   DESCRIPTION = 00/01/02/03/07/08/10
* *   SYMBOLIC LABEL/NAME = INAREA
* *   DESCRIPTION = USER INPUT
* * OUTPUT = PARAMETERS EXPLICITLY RETURNED:
* *   COMMON AREA.
* *   SYMBOLIC LABEL/NAME = OUTAREA
* *   DESCRIPTION =GENERAL MENU OR
* *   SECONDARY SELECTION MENU
* *   SYMBOLIC LABEL/NAME = LASTSCR
* *   DESCRIPTION = DSN8001/DSN8002
* * EXIT-NORMAL = DSN8IP6
* * EXIT-ERROR = DSN8IP6
* * RETURN CODE = NONE
* * ABEND CODES = NONE
* * ERROR-MESSAGES = NONE
* * EXTERNAL REFERENCES =
* * ROUTINES/SERVICES = NONE
* * DATA-AREAS = DSN8MPCA - PLI STRUCTURE FOR COMMAREA
```
**PSEUDOCODE**

PROCEDURE

INCLUDE DECLARATIONS.

INCLUDE DSN8MP1.

INCLUDE ERROR HANDLER.

CP1EXIT: (REFERRED BY DSN8MP1)

RETURN.

CP1CALL: (REFERRED BY DSN8MP1)

CALL 'DSN8IP8'(COMMPTR).

GO TO MP1SAVE. (LABEL IN DSN8MP1)

INCLUDE VALIDATION MODULES.

END.

EXEC SQL WHENEVER SQLERROR GO TO DB_ERROR;

EXEC SQL WHENEVER SQLWARNING GO TO DB_ERROR;

DCL MODULE CHAR (07) INIT ('DSN8IP7');

DCL OUTMSG CHAR (69);

EXEC SQL INCLUDE DSN8MPCA; /* INCLUDE COMMAREA */

DSN8_MODULE_NAME.MAJOR = 'DSN8IP7 '; /* INITIALIZE MODULE NAME*/

EXEC SQL INCLUDE DSN8MPCS; /* INCLUDE PCONA */

EXEC SQL INCLUDE DSN8MPOV; /* INCLUDE POPTVAL */

EXEC SQL INCLUDE DSN8MPVO; /* INCLUDE CURSOR */

EXEC SQL INCLUDE DSN8MP5; /* INCLUDE SEARCH CRITERIA*/

EXEC SQL INCLUDE DSN8MP1; /* INCLUDE SQL COMMAREA*/

EXEC SQL INCLUDE DSN8MPXX; /* INCLUDE ERRORHANDLER */

CP1EXIT:

RETURN; /* EXIT */

CP1CALL:

/* GO TO DSN8IP8 (SQL2) */

CALL DSN8IP8 (COMMPTR);

GO TO MP1SAVE;

EXEC SQL INCLUDE DSN8MP3; /* INCLUDE ACTION VALIDATION*/

EXEC SQL INCLUDE DSN8MP4; /* INCLUDE OBJECT VALIDATION*/

EXEC SQL INCLUDE DSN8MP5; /* INCLUDE SEARCH CRITERIA*/

END; /* VALIDATION */

Related reference

“Sample applications in IMS” on page 1346
A set of Db2 sample applications run in the IMS environment.

**DSN8IP8**

ROUTER FOR SECONDARY SELECTION AND/OR DETAIL PROCESSING CALLS SECONDARY SELECTION MODULES DSN8MPM CALLS DETAIL MODULES DSN8MPV DSN8MPW DSN8MPX DSN8MPZ CALLED BY DSN8IP7 (SQL1).

```plaintext
DSN8IP8: PROC(COMMPTR) ;

/********************************************************************************
%PAGE;       00010000
DSN8IP8: PROC (COMMPTR) ;

/* ******************************************************************************
     * MODULE NAME = DSN8IP8                                               00020000
     * DESCRIPTIVE NAME = DB2 SAMPLE APPLICATION                              00030000
     *      SQL 2 COMMON MODULE                                               00040000
     *      IMS                                                              00050000
     *      PL/I                                                             00060000
     *      PROJECT                                                          00070000
     *      LICENSED MATERIALS - PROPERTY OF IBM                              00080000
     *      5695-DB2                                                          00090000
     *      (C) COPYRIGHT 1982, 1995 IBM CORP.  ALL RIGHTS RESERVED.           00010000
     *      STATUS = VERSION 4                                                00011000
     *      FUNCTION = ROUTER FOR SECONDARY SELECTION AND/OR DETAIL PROCESSIN 00012000
     *          CALLS SECONDARY SELECTION MODULES                              00013000
     *          DSN8MPM                                                          00014000
     *          CALLS DETAIL MODULES                                            00015000
     *          DSN8MPV DSN8MPW DSN8MPX DSN8MPZ                                00016000
     *          CALLED BY DSN8IP7 (SQL1)                                       00017000
     *      NOTES = NONE                                                       00018000
     *      MODULE TYPE = BLOCK OF PL/I CODE                                   00019000
     *      PROCESSOR = DB2 PRECOMPILER, PL/I OPTIMIZER                       00020000
     *      MODULE SIZE = SEE LINKEDIT                                         00021000
     *      ATTRIBUTES = REUSABLE                                              00022000
     *      ENTRY POINT = DSN8IP8                                             00023000
     *      PURPOSE = SEE FUNCTION                                            00024000
     *      LINKAGE = NONE                                                     00025000
     *      INPUT =                                                           00026000
     *          SYMBOLOC LABEL/NAME = COMMPTR                                 00027000
     *          DESCRIPTION = POINTER TO COMMAREA                             00028000
     *      OUTPUT =                                                          00029000
     *          SYMBOLOC LABEL/NAME = COMMPTR                                 00030000
     *          DESCRIPTION = POINTER TO COMMAREA                             00031000
     *      EXIT-NORMAL =                                                     00032000
     *      EXIT-ERROR = IF SQLERROR OR SQLWARNING, SQL WHENEVER CONDITION    00033000
     *          SPECIFIED IN DSN8IP8 WILL BE RAISED AND PROGRAM               00034000
     *          WILL GO TO THE LABEL DB_ERROR.                                 00035000
     *      RETURN CODE = NONE                                                00036000
     *      ABEND CODES = NONE                                                00037000
     *      ERROR-MESSAGES =                                                  00038000
     *          DSN8062E-AN OBJECT WAS NOT SELECTED                            00039000
     *          DSN8066E-UNSUPPORTED PFK OR LOGIC ERROR                        00040000
     *          DSN8072E-INVALID SELECTION ON SECONDARY SCREEN                00041000
     *          SPECIFIED IN DSN8IP8 WILL BE RAISED AND PROGRAM               00042000
     *          WILL GO TO THE LABEL DB_ERROR.                                 00043000
     *          ABEND CODES = NONE                                             00044000
     *      EXTERNAL REFERENCES = NONE                                       00045000
     *      ROUTINES/SERVICES = 8 MODULES LISTED ABOVE                        00046000
     *          DSN8MPG - ERROR MESSAGE ROUTINE                               00047000
     *          DSN8MPV - DCLGEN FOR DEPARTMENT                                00048000
     *          DSN8MPW - DCLGEN FOR DEPARTMENT                                00049000
     *          DSN8MPX - DCLGEN FOR DEPARTMENT                                00050000
     *          DSN8MPZ - DCLGEN FOR DEPARTMENT                                00051000
     *          DSN8MPAC - DCLGEN FOR ACTIVITY TYPES                          00052000
     *          DSN8MPAS - CURSOR SECONDARY SELECTION FOR STAFF               00053000
     *          DSN8MPCA - COMMUNICATION AREA BETWEEN MODULES                  00054000
     *          DSN8MPDH - CURSOR FOR DISPLAY TEXT FROM                       00055000
     *          TDSPTXT TABLE                                                   00056000
     *          DSN8MPPO - DCLGEN FOR DEPARTMENT                                00057000
     *     STATUS = VERSION 4                                                00058000
     *     (C) COPYRIGHT 1982, 1995 IBM CORP.  ALL RIGHTS RESERVED.           00010000
     *     LICENSED MATERIALS - PROPERTY OF IBM                              00011000
     *     PROCESSOR = DB2 PRECOMPILER, PL/I OPTIMIZER                       00012000
     *     MODULE SIZE = SEE LINKEDIT                                         00013000
     *     ATTRIBUTES = REUSABLE                                              00014000
     *     ENTRY POINT = DSN8IP8                                             00015000
     *     PURPOSE = SEE FUNCTION                                            00016000
     *     LINKAGE = NONE                                                     00017000
     *     INPUT =                                                           00018000
     *        SYMBOLOC LABEL/NAME = COMMPTR                                 00019000
     *        DESCRIPTION = POINTER TO COMMAREA                             00020000
     *     OUTPUT =                                                          00021000
     *        SYMBOLOC LABEL/NAME = COMMPTR                                 00022000
     *        DESCRIPTION = POINTER TO COMMAREA                             00023000
     *     EXIT-NORMAL =                                                     00024000
     *     EXIT-ERROR = IF SQLERROR OR SQLWARNING, SQL WHENEVER CONDITION    00025000
     *        SPECIFIED IN DSN8IP8 WILL BE RAISED AND PROGRAM               00026000
     *        WILL GO TO THE LABEL DB_ERROR.                                 00027000
     *     RETURN CODE = NONE                                                00028000
     *     ABEND CODES = NONE                                                00029000
     *     ERROR-MESSAGES =                                                  00030000
     *        DSN8062E-AN OBJECT WAS NOT SELECTED                            00031000
     *        DSN8066E-UNSUPPORTED PFK OR LOGIC ERROR                        00032000
     *        DSN8072E-INVALID SELECTION ON SECONDARY SCREEN                00033000
     *        SPECIFIED IN DSN8IP8 WILL BE RAISED AND PROGRAM               00034000
     *        WILL GO TO THE LABEL DB_ERROR.                                 00035000
     *        ABEND CODES = NONE                                             00036000
     *     EXTERNAL REFERENCES = NONE                                       00037000
     *     ROUTINES/SERVICES = 8 MODULES LISTED ABOVE                        00038000
     *        DSN8MPG - ERROR MESSAGE ROUTINE                               00039000
     *        DSN8MPV - DCLGEN FOR DEPARTMENT                                00040000
     *        DSN8MPW - DCLGEN FOR DEPARTMENT                                00041000
     *        DSN8MPX - DCLGEN FOR DEPARTMENT                                00042000
     *        DSN8MPZ - DCLGEN FOR DEPARTMENT                                00043000
     *        DSN8MPAC - DCLGEN FOR ACTIVITY TYPES                          00044000
     *        DSN8MPAS - CURSOR SECONDARY SELECTION FOR STAFF               00045000
     *        DSN8MPCA - COMMUNICATION AREA BETWEEN MODULES                  00046000
     *        DSN8MPDH - CURSOR FOR DISPLAY TEXT FROM                       00047000
     *        TDSPTXT TABLE                                                   00048000
     *        DSN8MPPO - DCLGEN FOR DEPARTMENT                                00049000
     */

1374  Db2 12 for z/OS: Application Programming and SQL Guide
```
CALL THE DETAIL PROCESSOR
ELSE
RETURN TO SQL 1

CALL SECONDARY SELECTION

SUBCASE ('DISPLAY','ERASE','UPDATE')

ENDSUB

RETURN TO SQL 1

DETAIL PROCESSOR

SUBCASE ('ADD')

CASE (NEW REQUEST)

DETERMINE IF NEW REQUEST.

IF ANSWER TO A SECONDARY SELECTION THEN
ELSE
NEW REQUEST

IF NOT WILLING TO ACCEPT AN ANSWER THEN
NEW REQUEST

CASE (NEW REQUEST)

SUBCASE ('ADD')

CALL SECONDARY SELECTION

IF # OF POSSIBLE CHOICES IS  1 THEN
RETURN TO SQL 1
ELSE
CALL THE DETAIL PROCESSOR

RETURN TO SQL 1

*PSEUDOCODE*

CHANGE-ACTIVITY =  NONE
TABLES =  NONE

SQLCA           - SQL COMMUNICATION AREA

CONTROL-BLOCKS =

SQLCA           - SQL COMMUNICATION AREA

TABLES =  NONE

SQLCA           - SQL COMMUNICATION AREA

CHANGE-ACTIVITY =  NONE

*PSEUDOCODE*

THIS MODULE DETERMINES WHICH SECONDARY SELECTION AND/OR
DETAIL MODULE(S) ARE TO BE CALLED IN THE IMS/PL/I
ENVIRONMENT.

WHAT HAS HAPPENED SO FAR?.................THE SUBSYSTEM

DEPENDENT MODULE (IMS,CICS,TSO) OR (SQL 0) HAS

READ THE INPUT SCREEN, FORMATTED THE INPUT AND PASSED CONTROL

TO SQL 1. SQL 1 PERFORMS VALIDATION ON THE SYSTEM DEPENDENT

FIELDS (MAJOR SYSTEM, ACTION, OBJECT, SEARCH CRITERIA). IF

ALL SYSTEM FIELDS ARE VALID SQL 1 PASSED CONTROL TO THIS

MODULE. PASSED PARAMETERS CONSIST ONLY OF A POINTER WHICH

POINTS TO A COMMUNICATION CONTROL AREA USED TO COMMUNICATE

BETWEEN SQL 0, SQL 1, SQL 2 AND THE SECONDARY SELECTION

AND DETAIL MODULES.

WHAT IS 'INCLUDED' IN THIS MODULE?............

ALL SECONDARY SELECTION AND DETAIL MODULES ARE 'INCLUDED'.

ALL VARIABLES KNOWN IN THIS PROCEDURE ARE KNOWN IN THE

SUB PROCEDURES. ALL SQL CURSOR DEFINITIONS AND

SQL 'INCLUDES' ARE DONE IN THIS PROCEDURE. BECAUSE OF THE

RESTRICTION THAT CURSOR HOST VARIABLES MUST BE DECLARED BEFORE

THE CURSOR DEFINITION ALL CURSOR HOST VARIABLES ARE DECLARED

IN THIS PROCEDURE.

PROCEDURE

IF ANSWER TO DETAIL SCREEN & DETAIL PROCESSOR

IS NOT WILLING TO ACCEPT AN ANSWER THEN

NEW REQUEST

ELSE

IF ANSWER TO A SECONDARY SELECTION THEN

DETERMINE IF NEW REQUEST.

CASE (NEW REQUEST)

SUBCASE ('ADD')

DETAIL PROCESSOR

RETURN TO SQL 1

ENDSUB

SUBCASE ('DISPLAY','ERASE','UPDATE')

CALL SECONDARY SELECTION

IF # OF POSSIBLE CHOICES IS  1 THEN
RETURN TO SQL 1
ELSE
CALL THE DETAIL PROCESSOR

RETURN TO SQL 1
* ENDSUB * 01529000
* ENDCASE * 01530000
* IF ANSWER TO SECONDARY SELECTION AND A SELECTION HAS * 01558000
* ACTUALLY BEEN MADE THEN * 01566000
* VALID SELECTION #? * 01579000
* IF IT IS VALID THEN * 01590000
* CALL DETAIL PROCESSOR * 01600000
* RETURN TO SQL 1 * 01610000
* ELSE * 01620000
* PRINT ERROR MSG * 01630000
* RETURN TO SQL 1. * 01640000
* IF ANSWER TO SECONDARY SELECTION THEN * 01656000
* CALL SECONDARY SELECTION * 01660000
* RETURN TO SQL 1. * 01670000
* IF ANSWER TO DETAIL THEN * 01700000
* CALL DETAIL PROCESSOR * 01710000
* RETURN TO SQL 1. * 01720000
* END. * 01730000
* 01740000
* +EXAMPLE- A ROW IS SUCCESSFULLY ADDED, THE OPERATOR RECEIVES * 01750000
* THE SUCCESSFULLY ADDED MESSAGE AND JUST HITS ENTER. * 01760000
* 01770000
* END. * 01780000
* +--------------------------------------------------------------------------------------*/ 01790000
*/ INCLUDE DECLARES */
EXEC SQL INCLUDE DSN8MPCA; /*COMMUNICATION AREA BETWEEN MODULES */ 01800000
EXEC SQL INCLUDE SQLCA; /*SQL COMMUNICATION AREA */ 01830000
EXEC SQL INCLUDE DSN8MPDP; /*DCLGEN FOR DEPARTMENT */ 01850000
EXEC SQL INCLUDE DSN8MPEMP; /* DCLGEN FOR EMPLOYEE */ 01860000
EXEC SQL INCLUDE DSN8MPPP; /* DCLGEN FOR PROJECTS */ 01870000
EXEC SQL INCLUDE DSN8MPPT; /* DCLGEN FOR ACTIVITY TYPES */ 01880000
EXEC SQL INCLUDE DSN8MPPA; /* DCLGEN FOR PROJECT/ACTIVITIES */ 01890000
EXEC SQL INCLUDE DSN8MPMP; /* DCLGEN FOR PROJECT/STAFFING */ 01900000
EXEC SQL INCLUDE DSN8MPPR; /* DCLGEN FOR PROJ/RESP EMPLOYEE */ 01910000
EXEC SQL INCLUDE DSN8MPDD; /* DCLGEN FOR PROJ STRUCTURE DETAIL */ 01920000
EXEC SQL INCLUDE DSN8MPPD2; /* DCLGEN FOR PROJ STRUCTURE DETAIL */ 01930000
EXEC SQL INCLUDE DSN8MPSA; /* DCLGEN FOR PROJ ACTIVITY LISTING */ 01940000
EXEC SQL INCLUDE DSN8MPSA2; /* DCLGEN FOR PROJ ACTIVITY LISTING */ 01950000
EXEC SQL INCLUDE DSN8MPPE; /* DCLGEN FOR PROJECT-EMPLOYEE */ 01955000
EXEC SQL INCLUDE DSN8MPED; /* DCLGEN FOR EMPLOYEE-DEPT */ 01957000
EXEC SQL INCLUDE DSN8MPPOV; /* DCLGEN FOR OPTION VALIDATION */ 01960000
EXEC SQL INCLUDE DSN8MPDT; /* DCLGEN FOR DISPLAY TEXT TABLE */ 01970000
EXEC SQL INCLUDE DSN8MPDP; /* DCLGEN FOR DEPARTMENT */ 01980000
EXEC SQL INCLUDE DSN8MPCA; /* COMMUNICATION AREA BETWEEN MODULES */ 01990000
EXEC SQL INCLUDE SQLCA; /* SQL COMMUNICATION AREA */ 02000000
EXEC SQL INCLUDE DSN8MPCA; 02010000
EXEC SQL INCLUDE DSN8MPCA; 02020000
EXEC SQL INCLUDE DSN8MPCA; 02030000
EXEC SQL INCLUDE DSN8MPCA; 02040000
EXEC SQL INCLUDE DSN8MPCA; 02050000
EXEC SQL INCLUDE DSN8MPCA; 02060000
EXEC SQL INCLUDE DSN8MPCA; 02070000
EXEC SQL INCLUDE DSN8MPCA; 02080000
EXEC SQL INCLUDE DSN8MPCA; 02090000
EXEC SQL INCLUDE DSN8MPCA; 02100000
EXEC SQL INCLUDE DSN8MPCA; 02110000
EXEC SQL INCLUDE DSN8MPCA; 02120000
EXEC SQL INCLUDE DSN8MPCA; 02130000
EXEC SQL INCLUDE DSN8MPCA; 02140000
EXEC SQL INCLUDE DSN8MPCA; 02150000
EXEC SQL INCLUDE DSN8MPCA; 02160000
EXEC SQL INCLUDE DSN8MPCA; 02170000
EXEC SQL INCLUDE DSN8MPCA; 02180000
EXEC SQL INCLUDE DSN8MPCA; 02190000
EXEC SQL INCLUDE DSN8MPCA; 02200000
EXEC SQL INCLUDE DSN8MPCA; 02210000
EXEC SQL INCLUDE DSN8MPCA; 02220000
EXEC SQL INCLUDE DSN8MPCA; 02230000
EXEC SQL INCLUDE DSN8MPCA; 02240000
EXEC SQL INCLUDE DSN8MPCA; 02250000
EXEC SQL INCLUDE DSN8MPCA; 02260000
EXEC SQL INCLUDE DSN8MPCA; 02270000
EXEC SQL INCLUDE DSN8MPCA; 02280000
EXEC SQL INCLUDE DSN8MPCA; 02290000
EXEC SQL INCLUDE DSN8MPCA; 02300000
EXEC SQL INCLUDE DSN8MPCA; 02310000
/* CURSORS */
EXEC SQL INCLUDE DSN8MPPL; /* MAJYS P - SEC SEL FOR PS, AL, PR*/ 02000000
EXEC SQL INCLUDE DSN8MPPL; /* MAJYS P - SEC SEL FOR AE */ 02010000
EXEC SQL INCLUDE DSN8MPPL; /* MAJYS P - SEC SEL FOR AS */ 02020000
EXEC SQL INCLUDE DSN8MPPL; /* MAJYS P - DETAIL FOR PS */ 02030000
EXEC SQL INCLUDE DSN8MPPL; /* MAJYS P - DETAIL FOR AL */ 02040000
EXEC SQL INCLUDE DSN8MPPL; /* PROG TABLES - DISPLAY HEADINGS */ 02050000
EXEC SQL INCLUDE DSN8MPPL; /* PROG TABLES - DISPLAY HEADINGS */ 02060000
EXEC SQL INCLUDE DSN8MPPL; /* PROG TABLES - DISPLAY HEADINGS */ 02070000
EXEC SQL INCLUDE DSN8MPPL; /* PROG TABLES - DISPLAY HEADINGS */ 02080000
EXEC SQL INCLUDE DSN8MPPL; /* PROG TABLES - DISPLAY HEADINGS */ 02090000
EXEC SQL INCLUDE DSN8MPPL; /* PROG TABLES - DISPLAY HEADINGS */ 02100000
EXEC SQL INCLUDE DSN8MPPL; /* PROG TABLES - DISPLAY HEADINGS */ 02110000
EXEC SQL INCLUDE DSN8MPPL; /* PROG TABLES - DISPLAY HEADINGS */ 02120000
EXEC SQL INCLUDE DSN8MPPL; /* PROG TABLES - DISPLAY HEADINGS */ 02130000
EXEC SQL INCLUDE DSN8MPPL; /* PROG TABLES - DISPLAY HEADINGS */ 02140000
EXEC SQL INCLUDE DSN8MPPL; /* PROG TABLES - DISPLAY HEADINGS */ 02150000
EXEC SQL INCLUDE DSN8MPPL; /* PROG TABLES - DISPLAY HEADINGS */ 02160000
EXEC SQL INCLUDE DSN8MPPL; /* PROG TABLES - DISPLAY HEADINGS */ 02170000
EXEC SQL INCLUDE DSN8MPPL; /* PROG TABLES - DISPLAY HEADINGS */ 02180000
EXEC SQL INCLUDE DSN8MPPL; /* PROG TABLES - DISPLAY HEADINGS */ 02190000
EXEC SQL INCLUDE DSN8MPPL; /* PROG TABLES - DISPLAY HEADINGS */ 02200000
EXEC SQL INCLUDE DSN8MPPL; /* PROG TABLES - DISPLAY HEADINGS */ 02210000
EXEC SQL INCLUDE DSN8MPPL; /* PROG TABLES - DISPLAY HEADINGS */ 02220000
EXEC SQL INCLUDE DSN8MPPL; /* PROG TABLES - DISPLAY HEADINGS */ 02230000
EXEC SQL INCLUDE DSN8MPPL; /* PROG TABLES - DISPLAY HEADINGS */ 02240000
EXEC SQL INCLUDE DSN8MPPL; /* PROG TABLES - DISPLAY HEADINGS */ 02250000
EXEC SQL INCLUDE DSN8MPPL; /* PROG TABLES - DISPLAY HEADINGS */ 02260000
EXEC SQL INCLUDE DSN8MPPL; /* PROG TABLES - DISPLAY HEADINGS */ 02270000
EXEC SQL INCLUDE DSN8MPPL; /* PROG TABLES - DISPLAY HEADINGS */ 02280000
EXEC SQL INCLUDE DSN8MPPL; /* PROG TABLES - DISPLAY HEADINGS */ 02290000
EXEC SQL INCLUDE DSN8MPPL; /* PROG TABLES - DISPLAY HEADINGS */ 02300000
EXEC SQL INCLUDE DSN8MPPL; /* PROG TABLES - DISPLAY HEADINGS */ 02310000
DCL LENGTH BUILTIN;
/*****************************************************************************/ 02090000
*/ ** FIELDS SENT TO MESSAGE ROUTINE **/ 02100000
/*****************************************************************************/ 02110000
DCL MODULE CHAR (07) INIT ('DSN8MP8'); 02120000
DCL OUTMSG CHAR (69); 02130000

DCL DSN8MPG EXTERNAL ENTRY; 02140000

/* SQL RETURN CODE HANDLING */ 02150000
/*****************************************************************************/ 02160000
EXEC SQL WHENEVER SQLERROR GO TO DB_ERROR; 02170000
EXEC SQL WHENEVER SQLWARNING GO TO DB_ERROR; 02180000

0 DCL UNSPEC BUILTIN;
DCL VERIFY BUILTIN;
/*****************************************************************************/ 02190000
*/ INITIALIZATIONS */ 02200000
/*****************************************************************************/ 02210000
DSNB_MODULE_NAME.MAJOR='DSN8IP8';

DSNB_MODULE_NAME.MINOR=' ';

/***********************************************************/
/* DETERMINES WHETHER NEW REQUEST OR NOT */
/***********************************************************/
/* IF 'NO ANSWER POSSIBLE' SET BY DETAIL PROCESSOR THEN FORCE A */
/* NEW REQUEST. */
IF PCONVSTA.PREV = ' ' THEN COMPARM.NEWREQ = 'Y';

/* IF ANSWER TO SECONDARY SELECTION THEN DETERMINE IF REALLY A */
/* NEW REQUEST. IT WILL BE CONSIDERED A NEW REQUEST IF POSITIONS*/
/* 3 TO 60 ARE NOT ALL BLANK AND THE ENTERED DATA IF NOT 'NEXT' */
IF COMPARM.NEWREQ = 'N' & PCONVSTA.PREV = 'S' &
    SUBSTR(COMPARM.DATA,3,58) ^= ' ' &
    COMPARM.DATA ^= 'NEXT'
THEN COMPARM.NEWREQ = 'Y';

/***********************************************************/
/* IF NEW REQUEST AND ACTION IS 'ADD' THEN */
/* CALL DETAIL PROCESSOR */
/* ELSE CALL SECONDARY SELECTION */
/***********************************************************/
IF COMPARM.NEWREQ='Y' THEN
    DO;
        IF COMPARM.ACTION = 'A' THEN
            DO;
                CALL DETAIL; /* CALL DETAIL PROCESSOR */
                GO TO EXIT; /* RETURN */
            END;
        END;
        CALL SECSEL; /* CALL SECONDARY SELECTION */
    END;

IF MAXSEL = 1 THEN /* IF NO. OF CHOICES = 1 */
    CALL DETAIL; /* CALL DETAIL PROCESSOR */
    GO TO EXIT; /* RETURN */
END;

/* IF ANSWER TO SECONDARY SELECTION AND NOT A SCROLLING REQUEST */
/* (INPUT NOT EQUAL TO 'NEXT') AND THE POSITIONS */
/* 1 TO 2 IN INPUT DATA FIELD NOT EQUAL TO POSITIONS 1 TO 2 */
/* IN OUTPUT DATA FIELD THEN SEE IF VALID SELECTION. */
/***********************************************************/
/* DETERMINES IF VALID SELECTION NUMBER */
/***********************************************************/
IF PCONVSTA.PREV ^= 'S' THEN GO TO IP201; /* TO SECONDARY SEL */
IF PCONVSTA.MAXSEL < 1  THEN GO TO IP201; /* NO VALID CHOICES */
IF COMPARM.DATA = 'NEXT' THEN GO TO IP201; /* SCROLL REQUEST */
IF SUBSTR(COMPARM.DATA,1,2) = SUBSTR(PCONVSTA.DATA,1,2)
THEN GO TO IP201; /* NO CHANGE ON INPUT SCREEN */
IF SUBSTR(COMPARM.DATA,2,1) = ' ' THEN /* SECOND CHAR BLANK */
    IF VERIFY(SUBSTR(COMPARM.DATA,1,1),'123456789') = 0 THEN
        SUBSTR(COMPARM.DATA,2,1) = SUBSTR(COMPARM.DATA,1,1);
        END;
    END;
    IF VERIFY(SUBSTR(COMPARM.DATA,1,2),'0123456789') = 0 &
        SUBSTR(COMPARM.DATA,1,2) > '00' THEN
        IF DATAP <= PCONVSTA.MAXSEL THEN
            DO;
                COMPARM.NEWREQ = 'Y'; /* TELL DETAIL PROCESSOR NEW REQ */
                CALL DETAIL; /* CALL DETAIL PROCESSOR */
                GO TO EXIT; /* RETURN */
            END;
        END;
        /* INVALID SECONDARY */
        /* SELECTION */
        /* PRINT ERROR MESSAGE */
        CALL DSN8MPG (MODULE, '072E', OUTMSG);
PCONVSTA.MSG = OUTMSG;
GO TO EXIT;      /* RETURN */

/*****************************/
/* DETERMINES WHETHER SECONDARY SELECTION OR DETAIL */
/*****************************/

/* MUST BE ANY ANSWER TO EITHER SEC SEL OR DETAIL */
IP201:
IF PCONVSTA.PREV = 'S' THEN
  DO;
    CALL SECSEL;   /* CALL SECONDARY SELECTION*/
    GO TO EXIT;   /* RETURN */
  END;
IF PCONVSTA.PREV = 'D' THEN
  DO;
    CALL DETAIL;   /* CALL DETAIL PROCESSOR */
    GO TO EXIT;   /* RETURN */
  END;
/* LOGIC ERROR */
/* PRINT ERROR MESSAGE */
CALL DSN8MPG (MODULE, '066E', OUTMSG);
PCONVSTA.MSG= OUTMSG;
GO TO EXIT;   /* RETURN */
EXEC SQL INCLUDE DSN8MPXX;   /* HANDLES SQL ERRORS */
GO TO EXIT;   /* RETURN */

/*****************************/
/* CALLS SECONDARY SELECTION AND RETURNS TO SQL 1 */
/*****************************/

SECSEL: PROC;   /* CALL APPROPRIATE SECONDARY SELECTION MODULE */
PCONVSTA.LASTSCR = 'DSN8001';  /* NOTE GENERAL SCREEN */
IF COMPARM.OBJFLD='AE' |  /*ACTIVITY ESTIMATE */
  COMPARM.OBJFLD='AL' |  /*PROJECT ACTIVITY LISTING */
  COMPARM.OBJFLD='AS' |  /*INDIVIDUAL PROJECT STAFFING */
  COMPARM.OBJFLD='PR' |  /*INDIVIDUAL PROJECT PROCESSING*/
  COMPARM.OBJFLD='PS' THEN  /*PROJECT STRUCTURE */
    DO;
      CALL DSN8MPM;   /*SECONDARY SELECTION FOR PROJECTS*/
      RETURN;        /*RETURN */
    END;
    /* SECONDARY SELECTION MISSING */
    /* PRINT ERROR MESSAGE */
    CALL DSN8MPG (MODULE, '062E', OUTMSG);
    PCONVSTA.MSG= OUTMSG;
    GO TO EXIT;
END SECSEL;
/* CALLS DETAIL PROCESSOR AND RETURNS TO SQL 1 */
/* NOTE - SAME SECONDARY SELECTION MODULE FOR DS, DE AND EM */
/*****************************/

DETAIL: PROC;   /* CALL APPROPRIATE DETAIL MODULE */
PCONVSTA.LASTSCR = 'DSN8002';  /* SET FOR DETAIL MAP */
IF COMPARM.OBJFLD='PS' THEN
  DO;
    CALL DSN8MPV;   /* PROJECT STRUCTURE DETAIL */
    RETURN;        /*RETURN */
  END;
IF COMPARM.OBJFLD='AL' THEN
  DO;
    CALL DSN8MPT;   /* PROJECT ACTIVITY LIST */
    RETURN;        /*RETURN */
  END;
IF COMPARM.OBJFLD='PR' THEN
  DO;
    CALL DSN8MPZ;    /* PROJECT DETAIL */
    RETURN;         /*RETURN */
  END;
IF COMPARM.OBJFLD='AE' THEN
  DO;
    CALL DSN8MPX;    /* ACTIVITY ESTIMATE DETAIL */
    RETURN;         /*RETURN */
  END;
Related reference

“Sample applications in IMS” on page 1346
A set of Db2 sample applications run in the IMS environment.

**DSN8IP3**

**THIS MODULE LISTS EMPLOYEE PHONE NUMBERS AND UPDATES THEM IF DESIRED.**

```
DSN8IP3: PROC(IOPCB_ADDR,ALTPCB_ADDR) OPTIONS (MAIN);
/*------------------------------------------------------------------------------------------*/
/* * MODULE NAME = DSN8IP3 */
/* * DESCRIPTIVE NAME = DB2 SAMPLE APPLICATION */
/* PHONE APPLICATION */
/* IMS */
/* PL/I */
/* * COPYRIGHT = 5665-DB2 (C) COPYRIGHT IBM CORP 1982, 1991 */
/* SEE COPYRIGHT INSTRUCTIONS */
/* LICENSED MATERIALS - PROPERTY OF IBM */
/* * STATUS = VERSION 2 RELEASE 3, LEVEL 0 */
/* * FUNCTION = THIS MODULE LISTS EMPLOYEE PHONE NUMBERS AND UPDATES THEM IF DESIRED. */
/* * NOTES = */
/* * DEPENDENCIES = TWO MFS MAPS ARE REQUIRED: */
/* DSN8IP1 AND DSN8IPN */
/* * RESTRICTIONS = NONE */
/* * MODULE TYPE = PL/I PROC OPTIONS(MAIN) */
/* PROCESSOR = DB2 PRECOMPILER, PL/I OPTIMIZER */
/* MODULE SIZE = SEE LINKEDIT */
/* ATTRIBUTES = REENTRANT */
/* * ENTRY POINT = DSN8IP3 */
/* PURPOSE = SEE FUNCTION */
/* LINKAGE = INVOKED FROM IMS */
/* * INPUT = PARAMETERS EXPLICITLY PASSED TO THIS FUNCTION: */
/* input-message: */
/* * SYMBOLIC LABEL/NAME = DSN8IPNO */
/* DESCRIPTION = PHONE MENU 1 (SELECT) */
/* * SYMBOLIC LABEL/NAME = DSN8IPL */
/* DESCRIPTION = PHONE MENU 2 (UPDATE) */
/* * SYMBOLIC LABEL/NAME = VPHONE */
/* DESCRIPTION = VIEW OF TELEPHONE INFORMATION */
/* * SYMBOLIC LABEL/NAME = VEMPLP */
```
DESCRIPTION = VIEW OF EMPLOYEE INFORMATION

OUTPUT = PARAMETERS EXPLICITLY RETURNED:

OUTPUT-MESSAGE:

SYMBOLIC LABEL/NAME = DSN8IPNO
DESCRIPTION = PHONE MENU 1 (SELECT)

SYMBOLIC LABEL/NAME = DSN8IPLO
DESCRIPTION = PHONE MENU 2 (UPDATE)

EXIT-NORMAL = RETURN CODE 0 NORMAL COMPLETION

EXIT-ERROR =

RETURN CODE = NONE
ABEND CODES = NONE

ERROR-MESSAGES =
DSN8004I - EMPLOYEE SUCCESSFULLY UPDATED
DSN8007E - EMPLOYEE DOES NOT EXIST, UPDATE NOT DONE
DSN8008I - NO EMPLOYEE FOUND IN TABLE
DSN8058I - PRESS PA1 FOR NEXT PAGE / ENTER FOR
SELECTION MENU
DSN8060E - SQL ERROR, RETURN CODE IS:
DSN8064E - INVALID DL/I STC-CODE ON GU MSG
DSN8065E - INVALID DL/I STC-CODE ON ISRT MSG

EXTERNAL REFERENCES =
ROUTINES/SERVICES =
DSN80MPG - ERROR MESSAGE ROUTINE

DATA AREAS =
IN_MESSAGE - MFS INPUT
OUT_MESSAGE - MFS OUTPUT

CONTROL BLOCKS =
SQLCA - SQL COMMUNICATION AREA

TABLES = NONE

CHANGE-ACTIVITY = NONE

*PSEUDOCODE*

*PROCEDURE*
CALL DLI GU INPUT MESSAGE.
IF STATUS CODE NOT OK THEN SEND ERROR MESSAGE, PGM-STOP.
CASE (ACTION)

SUBCASE ('L')
IF LASTNAME IS '*' THEN
LIST ALL EMPLOYEES
PREPARE OUTPUT_MESSAGE
CALL DLI ISRT OUTPUT_MESSAGE
ELSE
IF LASTNAME CONTAINS ' %' THEN
LIST EMPLOYEES GENERIC
PREPARE OUTPUT_MESSAGE
CALL DLI ISRT OUTPUT_MESSAGE
ELSE
LIST EMPLOYEES SPECIFIC
PREPARE OUTPUT_MESSAGE
CALL DLI ISRT OUTPUT_MESSAGE
ENDSUB

SUBCASE ('U')
DO WHILE INPUT PHONE_NO ^= BLANK
UPDATE PHONE_NO FOR EMPLOYEE
END
PREPARE OUTPUT_MESSAGE
CALL DLI ISRT OUTPUT_MESSAGE
OTHERWISE
UNFORMATTED SCREEN
PREPARE OUTPUT_MESSAGE
CALL DLI ISRT OUTPUT MESSAGE
ENDSUB
* ENDCASE *
* IF SQL OR DL/I ERROR HAS OCCURRED THEN *
* ROLLBACK *
* PGM-STOP. *
* END. *
*
/-------------------------------------------------------------------*/
1/*********************************************************************/
/*       DECLARATION FOR INPUT: MODNAME DSN8IPNI/DSN8IPLI           */
/*********************************************************************/
0DCL  1 IN_MESSAGE       STATIC,
   2 LL             BIN FIXED (31),
   2 ZZ             CHAR (2),
   2 TC_CODE        CHAR (7),
   2 ACTION         CHAR (1),
   2 MESSAGE        CHAR (500);
0DCL  1 INPUT_1         BASED(ADDR(MESSAGE)),
   2 LNAME          CHAR (15),
   2 FNAME          CHAR (12);
0DCL  1 INPUT_2 (15)    BASED(ADDR(MESSAGE)),
   2 NEWNO          CHAR (4),
   2 ENO            CHAR (6);
-/*********************************************************************/
/*        DECLARATION FOR OUTPUT: MODNAME DSN8IPNO/DSN8IPLO          */
/*********************************************************************/
0DCL  1 OUT_MESSAGE      STATIC,
   2 LL             BIN FIXED (31),
   2 ZZ             BIN FIXED (15) INIT (0),
   2 ERROR          CHAR (79),
   2 OUTPUT         CHAR (1095);
0DCL  1 OUTPUT_1         BASED(ADDR(OUTPUT)),
   2 LASTNAME       CHAR (15),
   2 FIRSTNAME      CHAR (12);
0DCL  1 OUTPUT_2 (15)    BASED(ADDR(OUTPUT)),
   2 FIRSTNAME      CHAR (12),
   2 MIDDLEINITIAL  CHAR (1),
   2 LASTNAME       CHAR (15),
   2 PHONENUMBER    CHAR (4),
   2 EMPLOYEENUMBER CHAR (6),
   2 DEPTNUMBER     CHAR (3),
   2 DEPTNAME       CHAR (32);  /* 32 TO FIT ON ONE LINE */
DCL  CHAR_SQLCODE CHAR (14);
DCL  1 CHAR_SQLSTR BASED(ADDR(CHAR_SQLCODE)),
   2 CHAR_BLNK   CHAR(4),
   2 CHAR_SQLCOD CHAR(10);
0/*********************************************************************/
/*        DECLARATION FOR IO / ALTPCB MASK                           */
/*********************************************************************/
0DCL  (IOPCB_ADDR,ALTPCB_ADDR) POINTER;
0DCL  1 IOPCB        BASED (IOPCB_ADDR),
   2 IOLTERM    CHAR (8),
   2 FILLER     CHAR (2),
   2 STC_CODE   CHAR (2);
0DCL  1 ALTPCB       BASED (ALTPCB_ADDR),
   2 ALTLTERM   CHAR (8),
   2 FILLER     CHAR (2),
   2 STC_CODE   CHAR (2);
0/*********************************************************************/
/*        DECLARATION FOR PGM-LOGIC                                  */
/*********************************************************************/
0DCL  ONE          BIN FIXED (31) INIT (1) STATIC;
DCL  THREE        BIN FIXED (31) INIT (3) STATIC;
DCL  FOUR         BIN FIXED (31) INIT (4) STATIC;
0DCL  GU_FKT       CHAR (4) INIT ('GU ') STATIC;
DCL  ISRT_FKT     CHAR (4) INIT ('ISRT') STATIC;
DCL  CHNG_FKT     CHAR (4) INIT ('CHNG') STATIC;
DCL  ROLL_FKT     CHAR (4) INIT ('ROLL') STATIC;
0DCL  MODNAME      CHAR (8) STATIC;
0DCL  FIRST        BIT  (1) STATIC;
DCL  PHONE_NO     CHAR (6) STATIC;
0DCL  (I,M,ITAB) BIN FIXED(15);
0DCL (ADDR,INDEX,SUBSTR) BUILTIN;
0DCL  TRANSLATE BUILTIN;
0DCL  SYSPRINT EXTERNAL PRINT FILE;
0DCL  PLITDLI EXTERNAL ENTRY;
0DCL  DSN8MPG EXTERNAL ENTRY;
0DCL  DSN8MPG EXTERNAL ENTRY;

Chapter 12. Sample data and applications supplied with Db2 for z/OS 1381
DCL MODULE CHAR (07) INIT ('DSN8IP3');
DCL OUTMSG CHAR (69);

EXEC SQL INCLUDE SQLCA;
/* SQL COMMUNICATION AREA */
/* SQL DECLARATION FOR VIEW PHONE */
EXEC SQL DECLARE VPHONE TABLE
  (LASTNAME       VARCHAR(15)       ,
   FIRSTNAME      VARCHAR(12)       ,
   MIDDLEINITIAL  CHAR(1)          ,
   PHONENUMBER    CHAR(4)          ,
   EMPLOYEENUMBER CHAR(6)          ,
   DEPTNUMBER     CHAR(3) NOT NULL,
   DEPTNAME       VARCHAR(36) NOT NULL);
/* STRUCTURE FOR PHONE RECORD */
DCL 1 PPHONE,
  2 LASTNAME   CHAR (15) VAR,
  2 FIRSTNAME  CHAR (12) VAR,
  2 MIDDLEINITIAL  CHAR (1),
  2 PHONENUMBER CHAR (4),
  2 EMPLOYEENUMBER CHAR (6),
  2 DEPTNUMBER  CHAR (3) VAR;

EXEC SQL DECLARE VEMPLP TABLE
  (EMPLOYEENUMBER CHAR(6)       ,
   PHONENUMBER    CHAR(4));
/* STRUCTURE FOR PEMLP RECORD */
DCL 1 PEMP,
  2 EMPLOYEENUMBER CHAR (6),
  2 PHONENUMBER    CHAR (4);

EXEC SQL DECLARE TELE1 CURSOR FOR
  SELECT *
  FROM VPHONE;
EXEC SQL DECLARE TELE2 CURSOR FOR
  SELECT *
  FROM VPHONE
  WHERE LASTNAME LIKE :INPUT_1.LNAME
     AND FIRSTNAME LIKE :INPUT_1.FNAME;
EXEC SQL DECLARE TELE3 CURSOR FOR
  SELECT *
  FROM VPHONE
  WHERE LASTNAME  =    :INPUT_1.LNAME
     AND FIRSTNAME LIKE :INPUT_1.FNAME;

EXEC SQL WHENEVER SQLERROR  GOTO P3_DBERROR;
EXEC SQL WHENEVER SQLWARNING GOTO P3_DBERROR;
EXEC SQL WHENEVER NOT FOUND CONTINUE;

*/INITIALIZATIONS */
0P3_START:
  IN_MESSAGE = ''; /* SCREEN INPUT */
  OUT_MESSAGE = ''; /* SCREEN OUTPUT */
  OUT_MESSAGE.LL = 83; /* LINE LENGTH */
  MODNAME = 'DSN8IPNO'; /* MODULE NAME */
  FIRST = '1';
  ITAB = 0; /* COUNTER */
  CALL PLITDLI (THREE,GU_FKT,IOPCB,IN_MESSAGE);
    /* IF INVALID DL/I */
    /* STC-CODE ON GU MSG */
    /* PRINT ERROR MESSAGE*/
 0 IF IOPCB.STC_CODE ^= ' ' THEN
    DO;

CALL DSN8MPG (MODULE, '064E', OUTMSG);
ERROR = OUTMSG||IOPCB.STC_CODE;
CALL P3_SEND;
END;

/******************************************************************************
*                  SELECT ACTION                                          */
/******************************************************************************
0 SELECT (ACTION);
   WHEN ('L') DO; /* ACTION - LIST */

/******************************************************************************
* REDISPLAY SELECTION SCREEN IF NO CRITERIA ENTERED                      */
/******************************************************************************
0 IF INPUT_1.LNAME = ' ' &
   INPUT_1.FNAME = ' ' THEN
   DO;
      CALL P3_SEND;
      GOTO P3_END;
   END;

MODNAME = 'DSN8IPLO'; /* SELECT "LISTING" PANEL */

/******************************************************************************
* LIST ALL EMPLOYEES                                                      */
/******************************************************************************
0 IF INPUT_1.LNAME = '*' THEN /* LIST ALL EMPLOYEES */
   DO;
      EXEC SQL OPEN TELE1; /* OPEN CURSOR */
      EXEC SQL FETCH TELE1 /* GET FIRST RECORD */
         INTO :PPHONE;
   0 IF SQLCODE = 100 THEN /* NO EMPLOYEES FOUND */
      DO; /* PRINT ERROR MESSAGE*/
         MODNAME = 'DSN8IPNO';
         CALL DSN8MPG (MODULE, '008I', OUTMSG);
         ERROR = OUTMSG;
         CALL P3_SEND;
         GOTO P3_SELECT_20;
      END;
      CALL P3_PREPARE.Screen; /* LIST FIRST EMPLOYEE*/
      P3_SELECT_10:
      EXEC SQL FETCH TELE1 /* GET NEXT RECORD */
         INTO :PPHONE;
   0 IF SQLCODE = 100 THEN /* FINISHED? */
      DO;
         ERROR = '';
         CALL P3_SEND;
         GOTO P3_SELECT_20;
      END;
      CALL P3_PREPARE.Screen; /* LIST EMPLOYEE */
      GOTO P3_SELECT_10; /* CONTINUE */
   P3_SELECT_20:
      EXEC SQL CLOSE TELE1; /* CLOSE CURSOR */
      GOTO P3_END;
   0 END IF;
   END;
   ELSE DO; /* SELECT EMPLOYEES BY NAME*/
      IF INPUT_1.LNAME = ' ' &
         INPUT_1.FNAME = ' ' THEN /* IS NAME BLANK*/
         MODNAME = 'DSN8IPNO';
         CALL DSN8MPG (MODULE, '008I', OUTMSG);
         ERROR = OUTMSG;
         CALL P3_SEND;
         GOTO P3_SELECT_20;
      END;
      IF INDEX(INPUT_1.LNAME,'%') > 0 THEN /* SET PATTERN */
      IF INDEX(INPUT_1.FNAME,'%') > 0 THEN /* YES, SEARCH ON */
         EXEC SQL OPEN TELE2; /* OPEN CURSOR */
         EXEC SQL FETCH TELE2 /* GET FIRST RECORD */
            INTO :PPHONE;
      END;
IF SQLCODE = 100 THEN /* NO EMPLOYEES FOUND */
    DO; /* PRINT ERROR MESSAGE */
        MODNAME = 'DSN8IPNO';
        CALL DSN8MPG (MODULE, '008I', OUTMSG);
        ERROR = OUTMSG;
        CALL P3_SEND;
        GOTO P3_SELECT_40;
    END;

    CALL P3_PREPARE_SCREEN; /* LIST FIRST EMPLOYEE */

P3_SELECT_30:
    EXEC SQL FETCH TELE2 /* GET NEXT RECORD */
        INTO :PPHONE;
    IF SQLCODE = 100 THEN /* FINISHED? */
        DO;
            ERROR = '';
            CALL P3_SEND;
            GOTO P3_SELECT_40;
        END;
    CALL P3_PREPARE_SCREEN; /* LIST EMPLOYEE */
    GOTO P3_SELECT_30; /* CONTINUE */

P3_SELECT_40:
    EXEC SQL CLOSE TELE2; /* CLOSE CURSOR */
    GOTO P3_END;
END; /* END IF */

ELSE DO; /* NO - SEARCH ON LAST NAME*/
    /* AND Optionally FIRST NAME*/
    INPUT_1.FNAME = TRANSLATE(INPUT_1.FNAME,'%',' ');
    EXEC SQL OPEN TELE3; /* OPEN CURSOR */
    EXEC SQL FETCH TELE3 /* GET FIRST RECORD */
        INTO :PPHONE;
    IF SQLCODE = 100 THEN /* EMPLOYEE NOT FOUND */
        DO; /* PRINT ERROR MESSAGE */
            MODNAME = 'DSN8IPNO';
            CALL DSN8MPG (MODULE, '008I', OUTMSG);
            ERROR = OUTMSG;
            CALL P3_SEND;
            GOTO P3_SELECT_60;
        END;
    CALL P3_PREPARE_SCREEN; /* LIST FIRST EMPLOYEE */

P3_SELECT_50:
    EXEC SQL FETCH TELE3 /* GET NEXT RECORD */
        INTO :PPHONE;
    IF SQLCODE = 100 THEN /* FINISHED? */
        DO;
            ERROR = '';
            CALL P3_SEND;
            GOTO P3_SELECT_60;
        END;
    CALL P3_PREPARE_SCREEN; /* LIST EMPLOYEE */
    GOTO P3_SELECT_50; /* CONTINUE */

P3_SELECT_60:
    EXEC SQL CLOSE TELE3; /* CLOSE CURSOR */
    GOTO P3_SELECT_50; /* CONTINUE */
END; /* END IF */
END; /* END WHEN */

EXEC SQL WHENEVER SQLERROR CONTINUE;
EXEC SQL WHENEVER SQLWARNING CONTINUE;

/* UPDATE PHONE NUMBERS FOR EMPLOYEES */
EXEC SQL WHENEVER SQLERROR CONTINUE;
EXEC SQL WHENEVER SQLWARNING CONTINUE;

WHEN ('U') DO; /* TELEPHONE UPDATE */
/* FIND WHICH NUMBERS HAVE BEEN UPDATED */
ERROR = '';                       /* SET IN CASE NO UPDATES */
0       DO I = 1 TO 15;
    IF INPUT_2.NEWNO(I) = ' ' THEN; /* NO UPDATE ON THIS LINE */
      ELSE DO;                        /* PERFORM UPDATE */
        EMPLOYEE_NO = INPUT_2.ENO(I);
        PHONE_NO = INPUT_2.NEWNO(I);
        EXEC SQL UPDATE VEMPLP
          SET PHONENUMBER = :PHONE_NO
          WHERE EMPLOYEENUMBER = :EMPLOYEE_NO;
      /* UPDATE SUCCESSFUL */
      /* PRINT CONFIRMATION */
      /* MESSAGE */
    END;                                     /*   END IF       */
    END;                                       /*   END WHEN     */
  END;
  CALL P3_SEND;
END;
0     OTHERWISE                            /* UNFORMATTED SCREEN */
  DO;
    OUT_MESSAGE.LL = 110;
    MODNAME        = 'DSN8IPNO';
    CALL P3_SEND;
  END;
END;                                   /* END SELECT */
0   GOTO P3_END;
1/*********************************************************************/
/*        SQL ERROR HANDLING */
1P3_DBERROR:
  CALL DSN8MPG (MODULE, '060E', OUTMSG);
  CHAR_SQLCODE = SQLCODE;
  ERROR = OUTMSG||CHAR_SQLCOD;
  PUT DATA (ERROR,SQLWARN0);     /*PRINT ERROR MESSAGE */
  GOTO P3_DBERROR2;
1P3_DBERROR2:
0   CALL PLITDLI (THREE,CHNG_FKT,ALTPCB,IOLTERM);
0   IF ALTPCB.STC_CODE ^= ' ' THEN;     /* PERFORM ROLLBACK */
      ELSE CALL PLITDLI (FOUR,ISRT_FKT,ALTPCB,OUT_MESSAGE,MODNAME);
0         CALL PLITDLI (ONE,ROLL_FKT);
0   GOTO P3_END;
1/*********************************************************************/
/*        PRINT INFORMATION ON SCREEN */
1P3_PREPARE_SCREEN:
  PROC;
    IF ITAB = 15 THEN                  /* PRINT SCROLLING MESSAGE */
      DO;
        CALL DSN8MPG (MODULE, '058I', OUTMSG);
        ERROR = OUTMSG;
        CALL P3_SEND;
        ITAB = 0;                       /* INITIALIZE COUNTER */
        OUT_MESSAGE.LL = 83;            /* INITIALIZE LINE LENGTH */
      END;
      ITAB = ITAB + 1;                 /* INCREMENT COUNTER */
      OUTPUT_2 (ITAB) = PPHONE , BY NAME;
    END;
  RETURN;
END P3_Prepare_Screen;
1P3_Send:
PROC;
    IF FIRST THEN
        DO;
            CALL PLITDLI (FOUR,ISRT_FKT,IOPCB,OUT_MESSAGE,MODNAME);
            FIRST = '0'B;
        END;
    ELSE CALL PLITDLI (THREE,ISRT_FKT,IOPCB,OUT_MESSAGE);
    IF IOPCB.STC_CODE = ' ' THEN RETURN;
    /* INVALID DL/I STC-CODE ON ISRT MSG*/
    /* PRINT ERROR MESSAGE */
    CALL DSN8MPG (MODULE, '065E', OUTMSG);
    ERROR = OUTMSG||IOPCB.STC_CODE;
    CALL PLITDLI (THREE,CHNG_FKT,ALTPCB,IOLTERM);
    IF ALTPCB.STC_CODE ^= ' ' THEN; /* PERFORM ROLLBACK */
        ELSE CALL PLITDLI (FOUR,ISRT_FKT,ALTPCB,OUT_MESSAGE,MODNAME);
        CALL PLITDLI (ONE,ROLL_FKT);
        RETURN;
    /*********************************************************************************/
END P3_Send;                                 /* END OF PROGRAM */
END DSN8IP3;

Related reference
“Sample applications in IMS” on page 1346
A set of Db2 sample applications run in the IMS environment.

DSNTEJ4C

THIS JCL PERFORMS THE PHASE 4 SETUP FOR THE SAMPLE APPLICATIONS AT SITES WITH COBOL.

//******************************************************************************
/* NAME = DSNTEJ4C
/* DESCRIPTIVE NAME = DB2 SAMPLE APPLICATION
/* PHASE 4
/* COBOL, IMS
/* LICENSED MATERIALS - PROPERTY OF IBM
/* 5650-DB2
/* (C) COPYRIGHT 1982, 2016 IBM CORP. ALL RIGHTS RESERVED.
/* STATUS = VERSION 12
/* FUNCTION = THIS JCL PERFORMS THE PHASE 4 SETUP FOR THE SAMPLE APPLICATIONS AT SITES WITH COBOL. IT PREPARES THE COBOL IMS PROGRAM.
/* CHANGE ACTIVITY = 08/18/2014 Single-phase migration s21938_inst1 s21938
/*******************************************************************************
/* JOBLIB DD DSN=DSN!!0.DSNLOAD,DISP=SHR
/* STEP 1: PREPARE SQL 0 PART OF PROGRAM
/* PH04CS01 EXEC DSNHICOB,
/* PARM PC=('HOST(IBMCOB)',APOST,APOSTSQL,SOURCE,NOXREF,'SQL(DB2)','DEC(31)'),
/* PARM.COB=('NOSEQUENCE,QUOTE,RENT,PGMNAME(LONGUPPER)'),
/* PARM.LKED='XREF,NCAL',
/* MEM=DSN8IC0
/* PC.DBRMLIB DD DSN=DSN!!0.DBRMLIB.DATA(DSN8IC0),
/* DISP=SHR
/* PC.SYSCIN DD DSN=&&DSNHOUT0
/* PC.SYSLIB DD DSN=DSN!!0.SRCLIB.DATA,
/* DISP=SHR
/* DD DSN=DSN!!0.SDSNSAMP,
/* DISP=SHR

1386  Db2 12 for z/OS: Application Programming and SQL Guide
/*        STEP  2: PREPARE SQL 1 PART OF PROGRAM */
PH04CS02 EXEC DSNHICOB,
  PARM.PC=('HOST(IBMCOB)',APOST,APOSTSQL,SOURCE,
          NOXREF,'SQL(BB2)', 'DEC(31)'),
  PARM.COB=('NOSEQUENCE,QUOTE,RENT,'PGMNAME(LONGUPPER)'),
  PARM.LKED='XREF,NCAL',
  COND=(4,LT),
  MEM=DSN8IC1
/*        STEP  3: PREPARE SQL 2 PART OF PROGRAM */
PH04CS03 EXEC DSNHICOB,
  PARM.PC=('HOST(IBMCOB)',APOST,APOSTSQL,SOURCE,
          NOXREF,'SQL(BB2)', 'DEC(31)'),
  PARM.COB=('NOSEQUENCE,QUOTE,RENT,'PGMNAME(LONGUPPER)'),
  PARM.LKED='XREF,NCAL',
  COND=(4,LT),
  MEM=DSN8IC2
/*        STEP  4: LINKEDIT THE ENTIRE PROGRAM */
PH04CS04 EXEC PGM=IEWL,PARM='LIST,XREF,LET',COND=(4,LT)
  SYSLIB   DD DISP=SHR,DSN=CEE.V!R!M!.SCEELKED
  DD DISP=SHR,DSN=SDSNSAMP
  DD DISP=SHR,DSN=IMSVS.RESLIB
  SYSLIN   DD DDNAME=SYSIN
  SYSLMOD  DD DISP=SHR,DSN=SRO10.RUNLIB.LOAD
  SYSPRINT DD SYSOUT=* 
  SYSUDUMP DD SYSOUT=* 
  SYSTSPRT DD SYSOUT=* 
  SYSPRINT DD SYSOUT=* 
  SYSIN   DD *
  INCLUDE SYSLIB(DFSLI000)
  INCLUDE SYSLMOD(DSN8IC0)
  INCLUDE SYSLMOD(DSN8IC1)
  INCLUDE SYSLMOD(DSN8IC2)
  INCLUDE SYSLMOD(DSN8MCG)
  ENTRY   DLITCBL
  NAME    DSN8IC0(R)
/*        STEP  5: BIND THE PLAN */
PH04CS05 EXEC PGM=IKJEFT01,DYNAMNBR=20,COND=(4,LT)
  DBRMLIB  DD DSN=SRO10.RUNLIB.LOAD,DISP=SHR
  SYSDUMP DD SYSOUT=* 
  SYSTSFRRT DD SYSOUT=* 
  SYSPRINT DD SYSOUT=* 
  SYIN    DD *
  SET CURRENT SQLID = 'SYSADM';
  GRANT BIND, EXECUTE ON PLAN DSN8IC0 TO PUBLIC;
  SYSSIN   DD *
  DSYSTEM(DSN)
  BIND PACKAGE (DSN8IC!!) MEMBER(DSN8IC1) APPLCOMPAT(V!!R1) +
  ACT(REP)  ISO(CS) CURRENTDATA(YES) ENCODING(EBCDIC)
  BIND PACKAGE (DSN8IC!!) MEMBER(DSN8IC2) APPLCOMPAT(V!!R1) +
Related reference
“Sample applications in IMS” on page 1346
A set of Db2 sample applications run in the IMS environment.

**DSNTEJ4P**

THIS JCL PERFORMS THE PHASE 4 SETUP FOR THE SAMPLE APPLICATIONS AT SITES WITH PL/I.
Chapter 12. Sample data and applications supplied with Db2 for z/OS 1389
/*        STEP  6: PREPARE SQL 1 PART OF PROGRAM */
PH04PS06 EXEC DSNHPLI,MEM=DSN8IP7,
       COND=(4,LT),
       PARM.PPLI='MACRO,NOSYNTAX,MDECK,NOINSOURCE,NOSOURCE',
       PARM.PC='HOST(PLI),CCSID(37),STDSQL(NO)',
       PARM.PL='(NOPT,SOURCE,OBJECT,MARGINS(2,72,0),NORENT',
       'LIMITS(EXTNAME(7)),OPTIONS','SYSTEM(IMS)'),
       PARM.LKED='NCAL'
/PPLI.SYSIN DD DSN=DSN!!0.SDSNSAMP(DSN8IP7),
       DISP=SHR
/PC.DBRMLIB DD DSN=DSN!!0.DBRMLIB.DATA(DSN8IP7),
       DISP=SHR
/PC.SYSCIN DD DSN=&&DSNHOUT7
/PC.SYSLIB DD DSN=DSN!!0.SRCLIB.DATA,
       DISP=SHR
       DD DSN=DSN!!0.SDSNSAMP,
       DISP=SHR
/PLI.SYSIN DD DSN=&&DSNHOUT7
/LKED.SYSLMOD DD DSN=DSN!!0.RUNLIB.LOAD(DSN8IP7),
       DISP=SHR
/*
*        STEP  7: PREPARE SQL 2 PART OF PROGRAM */
PH04PS07 EXEC DSNHPLI,MEM=DSN8IP8,
       COND=(4,LT),
       PARM.PPLI='MACRO,NOSYNTAX,MDECK,NOINSOURCE,NOSOURCE',
       PARM.PC='HOST(PLI),CCSID(37),STDSQL(NO)',
       PARM.PL='(NOPT,SOURCE,OBJECT,MARGINS(2,72,0),NORENT',
       'LIMITS(EXTNAME(7)),OPTIONS','SYSTEM(IMS)'),
       PARM.LKED='NCAL'
/PPLI.SYSIN DD DSN=DSN!!0.SDSNSAMP(DSN8IP8),
       DISP=SHR
/PC.DBRMLIB DD DSN=DSN!!0.DBRMLIB.DATA(DSN8IP8),
       DISP=SHR
/PC.SYSCIN DD DSN=&&DSNHOUT8
/PC.SYSLIB DD DSN=DSN!!0.SRCLIB.DATA,
       DISP=SHR
       DD DSN=DSN!!0.SDSNSAMP,
       DISP=SHR
/PLI.SYSIN DD DSN=&&DSNHOUT8
/LKED.SYSLMOD DD DSN=DSN!!0.RUNLIB.LOAD(DSN8IP8),
       DISP=SHR
/*
*        STEP  8: LINKEDIT PROGRAM TOGETHER */
PH04PS08 EXEC PGM=IEWL,PARM='LIST,XREF,LET',
       COND=(4,LT)
/SYSLIB DD DISP=SHR,DSN=CEE.V!R!M!.SCEELKED
       DD DISP=SHR,DSN=DSN!!0.SDSNLOAD
/SYSLIN DD DDNAME=SYSIN
/SYSLMOD DD DISP=SHR,DSN=DSN!!0.RUNLIB.LOAD
/SYSPRINT DD SYSOUT=*  
/SYSDUMPP DD SYSOUT=*  
/SYSDUMP DD SYSOUT=*  
/SYSUT1 DD UNIT=SYSDA,SPACE=(1024,(50,50))
/SYSIN DD *
INCLUDE SYSLIB(DFSLI000)
INCLUDE SYSLMOD(DSN8IP0)
INCLUDE SYSLMOD(DSN8IP1)
INCLUDE SYSLMOD(DSN8IP2)
INCLUDE SYSLMOD(DSN8MPG)
ENTRY CEESTART
NAME DSN8IP0(R)
INCLUDE SYSLIB(DFSLI000)
INCLUDE SYSLMOD(DSN8IP3)
INCLUDE SYSLMOD(DSN8MPG)
ENTRY CEESTART
NAME DSN8IP0(R)
INCLUDE SYSLIB(DFSLI000)
INCLUDE SYSLMOD(DSN8IP6)
INCLUDE SYSLMOD(DSN8IP7)
INCLUDE SYSLMOD(DSN8IP8)
INCLUDE SYSLMOD(DSN8MPG)
ENTRY  CEESTART
NAME  DSN8IQ0(R)
/**     STEP  9: BIND PROGRAMS
**//DYNSPS9  EXEC PGM=IKJEFT01,DYNNAMBR=20,COND=(4,LT)
//DBRMLIB  DD DISP=SHR,DSN=DSN!0.DBRMLIB.DATA
//SYSUDUMP DD SYSOUT=*  
//SYSSTSPRT DD SYSOUT=*  
//SYSPRINT DD SYSOUT=*  
//SYSIN   DD *
SET CURRENT SQLID = 'SYSADM';
GRANT BIND, EXECUTE ON PLAN DSN8IP0, DSN8IQ0, DSN8IH0
TO PUBLIC;
//SYSTSIN DD *
DSN SYSTEM(DSN)
BIND PACKAGE(DSN8IP!!) MEMBER(DSN8IP1) APPLCOMPAT(V!!R1) +
  ACT(REP) ISO(CS) CURRENTDATA(YES) ENCODING(EBCDIC)
BIND PACKAGE(DSN8IP!!) MEMBER(DSN8IP2) APPLCOMPAT(V!!R1) +
  ACT(REP) ISO(CS) CURRENTDATA(YES) ENCODING(EBCDIC)
BIND PACKAGE(DSN8IP!!) MEMBER(DSN8IP3) APPLCOMPAT(V!!R1) +
  ACT(REP) ISO(CS) CURRENTDATA(YES) ENCODING(EBCDIC)
BIND PACKAGE(DSN8IP!!) MEMBER(DSN8IP7) APPLCOMPAT(V!!R1) +
  ACT(REP) ISO(CS) CURRENTDATA(YES) ENCODING(EBCDIC)
BIND PACKAGE(DSN8IP!!) MEMBER(DSN8IP8) APPLCOMPAT(V!!R1) +
  ACT(REP) ISO(CS) CURRENTDATA(YES) ENCODING(EBCDIC)
BIND PLAN(DSN8IP0) +
  PKLIST(DSN8IP!!,DSN8IP1, DSN8IP!!,DSN8IP2) +
  ACTION(REPLACE) RETAIN +
  ISO(CS) CURRENTDATA(YES) ENCODING(EBCDIC)
BIND PLAN(DSN8IP0) +
  PKLIST(DSN8IP!!,DSN8IP7, DSN8IP!!,DSN8IP8) +
  ACTION(REPLACE) RETAIN +
  ISO(CS) CURRENTDATA(YES) ENCODING(EBCDIC)
BIND PLAN(DSN8IH0)  PKLIST(DSN8IP!!,DSN8IP3) +
  ACTION(REPLACE) RETAIN +
  ISO(CS) CURRENTDATA(YES) ENCODING(EBCDIC)
RUN  PROGRAM(DSNTIAD) PLAN(DSNTIA!!) -
  LIB('DSN!0.RUNLIB.LOAD')
END
/**     STEP 10: CREATE MFS MAPS FOR ORGANIZATION APPLICATION
//PH04PS10 EXEC MFSUTL,COND=(4,LT)
//SYSIN DD DSN=DSN!0.SDSNSAMP(DSN8IPG),
//      DISP=SHR
/**     STEP 11: CREATE MFS MAPS FOR ORGANIZATION APPLICATION
//PH04PS11 EXEC MFSUTL,COND=(4,LT)
//SYSIN DD DSN=DSN!0.SDSNSAMP(DSN8IPD),
//      DISP=SHR
/**     STEP 12: CREATE MFS MAPS FOR PROJECT APPLICATION
//PH04PS12 EXEC MFSUTL,COND=(4,LT)
//SYSIN DD DSN=DSN!0.SDSNSAMP(DSN8IPF),
//      DISP=SHR
/**     STEP 13: CREATE MFS MAPS FOR PROJECT APPLICATION
//PH04PS13 EXEC MFSUTL,COND=(4,LT)
//SYSIN DD DSN=DSN!0.SDSNSAMP(DSN8IPE),
//      DISP=SHR
/**     STEP 14: CREATE MFS MAPS FOR TELEPHONE APPLICATION
//PH04PS14 EXEC MFSUTL,COND=(4,LT)
//SYSIN DD DSN=DSN!0.SDSNSAMP(DSN8IPL),
//      DISP=SHR
/**     STEP 15: CREATE MFS MAPS FOR TELEPHONE APPLICATION
//PH04PS15 EXEC MFSUTL,COND=(4,LT)
//SYSIN DD DSN=DSN!0.SDSNSAMP(DSN8IPN),
//      DISP=SHR
/**     STEP 16: RUN PSBGEN
//PH04PS16 EXEC PSBGEN,MBR=DSN8IP0,COND=(4,LT)
//C.SYSIN DD *
PRINT NOGEN
PCB  TYPE=TP,EXPRESS=YES,ALTRESP=YES,MODIFY=YES,SAMETRM=YES
PSBGEN PSBNAME=DSN8IP0,LANG=PLI
END
/**     STEP 17: RUN ACBGEN
//PH04PS17 EXEC ACBGEN,COND=(4,LT)
//G.SYSIN DD *
BUILD PSB=DSN8IP0
Related reference

“Sample applications in IMS” on page 1346
A set of Db2 sample applications run in the IMS environment.

Sample applications in CICS

A set of Db2 sample applications run in the CICS environment.

Table 184. Sample Db2 applications for CICS

<table>
<thead>
<tr>
<th>Application</th>
<th>Program name</th>
<th>JCL member name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Organization</td>
<td>DSN8CC0</td>
<td>DSNTEJ5C</td>
<td>CICS COBOL Organization Application</td>
</tr>
<tr>
<td></td>
<td>DSN8CC1</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>DSN8CC2</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

| Organization | DSN8CP0      | DSNTEJ5P        | CICS PL/I Organization Application |
|             | DSN8CP1      |                 |             |
|             | DSN8CP2      |                 |             |

| Project     | DSN8CP6      | DSNTEJ5P        | CICS PL/I Project Application |
|            | DSN8CP7      |                 |             |
|            | DSN8CP8      |                 |             |

| Phone       | DSN8CP3      | DSNTEJ5P        | CICS PL/I Phone Application. This program lists employee telephone numbers and updates them if requested. |

Related reference

Data sets that the precompiler uses
When you invoke the precompiler you need to provide data sets that contain input for the precompiler, such as the host programming statements and SQL statements. You also need to provide data sets where the precompiler can store its output, such as the modified source code and diagnostics messages.

**DSN8CC0**

This module issues CICS receive map to retrieve input, calls DSN8CC1, and issue CICS send map after returning.

```
IDENTIFICATION DIVISION.
* ------------------------
PROGRAM-ID. DSN8CC0.

***** DSN8CC0 - SUBSYSTEM INTERFACE MODULE FOR CICS/VS - COBOL ***
*  
* MODULE NAME = DSN8CC0
*  
* DESCRIPTIVE NAME = DB2 SAMPLE APPLICATION
*  
* SUBSYSTEM INTERFACE MODULE
*  
* CICS
*  
* COBOL
*  
* Licensed Materials - Property of IBM
*  
* 5605-DB2
*  
*(C) COPYRIGHT 1982, 2010 IBM Corp. All Rights Reserved.
*  
* STATUS = Version 10
*  
* FUNCTION = This module issues CICS receive map to retrieve input, calls DSN8CC1, and issue CICS send map after returning.
*  
* NOTES =
*  
*  1. This is a CICS pseudo conversation program which initializes itself when a terminal operator enters input after viewing the screen sent by previous iterations of the program.
*  
* DEPENDENCIES = two CICS maps(DSECTS) are required:
*  
* DSN8MCMD AND DSN8MCMD
*  
* module DSN8CC1 is required.
*  
* DCLGEN structure DSN8MCCS is required
*  
* included COBOL structure DSN8MCCA is required.
*  
* RESTRICTIONS = none
*  
* MODULE TYPE =
*  
* PROCESSOR = DB2 PRECOMPILER, CICS TRANSLATOR, COBOL COMPIL
*  
* MODULE SIZE = SEE LINK-EDIT
*  
* ATTRIBUTES = REUSABLE
*  
* ENTRY POINT = DSN8CC0
*  
* PURPOSE = SEE FUNCTION
*  
* LINKAGE = CICS/OS/VS ENTRY
*  
* INPUT = parameters explicitly passed to this function:
*  
* SYMBOLIC LABEL/NAME = DSN8CCGI
* DESCRIPTION = CICS/OS/VS BMS MAP FOR GENERAL INPUT
*  
* SYMBOLIC LABEL/NAME = DSN8CCDI
* DESCRIPTION = CICS/OS/VS BMS MAP FOR DETAIL INPUT
*  
* OUTPUT = parameters explicitly returned:
*  
* SYMBOLIC LABEL/NAME = DSN8CCGO
* DESCRIPTION = CICS/OS/VS BMS MAP FOR GENERAL OUTPUT
*  
* SYMBOLIC LABEL/NAME = DSN8CCDO
* DESCRIPTION = CICS/OS/VS BMS MAP FOR DETAIL OUTPUT
*  
* EXIT-NORMAL = CICS RETURN TRANSID
*  
* EXIT-ERROR = SQL ERROR FOR SQL ERRORS
*  
* CICS ABEND FOR CICS PROBLEMS
```
* RETURN CODE = NONE
* ABEND CODES = LSCR - LOGICAL SCREEN SET INCORRECTLY
* ERROR-MESSAGES = NONE
* EXTERNAL REFERENCES = COMMON CICS REQUIREMENTS
* ROUTINES/SERVICES = CICS/VS SERVICES
* DSN8CC1 - SQL 1 MAINLINE CODE
* DATA-AREAS =
* DSN8MCCA - PARAMETER TO BE PASSED TO DSN8CC1
* COMMON AREA
* DSN8MCCS - DECLARE CONVERSATION STATUS
* DSN8MCC2 - COMMON AREA PART 2
* DSN8MCMD - CICS/OS/VS COBOL MAP, ORGANIZATION
* DSN8MCMG - CICS/OS/VS COBOL MAP, ORGANIZATION
* CONTROL-BLOCKS = SQLCA - SQL COMMUNICATION AREA
* TABLES = NONE
* CHANGE-ACTIVITY = 10/18/2005 PK03311 INITIALIZE UNINITIALIZED STORAGE 001
* PSEUDOCODE*

PROCEDURE DECLARATIONS.
ALLOCATE COBOL WORK AREA FOR COMMAREA.
PUT MODULE NAME 'DSN8CC0' IN AREA USED BY ERROR-HANDLER.
PUT CICS EIBTRMID IN PCONVSTA.CONVID TO BE PASSED TO DSN8CC1.
RETRIEVE LASTCR FROM VCONA USING THE CONVID TO DETERMINE WHICH OF THE TWO BMS MAPS SHOULD BE USED TO MAP IN DATA.

IF RETRIEVAL OF MAPS IS SUCCESSFUL, THEN DO;
EXEC CICS RECEIVE MAP ACCORDING TO SPECIFIED LASTSCR
IF MAPFAIL CONDITION IS RAISED* THEN DO;
COMPARM.PFKIN = '00'
GO TO CC0SEND
END
ELSE
PUT DATA FROM MAP INTO COMPARM **
ELSE
IT IS A NEW CONVERSATION
AND NO EXEC CICS RECEIVE MAP IS ISSUED.
CC0SEND:
EXEC CICS LINK PROGRAM('DSN8CC1') COMMAREA(COMMAREA).
UPON RETURN FROM DSN8CC1, EXEC CICS SEND MAP ACCORDING TO THE TYPE SPECIFIED IN PCONVSTA.LASTSCR.
EXEC CICS RETURN TRANSID(DBCS).
END.
I.E. LAST CONVERSATION EXISTS, BUT OPERATOR HAD ENTERED DATA FROM A CLEARED SCREEN OR HAD ERASED ALL DATA ON A FORMATTED SCREEN AND Pressed ENTER
** COMPARM.PFKIN = PF KEY ACTUALLY USED I.E. '01' FOR PF1 **

ENVIRONMENT DIVISION.

DATA DIVISION.

WORKING-STORAGE SECTION.
77 FOUND PIC S99.
EXEC SQL INCLUDE SQLCA END-EXEC.
EXEC SQL INCLUDE DSN8MCC2 END-EXEC.
01 COMMAREA.
   EXEC SQL INCLUDE DSNBMCCA END-EXEC.
EXEC SQL INCLUDE DSNBMCCS END-EXEC.
EXEC SQL INCLUDE DSNBMCMG END-EXEC.
EXEC SQL INCLUDE DSNBMCMD END-EXEC.
*****************************************************************
* MAPD REDEFINES THE COBOL STRUCTURE ASSOCIATED WITH THE
* CICS MAP DSN8CCD.
*****************************************************************
01 MAPD REDEFINES DSN8CCDI.
   02 FILLER PIC X(387).
   02 SUBMAP OCCURS 15 TIMES.
       03 COL1LEN PIC S9(4) COMP.
       03 COL1ATTR PIC X(1).
       03 COL1DATA PIC X(37).
       03 COL2LEN PIC S9(4) COMP.
       03 COL2ATTR PIC X(1).
       03 COL2DATA PIC X(40).
*****************************************************************
* PFKEYS IS AN ARRAY OF 24 ELEMENTS REPRESENTING THE DIFFERENT
* PFKEYS AS THEY WOULD BE REPRESENTED IN EIBAID.
*****************************************************************
01 PFKEYS-DUMB.
   02 PFKEYS-ALL PIC X(24) VALUE '123456789:#@ABCDEFGHI>.<'.
   02 PFKEYS REDEFINES PFKEYS-ALL PIC X(1) OCCURS 24 TIMES.
*****************************************************************
* PFK IS AN ARRAY OF 12 TWO-BYTE CHARS REPRESENTING THE PFKEYS
* ALLOWED AS INPUT TO DSN8CC1 AND DSN8CC2 ETC.
*****************************************************************
01 PFK-DUMB.
   02 PFK-ALL PIC X(24) VALUE '010203040506070809101112'.
   02 PFK REDEFINES PFK-ALL PIC X(2) OCCURS 12 TIMES.

PROCEDURE DIVISION.
*------------------
*****************************************************************
* SQL RETURN CODE HANDLING
*****************************************************************
EXEC SQL WHENEVER SQLERROR GO TO DB-ERROR END-EXEC
EXEC SQL WHENEVER SQLWARNING GO TO DB-ERROR END-EXEC.
*****************************************************************
* ALLOCATE COBOL WORK AREA / INITIALIZE VARIABLES
*****************************************************************
* INIT AREA INCLUDED BY DSNBMCCA
    MOVE SPACES TO COMMAREA.
* INIT AREA INCLUDED BY DSNBMCCS
    MOVE SPACES TO PCONA.
* INIT AREA INCLUDED BY DSNBMCMG
    MOVE SPACES TO COMMAREA.
    MOVE SPACES TO PCONA.
    MOVE ZEROES TO ATITLEL OF DSNBCCGI.
    MOVE ZEROES TO ATITLEI OF DSNBCCGI.
    MOVE ZEROES TO AMAJSYSL OF DSNBCCGI.
    MOVE ZEROES TO AMAJSYSI OF DSNBCCGI.
    MOVE ZEROES TO AACTIONL OF DSNBCCGI.
    MOVE ZEROES TO AACTIONI OF DSNBCCGI.
    MOVE ZEROES TO ADESCL2L OF DSNBCCGI.
    MOVE ZEROES TO ADESCL2I OF DSNBCCGI.
    MOVE ZEROES TO AOBJJCL OF DSNBCCGI.
    MOVE ZEROES TO AOBJJCTI OF DSNBCCGI.
    MOVE ZEROES TO ADESCL3L OF DSNBCCGI.
    MOVE ZEROES TO ADESCL3I OF DSNBCCGI.
    MOVE ZEROES TO ASEARCHL OF DSNBCCGI.
    MOVE ZEROES TO ASEARCHI OF DSNBCCGI.
    MOVE ZEROES TO ADESCL4L OF DSNBCCGI.
    MOVE ZEROES TO ADESCL4I OF DSNBCCGI.
    MOVE ZEROES TO ADATEAL OF DSNBCCGI.
    MOVE ZEROES TO ADATEAI OF DSNBCCGI.
    MOVE ZEROES TO AMSGL OF DSNBCCGI.
    MOVE ZEROES TO AMSGI OF DSNBCCGI.
    MOVE ZEROES TO ALINEL(1).
    MOVE ZEROES TO ALINEI(1).
    MOVE ZEROES TO ALINEL(2).
    MOVE ZEROES TO ALINEI(2).
    MOVE ZEROES TO ALINEL(3).
    MOVE ZEROES TO ALINEI(3).
    MOVE ZEROES TO ALINEL(4).
    MOVE ZEROES TO ALINEI(4).
    MOVE ZEROES TO ALINEL(5).
    MOVE ZEROES TO ALINEI(5).
    MOVE ZEROES TO ALINEL(6).
    MOVE ZEROES TO ALINEI(6).

MOVE ZEROES TO ALINEL(7).
MOVE SPACES TO ALINEI(7).
MOVE ZEROES TO ALINEL(8).
MOVE SPACES TO ALINEI(8).
MOVE ZEROES TO ALINEL(9).
MOVE SPACES TO ALINEI(9).
MOVE ZEROES TO ALINEL(10).
MOVE SPACES TO ALINEI(10).
MOVE ZEROES TO ALINEL(11).
MOVE SPACES TO ALINEI(11).
MOVE ZEROES TO ALINEL(12).
MOVE SPACES TO ALINEI(12).
MOVE ZEROES TO ALINEL(13).
MOVE SPACES TO ALINEI(13).
MOVE ZEROES TO ALINEL(14).
MOVE SPACES TO ALINEI(14).
MOVE ZEROES TO ALINEL(15).
MOVE SPACES TO ALINEI(15).

MOVE ZEROES TO APFKEYL OF DSN8CCGI.
MOVE SPACES TO APFKEYI OF DSN8CCGI.

* INIT AREA INCLUDED BY DSN8MCMD
MOVE ZEROES TO BTITLEL OF DSN8CCDI.
MOVE SPACES TO BTITLEI OF DSN8CCDI.
MOVE ZEROES TO BMJALSYSL OF DSN8CCDI.
MOVE SPACES TO BMJALSYSI OF DSN8CCDI.
MOVE ZEROES TO BACTIONL OF DSN8CCDI.
MOVE SPACES TO BACTIONI OF DSN8CCDI.
MOVE ZEROES TO BDESCL2L OF DSN8CCDI.
MOVE SPACES TO BDESCL2I OF DSN8CCDI.
MOVE ZEROES TO BDESCL3L OF DSN8CCDI.
MOVE SPACES TO BDESCL3I OF DSN8CCDI.
MOVE ZEROES TO BDESCL4L OF DSN8CCDI.
MOVE SPACES TO BDESCL4I OF DSN8CCDI.
MOVE ZEROES TO BDATALL OF DSN8CCDI.
MOVE SPACES TO BDATALL OF DSN8CCDI.
MOVE ZEROES TO BMSGL OF DSN8CCDI.
MOVE SPACES TO BMSGL OF DSN8CCDI.
MOVE ZEROES TO LINE1F1L OF DSN8CCDI.
MOVE SPACES TO LINE1F1L OF DSN8CCDI.
MOVE ZEROES TO LINE1F2L OF DSN8CCDI.
MOVE SPACES TO LINE1F2L OF DSN8CCDI.
MOVE ZEROES TO LINE2F1L OF DSN8CCDI.
MOVE SPACES TO LINE2F1L OF DSN8CCDI.
MOVE ZEROES TO LINE2F2L OF DSN8CCDI.
MOVE SPACES TO LINE2F2L OF DSN8CCDI.
MOVE ZEROES TO LINE3F1L OF DSN8CCDI.
MOVE SPACES TO LINE3F1L OF DSN8CCDI.
MOVE ZEROES TO LINE3F2L OF DSN8CCDI.
MOVE SPACES TO LINE3F2L OF DSN8CCDI.
MOVE ZEROES TO LINE4F1L OF DSN8CCDI.
MOVE SPACES TO LINE4F1L OF DSN8CCDI.
MOVE ZEROES TO LINE4F2L OF DSN8CCDI.
MOVE SPACES TO LINE4F2L OF DSN8CCDI.
MOVE ZEROES TO LINE5F1L OF DSN8CCDI.
MOVE SPACES TO LINE5F1L OF DSN8CCDI.
MOVE ZEROES TO LINE5F2L OF DSN8CCDI.
MOVE SPACES TO LINE5F2L OF DSN8CCDI.
MOVE ZEROES TO LINE6F1L OF DSN8CCDI.
MOVE SPACES TO LINE6F1L OF DSN8CCDI.
MOVE ZEROES TO LINE6F2L OF DSN8CCDI.
MOVE SPACES TO LINE6F2L OF DSN8CCDI.
MOVE ZEROES TO LINE7F1L OF DSN8CCDI.
MOVE SPACES TO LINE7F1L OF DSN8CCDI.
MOVE ZEROES TO LINE7F2L OF DSN8CCDI.
MOVE SPACES TO LINE7F2L OF DSN8CCDI.
MOVE ZEROES TO LINE8F1L OF DSN8CCDI.
MOVE SPACES TO LINE8F1L OF DSN8CCDI.
MOVE ZEROES TO LINE8F2L OF DSN8CCDI.
MOVE SPACES TO LINE8F2L OF DSN8CCDI.
MOVE ZEROES TO LINE9F1L OF DSN8CCDI.
MOVE SPACES TO LINE9F1L OF DSN8CCDI.
MOVE ZEROES TO LINE9F2L OF DSN8CCDI.
MOVE SPACES TO LINE9F2L OF DSN8CCDI.
MOVE ZEROES TO LINE9F3L OF DSN8CCDI.
MOVE SPACES TO LINE9F3L OF DSN8CCDI.
MOVE ZEROES TO LINEAFL1L OF DSN8CCDI.
MOVE SPACES TO LINEAFL1L OF DSN8CCDI.
MOVE ZEROES TO LINEAFL2L OF DSN8CCDI.
MOVE SPACES TO LINEAFL2L OF DSN8CCDI.
MOVE ZEROES TO LINEAFL3L OF DSN8CCDI.
MOVE SPACES TO LINEAFL3L OF DSN8CCDI.
MOVE ZEROES TO LINEAFL4L OF DSN8CCDI.
MOVE SPACES TO LINEAFL4L OF DSN8CCDI.
MOVE ZEROES TO LINEAFL5L OF DSN8CCDI.
MOVE SPACES TO LINEAFL5L OF DSN8CCDI.
MOVE ZEROES TO LINEAFL6L OF DSN8CCDI.
MOVE SPACES TO LINEAFL6L OF DSN8CCDI.
MOVE ZEROES TO LINEAFL7L OF DSN8CCDI.
MOVE SPACES TO LINEAFL7L OF DSN8CCDI.
MOVE ZEROES TO LINEAFL8L OF DSN8CCDI.
MOVE SPACES TO LINEAFL8L OF DSN8CCDI.
MOVE ZEROES TO LINEAFL9L OF DSN8CCDI.
MOVE SPACES TO LINEAFL9L OF DSN8CCDI.
MOVE ZEROES TO LINEAFL10L OF DSN8CCDI.
MOVE SPACES TO LINEAFL10L OF DSN8CCDI.
MOVE ZEROES TO LINEAFL11L OF DSN8CCDI.
MOVE SPACES TO LINEAFL11L OF DSN8CCDI.
MOVE ZEROES TO LINEAFL12L OF DSN8CCDI.
MOVE SPACES TO LINEAFL12L OF DSN8CCDI.
MOVE ZEROES TO LINEAFL13L OF DSN8CCDI.
MOVE SPACES TO LINEAFL13L OF DSN8CCDI.
MOVE ZEROES TO LINEAFL14L OF DSN8CCDI.
MOVE SPACES TO LINEAFL14L OF DSN8CCDI.
MOVE ZEROES TO LINEAFL15L OF DSN8CCDI.
MOVE SPACES TO LINEAFL15L OF DSN8CCDI.
MOVE ZEROES TO LINEAFL16L OF DSN8CCDI.
MOVE SPACES TO LINEAFL16L OF DSN8CCDI.
MOVE SPACES TO LINEBF1I OF DSN8CCDI.
MOVE ZEROES TO LINEBF1L OF DSN8CCDI.
MOVE SPACES TO LINEBF2I OF DSN8CCDI.
MOVE ZEROES TO LINEBF2L OF DSN8CCDI.
MOVE SPACES TO LINECF1I OF DSN8CCDI.
MOVE ZEROES TO LINECF1L OF DSN8CCDI.
MOVE SPACES TO LINECF2I OF DSN8CCDI.
MOVE ZEROES TO LINECF2L OF DSN8CCDI.
MOVE SPACES TO LINEDF1I OF DSN8CCDI.
MOVE ZEROES TO LINEDF1L OF DSN8CCDI.
MOVE SPACES TO LINEDF2I OF DSN8CCDI.
MOVE ZEROES TO LINEDF2L OF DSN8CCDI.
MOVE SPACES TO LINEEF1I OF DSN8CCDI.
MOVE ZEROES TO LINEEF1L OF DSN8CCDI.
MOVE SPACES TO LINEEF2I OF DSN8CCDI.
MOVE ZEROES TO LINEEF2L OF DSN8CCDI.
MOVE SPACES TO LINEFF1I OF DSN8CCDI.
MOVE ZEROES TO LINEFF1L OF DSN8CCDI.
MOVE SPACES TO LINEFF2I OF DSN8CCDI.
MOVE ZEROES TO LINEFF2L OF DSN8CCDI.
MOVE ZEROES TO BPFKEYL OF DSN8CCDI.
MOVE SPACES TO BPFKEYI OF DSN8CCDI.

*  MOVE 'DSN8CC0' TO MAJOR IN DSN8-MODULE-NAME.
MOVE 'O' TO MAJSYS IN OUTAREA.
MOVE '0' TO EXITCODE.
MOVE EIBTRMID TO CICSID OF PCONVSTA.
MOVE CONVID OF PCONVSTA TO SAVE-CONVID.

*****************************************************************
*    TRY TO RETRIEVE LAST CONVERSATION. IF SUCCESSFUL, USE THE
*    LAST SCREEN SPECIFIED TO RECEIVE INPUT FROM TERMINAL.
*****************************************************************
EXEC SQL SELECT LASTSCR
   INTO :PCONA.LASTSCR
   FROM   VCONA
   WHERE  CONVID = :SAVE-CONVID END-EXEC.

*****************************************************************
*    IF LAST CONVERSATION DOES NOT EXIST, THEN DO NOT ATTEMPT TO
*    RECEIVE INPUT MAP. GO DIRECTLY TO VALIDATION MODULES
*    TO GET TITLE ETC. FOR OUTPUT MAP.
*****************************************************************
IF SQLCODE = +100 THEN
   GO TO CC0SEND.

*****************************************************************
*    IF LAST CONVERSATION EXISTS, BUT OPERATOR HAS ENTERED DATA N
*    FROM A CLEARED SCREEN OR HAD ERASED ALL DATA ON A FORMATTED N
*    SCREEN AND PRESS Enter THEN ........
*    MOVE DATA INTO CORRESPONDING FIELDS IN INAREA AND GO TO
*    VALIDATION MODULES.
*****************************************************************
EXEC CICS HANDLE CONDITION MAPFAIL (CC0SEND) END-EXEC.
IF LASTSCR OF PCONA  NOT = 'DSN8002 ' THEN
   GO TO CC0-LABEL1.

EXEC CICS RECEIVE MAP ('DSN8CC0') MAPSET ('DSN8CCD')
   END-EXEC.
IF BMJASYSYL NOT = 0 THEN MOVE BMJASYSI TO MAJSYS OF INAREA
ELSE MOVE 'O' TO MAJSYSI OF INAREA.
IF BACTIONL NOT = 0 THEN MOVE BACTIONI TO ACTION OF INAREA
ELSE MOVE SPACES TO ACTION OF INAREA.
IF BOBJECTL NOT = 0 THEN MOVE BOBJECTI TO OBJFLD OF INAREA
ELSE MOVE SPACES TO OBJFLD OF INAREA.
IF BSEARCHL NOT = 0 THEN MOVE BSEARCHI TO SRCH OF INAREA
ELSE MOVE SPACES TO SRCH OF INAREA.
IF BDATAL   NOT = 0 THEN MOVE BDATAI TO DATAIN OF INAREA
ELSE MOVE SPACES TO DATAIN OF INAREA.
MOVE 1 TO I.

**GO TO VALIDATION MODULES
GO TO CC0-LABELX.
**ERROR ON LASTSCREEN?

CC0-LABEL1.
  IF LASTSCR OF PCONA NOT = 'DSN8001 ' THEN
    EXEC CICS ABEND ABCODE ('LSCR') END-EXEC
  GOBACK.

**USING LAST SCREEN
**SPECIFIED TO RECEIVE
**INPUT FROM TERMINAL

EXEC CICS RECEIVE MAP ('DSN8CCG') MAPSET('DSN8CCG') END-EXEC.

*****************************************************************
*    IF DATA IS RECEIVED FOR A FIELD, THEN MOVE THE DATA INTO THE
*    CORRESPONDING FIELD IN INAREA, OTHERWISE MOVE BLANKS.
*****************************************************************

IF AMAJSYSL NOT = 0 THEN MOVE AMAJSYSI TO MAJSYS OF INAREA
  ELSE MOVE 'O'      TO MAJSYS OF INAREA.
IF AACTIONL NOT = 0 THEN MOVE AACTIONI TO ACTION OF INAREA
  ELSE MOVE SPACES   TO ACTION OF INAREA.
IF AOBJECTL NOT = 0 THEN MOVE AOBJECTI TO OBJFLD OF INAREA
  ELSE MOVE SPACES   TO OBJFLD OF INAREA.
IF ASEARCHL NOT = 0 THEN MOVE ASEARCHI TO SRCH   OF INAREA
  ELSE MOVE SPACES   TO SRCH   OF INAREA.
IF ADATAL   NOT = 0 THEN MOVE ADATAI   TO DATAIN OF INAREA
  ELSE MOVE SPACES   TO DATAIN OF INAREA.
GO TO CC0-LABEL3.

CC0-LABELX.
  IF COL2LEN(I) NOT = 0 THEN MOVE COL2DATA(I) TO TRANDATA(I)
  ELSE MOVE SPACES   TO TRANDATA(I).
  ADD 1 TO I.

** CC0-LABELX LOOP

CC0-LOOPX.
  PERFORM CC0-LABELX UNTIL I > 15.

CC0-LABEL3.
  MOVE 1 TO I.
  MOVE 0 TO FOUND.

*****************************************************************
*    CONVERT THE PFKEY INFO IN EIBAID TO THE FORM ACCEPTED
*    BY DSN8CC1 AND DSN8CC2 EG. PF1 = '01' AND PF13 = '01'.
*****************************************************************

CC0-LABEL4.

**PF KEYS 1-12

IF PFKEYS(I) = EIBAID THEN MOVE 1 TO FOUND
  ELSE ADD 1 TO I.

** CC0-LABEL4 LOOP

CC0-LOOP4.
  PERFORM CC0-LABEL4 UNTIL
    I > 24 OR FOUND = 1.

**PF KEYS > 12

CC0-LABEL5.
  IF I >  12 THEN SUBTRACT 12 FROM I.
  IF FOUND = 1 THEN
    MOVE PFK(I) TO PFKIN OF INAREA
    ELSE MOVE SPACES TO PFKIN OF INAREA.
  GO TO CC0-LABEL6.

*****************************************************************
*    GO TO DSN8CC1, GET DCLGEN STRUCTURES AND TABLE DCL
*****************************************************************

CC0SEND.
  MOVE SPACES TO INAREA.
  MOVE '00' TO PFKIN OF INAREA.

CC0-LABEL6.
  MOVE 'O' TO MAJSYS IN INAREA.
  EXEC CICS LINK PROGRAM ('DSN8CC1') COMMAREA(COMMAREA)
    LENGTH(3000) END-EXEC.
  GO TO CC0-NORMAL.
  EXEC SQL INCLUDE DSN8MCXX END-EXEC.
**AFTER RETURN FROM DSN8CC1, MOVE DATA TO OUTPUT MAP AREA AND**
**SEND MAP ACCORDING TO MAP SPECIFIED IN LASTSCR OF PCONVSTA.**

**CC0-NORMAL.**

**IF LASTSCR OF PCONVSTA = 'DSN8002' THEN GO TO CC0-LABEL9.**

**MOVE HTITLE OF OUTAREA TO ATITLEO.**
**MOVE MAJSYS OF OUTAREA TO AMAJSYSO.**
**MOVE ACTION OF OUTAREA TO AACTIONO.**
**MOVE OBJFLD OF OUTAREA TO AOBJECTO.**
**MOVE SRCH OF OUTAREA TO ASEARCHO.**
**MOVE DATAOUT TO ADATAG.**
**MOVE MSG OF OUTAREA TO AMSGO.**
**MOVE DESC2 OF OUTAREA TO ADESCL2O.**
**MOVE DESC3 OF OUTAREA TO ADESCL3O.**
**MOVE DESC4 OF OUTAREA TO ADESCL4O.**
**MOVE PFKTEXT OF OUTAREA TO APFKEYO.**
**MOVE 1 TO I.**

****SEND MAP ACCORDING TO**
****PREVIOUS SCREEN**

**CC0-LABEL7.**
**MOVE LINE0(I) TO ALINEO(I).**
**ADD 1 TO I.**

****CC0-LABEL7 LOOP**

**PERFORM CC0-LABEL7 UNTIL**
**I > 15.**

**creates a dynamic cursor**

**SET CURSOR POSITION**

**CC0-LABEL8.**
**MOVE ZEROES TO CURSOR-VALUE.**
**IF AACTIONO = SPACES THEN MOVE +179 TO CURSOR-VALUE**
**ELSE IF AOBJECTO = SPACES THEN MOVE +259 TO CURSOR-VALUE**
**ELSE IF ASEARCHO = SPACES THEN MOVE +339 TO CURSOR-VALUE**
**ELSE IF AEXTAQ = SPACES OR AACTIONO = 'D' OR 'E' THEN**
**MOVE +419 TO CURSOR-VALUE.**

****SEND OUTPUT MAP**

**IF CURSOR-VALUE = ZEROES THEN**
**EXEC CICS SEND MAP('DSNBCCG') MAPSET('DSNBCCG') END-EXEC**
**ELSE**
**EXEC CICS SEND MAP('DSNBCCG') MAPSET('DSNBCCG') ERASE**
**CURSOR(CURSOR-VALUE) END-EXEC.**

****FINISHED?**
**IF EXITCODE = '1' THEN GO TO CC0-LABEL12.**
**EXEC CICS RETURN TRANSID('D8CS') END-EXEC.**

**moves data from output map area to**
**receive map according to map specified in lastscr of pconvsta**

**FROM OUTPUT FIELDS**

**CC0-LABEL9.**
**MOVE HTITLE OF OUTAREA TO BTITLEO.**
**MOVE MAJSYS OF OUTAREA TO BMASYSO.**
**MOVE ACTION OF OUTAREA TO BACTIONO.**
**MOVE OBJFLD OF OUTAREA TO BOBJECTO.**
**MOVE SRCH OF OUTAREA TO BSSEARCHO.**
**MOVE DATAOUT TO BDATAG.**
**MOVE MSG OF OUTAREA TO BMMSGO.**
**MOVE DESC2 OF OUTAREA TO BDESCL2O.**
**MOVE DESC3 OF OUTAREA TO BDESCL3O.**
**MOVE DESC4 OF OUTAREA TO BDESCL4O.**
**MOVE PFKTEXT OF OUTAREA TO BPKEYO.**
**MOVE 1 TO I.**

****RECEIVE MAP ACCORDING**
****TO PREVIOUS SCREEN**

Chapter 12. Sample data and applications supplied with Db2 for z/OS 1399
CC0-LABEL10.
   MOVE FIELD1(I) TO COL1DATA(I).
   ** CHECK FOR ATTRIBUTE OF 'CHAR'
   IF ATTR(I) = -16191 THEN MOVE -1 TO COL2LEN(I).
   MOVE ATTR2(I) TO COL2ATTR(I).
   MOVE FIELD2(I) TO COL2DATA(I).
   ADD 1 TO I.
*                          ** CC0-LABEL10 LOOP
CC0-LOOP10.
   PERFORM CC0-LABEL10 UNTIL
   I > 15.
CC0-LABEL11.
*******************************************************************
*    CREATES A DYNAMIC CURSOR
*******************************************************************
*                     **SET CURSOR POSITION
   MOVE ZERES TO CURSOR-VALUE.
   IF BACTIONO = SPACES THEN MOVE +179 TO CURSOR-VALUE
   ELSE IF BOBJECTO = SPACES THEN MOVE +259 TO CURSOR-VALUE
   ELSE IF BSEARCHO = SPACES THEN MOVE +339 TO CURSOR-VALUE
   ELSE IF BDATAO = SPACES OR BACTIONO = 'D' OR 'E' THEN
      MOVE +419 TO CURSOR-VALUE.
   **SEND INPUT MAP
   IF CURSOR-VALUE = ZERES THEN
      EXEC CICS SEND MAP('DSN8CCD') MAPSET('DSN8CCD') END-EXEC
   ELSE
      EXEC CICS SEND MAP('DSN8CCD') MAPSET('DSN8CCD') ERASE
         CURSOR(CURSOR-VALUE) END-EXEC.
   **FINISHED?
   IF EXITCODE = '1' THEN GO TO CC0-LABEL12.
   EXEC CICS RETURN TRANSID('DBCS') END-EXEC.
   GOBACK.
*                     **RETURN
   EXEC CICS RETURN END-EXEC.
   GOBACK.

Related reference
“Sample applications in CICS” on page 1392
A set of Db2 sample applications run in the CICS environment.

DSN8CC1

THIS MODULE PERFORMS THE INCLUDES TO BRING IN THE SQL TABLE DCLS AND DCLGEN STRUCTURES AS WELL AS THE PARAMETER AREA.
* NOTES =
* DEPENDENCIES = CALLED BY DSN8CC0, CALLS DSN8CC2(CICS LINKS).*
* RESTRICTIONS = NONE
* *
* MODULE TYPE =
* PROCESSOR = DB2 PRECOMPILE, CICS TRANSLATOR, COBOL COMPILER
* MODULE SIZE = SEE LINK_EDIT
* ATTRIBUTES = REUSABLE
* *
* ENTRY POINT = DSN8CC1
* PURPOSE = SEE FUNCTION
* LINKAGE = INCLUDED BY MODULE DSN8MC1
* *
* INPUT = PARAMETERS EXPLICITLY PASSED TO THIS FUNCTION:
* SYMBOLIC LABEL/NAME = NONE
* DESCRIPTION = NONE
* *
* OUTPUT = PARAMETERS EXPLICITLY RETURNED:
* SYMBOLIC LABEL/NAME = NONE
* DESCRIPTION = NONE
* *
* EXIT-NORMAL = DSN8CC0
* EXIT-ERROR = DSN8CC0
* RETURN CODE = NONE
* ABEND CODES = NONE
* ERROR-MESSAGES = NONE
* *
* EXTERNAL REFERENCES =
* ROUTINES/SERVICES = DSN8CC2
* *
* DATA-AREAS =
* DSN8BMC8 - COBOL STRUCTURE FOR DFHCOMMAREA
* DSN8BMC9 - VCONA TABLE DCL AND PCONA DCLGEN
* DSN8BMC2 - COMMON AREA PART 2
* DSN8MC0 - VOPTVAL TABLE DCL & POPTVAL DCLGEN
* DSN8MCV - FINDS VALID OPTIONS FOR ACTION,
* OBJECT, SEARCH CRITERIA
* DSN8MC1 - SQL1 COMMON MODULE FOR IMS AND CICS
* DSN8MC3 - DSN8MC5 - VALIDATION MODULES CALLED BY DSN8MC0*
* DSN8MCXX - SQL ERROR HANDLER
* *
* CONTROL-BLOCKS =
* SQLCA - SQL COMMUNICATION AREA
* *
* TABLES = NONE
* *
* CHANGE-ACTIVITY =
* 10/18/2005 PK03311 INITIALIZE UNINITIALIZED STORAGE @01
* *PSEUDOCODE*
* *
* PROCEDURE
* INCLUDE DECLARATIONS.
* INCLUDE DSN8MC1.
* INCLUDE ERROR HANDLER.
* *
* CC1EXIT: (REFERENCED BY DSN8MC1)
* EXEC CICS RETURN.
* *
* CC1ICALL: (REFERENCED BY DSN8MC1)
* EXEC CICS LINK PROGRAM('DSN8MC2')
* COMMAREA(DFHCOMMAREA).
* GO TO MC1SAVE. (LABEL IN DSN8MC1)
* *
* INCLUDE VALIDATION MODULES.
* *
* END.

*****************************************************************
ENVIRONMENT DIVISION.
------------------------
DATA DIVISION.
------------------------
WORKING-STORAGE SECTION.
*****************************************************************

Chapter 12. Sample data and applications supplied with Db2 for z/OS 1401
* * DECLARE FIELD PASSED TO MESSAGE ROUTINE * *
* * DECLARE CONVERSATION STATUS * *
* * DECLARE MESSAGE TEXT * *
* * DECLARE OPTION VALIDATION * *
* * DECLARE COMMON AREA AND COMMON AREA PART 2 * *
*****************************************************************
01 MSGCODE    PIC X(04).
01 OUTMSG     PIC X(69).
EXEC SQL INCLUDE DSN8MCCS END-EXEC.
EXEC SQL INCLUDE DSN8MCOV END-EXEC.
EXEC SQL INCLUDE SQLCA END-EXEC.
EXEC SQL INCLUDE DSN8MCC2 END-EXEC.
LINKAGE SECTION.
01 DFHCOMMAREA.
  EXEC SQL INCLUDE DSN8MCCA END-EXEC.
PROCEDURE DIVISION.
*------------------------
* INIT AREA INCLUDED BY DSN8MCCS @01
MOVE SPACES TO PCONA.
* INIT AREA INCLUDED BY DSN8MCOV @01
MOVE SPACES TO POPTVAL.
****************************************************************
* SQL RETURN CODE HANDLING
****************************************************************
EXEC SQL WHENEVER SQLError GO TO DB-ERROR END-EXEC
EXEC SQL WHENEVER SQLWarning GO TO DB-ERROR END-EXEC.

MOVE 'DSN8CC1' TO MAJOR IN DSN8-MODULE-NAME.
*******************************************************************************
* FIND VALID OPTIONS FOR ACTION, OBJECT, SEARCH CRITERION*
* RETRIEVE CONVERSATION, VALIDATE, CALL SQL2 *
*******************************************************************************
EXEC SQL INCLUDE DSN8MCVO END-EXEC.
EXEC SQL INCLUDE DSN8MC1 END-EXEC.
**RETURN
CC1-EXIT.
EXEC CICS RETURN END-EXEC.
*******************************************************************************
* VALIDATE ACTION, OBJECT, SEARCH CRITERIA *
* HANDLE ERRORS *
*******************************************************************************
CC1-CALL.
EXEC CICS LINK PROGRAM('DSN8CC2') COMMAREA(DFHCOMMAREA) LENGTH(3000) END-EXEC.
GO TO MC1-SAVE.
EXEC SQL INCLUDE DSN8MC3 END-EXEC.
EXEC SQL INCLUDE DSN8MC4 END-EXEC.
EXEC SQL INCLUDE DSN8MC5 END-EXEC.
EXEC SQL INCLUDE DSN8MCXX END-EXEC.
GOBACK.

Related reference
“Sample applications in CICS” on page 1392
A set of Db2 sample applications run in the CICS environment.

DSN8CC2
ROUTER FOR SECONDARY SELECTION AND/OR DETAIL PROCESSING CALLS SECONDARY SELECTION
MODULES DSN8MCA DSN8MCM CALLS DETAIL MODULES DSN8MCD DSN8MCE DSN8MCF DSN8MCT
DSN8MCV DSN8MCM DSN8MCX DSN8MCMZ CALLED BY DSN8MC1 (SQL1).

Identification Division.
*------------------------
Program-ID. DSN8CC2.
********** DSN8CC2 - COMMON MODULE FOR CICS - COBOL**********

* MODULE NAME = DSN8CC2
* DESCRIPTIVE NAME = DB2 SAMPLE APPLICATION
* SQL 2 COMMON MODULE
* CICS
* COBOL
* Licensed Materials - Property of IBM
* (C) COPYRIGHT 1982, 2010 IBM Corp. All Rights Reserved.
* STATUS = Version 10
*
* FUNCTION = ROUTER FOR SECONDARY SELECTION AND/OR
* DETAIL PROCESSING
* CALLS SECONDARY SELECTION MODULES
* DSN8MCA DSN8MCM
* CALLS DETAIL MODULES
* DSN8MCD DSN8MCCE DSN8MCF
* DSN8MCT DSN8MCV DSN8MCW DSN8MCX DSN8MCZ
* CALLED BY DSN8MC1 (SQL1)
*
* NOTES = NONE
*
* MODULE TYPE =
* PROCESSOR = DB2 PRECOMPILED, CICS TRANSLATOR,
* VS COBOL
* MODULE SIZE = SEE LINKEDIT
* ATTRIBUTES = REUSABLE
*
* ENTRY POINT = DSN8CC2
* PURPOSE = SEE FUNCTION
* LINKAGE = NONE
* INPUT =
* SYMBOLIC LABEL/NAME = COMMPTR
* DESCRIPTION = POINTER TO COMMAREA
* (COMMUNICATION AREA)
*
* OUTPUT =
* SYMBOLIC LABEL/NAME = COMMPTR
* DESCRIPTION = POINTER TO COMMAREA
* (COMMUNICATION AREA)
*
* EXIT-NORMAL = RETURN CODE 0 NORMAL COMPLETION
* EXIT-ERROR =
* IF SQLERROR OR SQLWARNING, SQL WHENEVER CONDITION
* SPECIFIED IN DSN8CC2 WILL BE RAISED AND PROGRAM
* WILL GO TO THE LABEL DB-ERROR.
*
* RETURN CODE = NONE
*
* ABEND CODES = NONE
*
* ERROR MESSAGES =
* DSN8062E-AN OBJECT WAS NOT SELECTED
* DSN8066E-UNSUPPORTED PFK OR LOGIC ERROR
* DSN8072E-INVALID SELECTION ON SECONDARY SCREEN
* EXTERNAL REFERENCES = NONE
* ROUTINES/SERVICES = 10 MODULES LISTED ABOVE
* DSN8MCG - ERROR MESSAGE ROUTINE
*
* DATA AREAS =
* DSN8MCA - SECONDARY SELECTION FOR ORGANIZATION
* DSN8MCAE - DECLARE ADMINISTRATION DETAIL
* DSN8MCAEM - CURSOR EMPLOYEE LIST
* DSN8MCAEM - CURSOR ADMINISTRATION LIST
* DSN8MCAEM - DECLARE ADMINISTRATION DETAIL
* DSN8MCAEM - COMMON AREA
* DSN8MCAEM - COMMON AREA PART 2
* DSN8MCAEM - DEPARTMENT STRUCTURE DETAIL

Chapter 12. Sample data and applications supplied with Db2 for z/OS 1403
* DSNBMCDA - CURSOR ADMINISTRATION DETAIL *
* DSNBMCDB - CURSOR FOR DISPLAY TEXT FROM *
* TDSPTXT TABLE *
* DSNBMCDD - DECLARE DEPARTMENT MANAGER *
* DSNBMCDF - DECLARE DEPARTMENT *
* DSNBMCdT - DECLARE DISPLAY TEXT *
* DSNBMCDF - DEPARTMENT DETAIL *
* DSNBMCBE - DECLARE DEPARTMENT *
* DSNBMCED - DECLARE EMPLOYEE-DEPARTMENT *
* DSNBMCED - EMPLOYEE DETAIL *
* DSNBMCFO - DECLARE OPTION VALIDATION *
* DSNBMCXX - ERROR HANDLER *

* CONTROL-BLOCKS = *
* SQLCA - SQL COMMUNICATION AREA *
* TABLES = NONE *

* CHANGE-ACTIVITY = *
* - ADD NEW VARIABLES FOR REFERENCE INTEGRITY V2R1 *
* - 10/18/2005 PK03311 INITIALIZE UNINITIALIZED STORAGE @01 *

* PSEUDOCODE *
* *
* THIS MODULE DETERMINES WHICH SECONDARY SELECTION AND/OR *
* DETAIL MODULE(S) ARE TO BE CALLED IN THE CICS/COBOL *
* ENVIRONMENT.

* WHAT HAS HAPPENED SO FAR?..........THE SUBSYSTEM *
* DEPENDENT MODULE (IMS,CICS,TSO) OR (SQL 0) HAS *
* READ THE INPUT SCREEN, FORMATTED THE INPUT AND PASSED CONTROL *
* TO SQL 1. SQL 1 PERFORMS VALIDATION ON THE SYSTEM DEPENDENT *
* FIELDS (MAJOR SYSTEM, ACTION, OBJECT, SEARCH CRITERIA). IF *
* ALL SYSTEM FIELDS ARE VALID SQL 1 PASSED CONTROL TO THIS *
* MODULE. PASSED PARAMETERS CONSIST ONLY OF A POINTER WHICH *
* POINTS TO A COMMUNICATION CONTROL AREA USED TO COMMUNICATE *
* BETWEEN SQL 0 , SQL 1, SQL 2 AND THE SECONDARY SELECTION *
* AND DETAIL MODULES.

* WHAT IS INCLUDED IN THIS MODULE?...........
* ALL SECONDARY SELECTION AND DETAIL MODULES ARE 'INCLUDED'. *
* ALL VARIABLES KNOWN IN THIS PROCEDURE ARE KNOWN IN THE *
* SUB PROCEDURES. ALL SQL CURSOR DEFINITIONS AND *
* SQL 'INCLUDES' ARE DONE IN THIS PROCEDURE, BECAUSE OF THE *
* RESTRICTION THAT CURSOR HOST VARIABLES MUST BE DECLARED BEFORE *
* THE CURSOR DEFINITION ALL CURSOR HOST VARIABLES ARE DECLARED *
* IN THIS PROCEDURE.

* PROCEDURE *
* IF ANSWER TO DETAIL SCREEN AND DETAIL PROCESSOR *
* IS NOT WILLING TO ACCEPT AN ANSWER THEN *
* NEW REQUEST*
* ELSE *
* IF ANSWER TO A SECONDARY SELECTION THEN *
* DETERMINE IF NEW REQUEST.

* CASE (NEW REQUEST) *
* SUBCASE ('ADD') *
* DETAIL PROCESSOR *
* RETURN TO SQL 1 *

* SUBCASE ('ERASE', 'DISPLAY', 'UPDATE') *
* CALL SECONDARY SELECTION *
* IF # OF POSSIBLE CHOICES IS ^= 1 THEN *
* RETURN TO SQL 1 *
* ELSE *
* CALL THE DETAIL PROCESSOR *
* RETURN TO SQL 1 *

* ENDCASE *
* IF ANSWER TO SECONDARY SELECTION AND A SELECTION HAS *
* ACTUALLY BEEN MADE THEN *
* IF IT IS A VALID SELECTION NUMBER THEN *
* CALL DETAIL PROCESSOR *
* RETURN TO SQL 1 *
EXEC SQL INCLUDE SQLCA END-EXEC.
EXEC SQL INCLUDE DSN8MCC2 END-EXEC.
EXEC SQL INCLUDE DSN8MCDP END-EXEC.
EXEC SQL INCLUDE DSN8MCEM END-EXEC.
EXEC SQL INCLUDE DSN8MCDM END-EXEC.
EXEC SQL INCLUDE DSN8MCAD END-EXEC.
EXEC SQL INCLUDE DSN8MCA2 END-EXEC.
EXEC SQL INCLUDE DSN8MCOV END-EXEC.
EXEC SQL INCLUDE DSN8MCDT END-EXEC.
EXEC SQL INCLUDE DSN8MCED END-EXEC.

01 CONSTRAINTS.
 03 PARM-LENGTH PIC S9(4) COMP-4.
 03 REF-CONSTRAINT PIC X(08).
 03 FILLER PIC X(62).
 01 MGRNO-CONSTRAINT PIC X(08) VALUE 'RDE '.

LINKAGE SECTION.
01 DFHCOMMAREA.
EXEC SQL INCLUDE DSN8MCCA END-EXEC.

PROCEDURE DIVISION.
*------------------
EXEC SQL INCLUDE DSN8MCAL END-EXEC.
EXEC SQL INCLUDE DSN8MCDH END-EXEC.
EXEC SQL INCLUDE DSN8MCDA END-EXEC.

*******************************************************************
Chapter 12. Sample data and applications supplied with Db2 for z/OS 1405
* SQL RETURN CODE HANDLING                               *
* *****************************************************************
EXEC SQL WHENEVER SQLERROR   GO TO DB-ERROR END-EXEC
EXEC SQL WHENEVER SQLWARNING GO TO DB-ERROR END-EXEC.
* *****************************************************************

* INITIALIZATIONS                                        *
* ***********************************************************
MOVE 'DSN8CC2' TO MAJOR.
MOVE SPACES TO MINOR.
* INIT AREA INCLUDED BY DSN8MCDP                              @01
  MOVE SPACES TO PDEPT.
* INIT AREA INCLUDED BY DSN8MCEM                              @01
  MOVE SPACES TO PEMP.
* INIT AREA INCLUDED BY DSN8MCDM                              @01
  MOVE SPACES TO PDEPMGR.
* INIT AREA INCLUDED BY DSN8MCA2                              @01
  MOVE SPACES TO PASTRDET.
* INIT AREA INCLUDED BY DSN8MCAD                              @01
  MOVE SPACES TO PASTRDE2.
* INIT AREA INCLUDED BY DSN8MCVO                              @01
  MOVE SPACES TO PLOPTVAL.
* INIT AREA INCLUDED BY DSN8MCDT                              @01
  MOVE SPACES TO PDSPTXT.
* INIT AREA INCLUDED BY DSN8MCDM                              @01
  MOVE SPACES TO PEMPDEPT.

* DETERMINES WHETHER NEW REQUEST OR NOT                  *
* *****************************************************************
IC200B.

  IF PREV OF PCONVSTA = ' ' THEN
    MOVE 'Y' TO NEWREQ OF COMPARM.
  IF NEWREQ OF COMPARM = 'N' AND PREV OF PCONVSTA = 'S'
    AND DATA01 NOT = ' ' AND PFKIN  NOT = '08'
    THEN MOVE 'Y' TO NEWREQ OF COMPARM.
  IF NEWREQ OF COMPARM NOT = 'Y' THEN
    GO TO IC2010.

* IF NEW REQUEST AND ACTION IS 'ADD' THEN                *
  CALL DETAIL PROCESSOR                             *
* ELSE CALL SECONDARY SELECTION                          *
* ***********************************************************
IC201.

* DETERMINES IF VALID SELECTION NUMBER                   *
* ***********************************************************
IC2010.

  **VALID SELECTION NO. GIVEN
  IF PREV OF PCONVSTA NOT = 'S' OR
    MAXSEL < 1 OR
    PFKIN  = '08' OR
    DATA2 = DATO2 THEN
    GO TO IC201.

  **DETAIL SELECTION GIVEN
  IF DAT1 NUMERIC AND DAT2 = ' ' THEN
    MOVE DAT1 TO DAT2
    MOVE '0' TO DAT1.
  IF DAT2 NUMERIC
    AND DAT2 > '00' AND DAT2 NOT > MAXSEL THEN
    MOVE 'Y' TO NEWREQ OF COMPARM
    GO TO DETAIL0.

  **INVALID SELECTION NO.
  **PRINT ERROR MESSAGE
  MOVE '872E' TO MSGCODE.
  CALL 'DSN8MCG' USING MAJOR MSGCODE OUTMSG.
  MOVE OUTMSG TO MSG OF OUTAREA.
GO TO EXITO.

***************************************************************************
* DETERMINES WHETHER SECONDARY SELECTION OR DETAIL                  *
***************************************************************************
IC201.
* **SECONDARY SELECTION
* IF PREV OF PCONVSTA = 'S' THEN
  PERFORM SECSEL THRU END-SECSEL
  GO TO EXITO
ELSE
* **DETAIL PROCESSOR
* IF PREV OF PCONVSTA = 'D' THEN GO TO DETAIL0.
* **LOGIC ERROR
* **PRINT ERROR MESSAGE
  MOVE '066E' TO MSGCODE.
  CALL 'DSN8MCG' USING MAJOR MSGCODE OUTMSG.
  MOVE OUTMSG TO MSG OF OUTAREA.
  GO TO EXITO.
* **HANDLES ERRORS
  EXEC SQL INCLUDE DSN8MCXX END-EXEC.
  GO TO EXITO.

***************************************************************************
* CALLS SECONDARY SELECTION AND RETURNS TO SQL 1                      *
***************************************************************************
SECSEL.
* **ADMINISTRATIVE
* **DEPARTMENT STRUCTURE
  MOVE 'DSN8001' TO LASTSCR IN PCONVSTA.
* **INDIVIDUAL DEPARTMENT
* **PROCESSING
  IF OBJFLD OF INAREA = 'DS' THEN
    PERFORM DSN8MCA THRU END-DSN8MCA
  ELSE
* **INDIVIDUAL EMPLOYEE
* **PROCESSING
  IF OBJFLD OF INAREA = 'EM' THEN
    PERFORM DSN8MCA THRU END-DSN8MCA
  ELSE
* **ERROR MESSAGE
* **MISSING SECONDARY SEL
  MOVE '062E' TO MSGCODE
  CALL 'DSN8MCG' USING MAJOR MSGCODE OUTMSG
  MOVE OUTMSG TO MSG OF OUTAREA.
END-SECSEL.

***************************************************************************
* CALLS DETAIL PROCESSOR AND RETURNS TO SQL 1                        *
***************************************************************************
DETAIL0.
* **ADMINISTRATIVE
* **DEPARTMENT STRUCTURE
  MOVE 'DSN8002' TO LASTSCR IN PCONVSTA.
* **INDIVIDUAL DEPARTMENT
* **PROCESSING
  IF OBJFLD OF INAREA = 'DS' THEN
    PERFORM DSN8MCD THRU END-DSN8MCD
  ELSE
* **INDIVIDUAL EMPLOYEE
* **PROCESSING
  IF OBJFLD OF INAREA = 'EM' THEN
    PERFORM DSN8MCE THRU END-DSN8MCE
  ELSE
* **ERROR MESSAGE
* **MISSING DETAIL MODULE
  MOVE '062E' TO MSGCODE
  CALL 'DSN8MCG' USING MAJOR MSGCODE OUTMSG
  MOVE OUTMSG TO MSG OF OUTAREA.
GO TO EXITO.
**RETURNS TO SQL 1

EXITO.

EXEC CICS RETURN END-EXEC.
EXEC SQL INCLUDE DSN8MCA END-EXEC.
EXEC SQL INCLUDE DSN8MCD END-EXEC.
EXEC SQL INCLUDE DSN8MCE END-EXEC.
EXEC SQL INCLUDE DSN8MCF END-EXEC.
GOBACK.

Related reference
“Sample applications in CICS” on page 1392
A set of Db2 sample applications run in the CICS environment.

DSN8CP0

THIS MODULE ISSUES A CICS RECEIVE MAP TO RETRIEVE INPUT, CALLS DSN8CP1, AND ISSUES A CICS SEND MAP AFTER RETURNING.
Chapter 12. Sample data and applications supplied with Db2 for z/OS
/* LAST SCREEN SPECIFIED TO RECEIVE INPUT FROM TERMINAL. */
EXEC SQL SELECT LASTSCR INTO :PCONA.LASTSCR FROM VCONA WHERE CONVID = :CONVID ;
/* IF LAST CONVERSATION DOES NOT EXIST, THEN DO NOT ATTEMPT TO RECEIVE INPUT MAP. GO DIRECTLY TO VALIDATION MODULES */
/* TO GET TITLE ETC. FOR OUTPUT MAP. */
IF SQLCODE = +100 THEN GO TO CP0SEND;
/* IF DATA IS RECEIVED FOR A FIELD, THEN MOVE THE DATA INTO THE CORRESPONDING FIELDS IN INAREA, OTHERWISE MOVE BLANKS. */
/* IF LAST CONVERSATION EXISTS, BUT OPERATOR HAS ENTERED DATA FROM A CLEARED SCREEN OR HAD ERASED ALL DATA ON A FORMATTED SCREEN AND PRESSED ENTER THEN MOVE DATA INTO CORRESPONDING FIELDS IN INAREA AND GO TO VALIDATION MODULES. */
IF PCONA.LASTSCR = 'DSN8001' THEN DO;
USING LAST SCREEN */ SPECIFIED TO RECEIVE/ INPUT FROM TERMINAL*/ EXEC CICS RECEIVE MAP ('DSN8CPG') MAPSET ('DSN8CPG');
IF AMAJSYSL ^= 0 THEN COMPARM.MAJSYS = AMAJSYSI; ELSE COMPARM.MAJSYS = 'O';
IF AACTIONL ^= 0 THEN COMPARM.ACTION = AACTIONI; ELSE COMPARM.ACTION = ' '; IF AOBJECTCL ^= 0 THEN COMPARM.OBJFLD = AOBJECTI; ELSE COMPARM.OBJFLD = ' '; IF ASEARCHL ^= 0 THEN COMPARM.SEARCH = ASEARCHI; ELSE COMPARM.SEARCH = ' '; IF ADATAL ^= 0 THEN COMPARM.DATA = ADATAI ELSE COMPARM.DATA = ' '; END;
DO I = 1 TO 15;
IF SUBMAP.COL2LEN(I) ^= 0 THEN COMPARM.TRANDATA(I) = SUBMAP.COL2DATA(I) ELSE COMPARM.TRANDATA(I) = ' '; END;
ELSE IF PCONA.LASTSCR = 'DSN8002' THEN DO;
MOVE DATA INTO */ INPUT FIELDS */ EXEC CICS RECEIVE MAP ('DSN8CPD') MAPSET ('DSN8CPD');
IF BMAJSYSL ^= 0 THEN COMPARM.MAJSYS = BMAJSYSI; ELSE COMPARM.MAJSYS = 'O';
IF BACTIONL ^= 0 THEN COMPARM.ACTION = BACTIONI; ELSE COMPARM.ACTION = ' '; IF BOBJECTCL ^= 0 THEN COMPARM.OBJFLD = BOBJECTI; ELSE COMPARM.OBJFLD = ' '; IF BSEARCHL ^= 0 THEN COMPARM.SEARCH = BSEARCHI; ELSE COMPARM.SEARCH = ' '; IF BDATAL ^= 0 THEN COMPARM.DATA = BDATAI ELSE COMPARM.DATA = ' '; END;
DO I = 1 TO 15;
IF SUBMAP.COL2LEN(I) ^= 0 THEN COMPARM.TRANDATA(I) = SUBMAP.COL2DATA(I) ELSE COMPARM.TRANDATA(I) = ' '; END;
ELSE /* WRONG LASTSCREEN NAME*/ EXEC CICS ABEND ABCODE ('MAPI');
/* CONVERT THE PFKEY INFO IN EIBAID TO THE FORM ACCEPTED BY DSN8CP1 AND DSN8CP2 ETC. Eg. PF1 = '01' AND PF13 = '01'. */
N = INDEX (PFSTRG, EIBAID);
IF N ^= 0 THEN                            /* IF PF KEY USED    */
  DO;
    IF N > 12 THEN N = N - 12 ;         /* PF13 = PF1 ETC.   */
    COMPARM.PFKIN = PFK(N) ;
  END;
ELSE COMPARM.PFKIN = ' ' ;                /* IF ENTER | PAKEYS */
  GO TO CP0CP1;
END;                                                            /*     */
GO TO DSN8CP1, GET DCLGEN STRUCTURES AND TABLE DCL
/******  */
CP0SEND :                                                /*     */
INAREA = ' ' ;                       /*BLANK OUT INAREA */
COMPARM.PFKIN = '00' ;               /*PUT '00' INTO PFKIN*/
/GO TO DSN8CP1 */
EXEC CICS LINK PROGRAM ('DSN8CP1') COMMAREA(COMMAREA) LENGTH(3000); 0
EXEC SQL INCLUDE DSN8MPXX;              /*GET DCLGEN STRUCTURES*/
/******  */
IF PCONVSTA.LASTSCR = 'DSN8001 ' THEN        /*MOVE DATA INTO */
  DO;                                         /*OUTPUT FIELDS */
    ATITLEO = PCONVSTA.TITLE ;                                    0
    AMAJSYSO= PCONVSTA.MAJSYS;                                    0
    AACTIONO= PCONVSTA.ACTION;                                    0
    AOBJECTO= PCONVSTA.OBJFLD;                                    0
    ASEARCHO= PCONVSTA.SEARCH;                                    0
    ADATAO  = PCONVSTA.DATA  ;                                    0
    AMSGO   = PCONVSTA.MSG ;                                      0
    ADESCL2O= PCONVSTA.DESC2 ;                                    0
    ADESCL3O= PCONVSTA.DESC3 ;                                    0
    ADESCL4O= PCONVSTA.DESC4 ;                                    0
    APFKEYO = PCONVSTA.PFKTEXT;                                   0
  END;
  DO I = 1 TO 15;                    /*SEND MAP ACCORDING TO */
    ALINEO(I) = PCONVSTA.OUTPUT.LINE(I); /*PREVIOUS SCREEN*/
  END;                                                          0
  IF CURSOR_VALUE = 0 THEN           /*SEND OUTPUT MAP */       0
    EXEC CICS SEND MAP('DSN8CPG') MAPSET('DSN8CPG') ERASE;      0
  ELSE                                                          0
    EXEC CICS SEND MAP('DSN8CPG') MAPSET('DSN8CPG') ERASE       0
    CURSOR(CURSOR_VALUE);                           0
  0
  IF EXITCODE = '1' THEN             /*  FINISHED ?  */         0
    EXEC CICS RETURN ; /* EXIT, DON'T REINVOKETRANSACTION */   0
*/
Related reference
“Sample applications in CICS” on page 1392
A set of Db2 sample applications run in the CICS environment.

**DSN8CP1**

THIS MODULE PERFORMS THE INCLUDES TO BRING IN THE SQL TABLE DCLS AND DCLGEN STRUCTURES AS WELL AS PARAMETER AREA.
* DSN8MP3 -- DSN8MP5 - VALIDATION MODULES CALLED BY DSN8MP1 *
* DSN8MPXX - SQL ERROR HANDLER *
* *
* CONTROL-BLOCKS = *
* SQLCA - SQL COMMUNICATION AREA *
* *
* TABLES = NONE *
* *
* CHANGE-ACTIVITY = NONE *
* *
* *PSEUDOCODE* *
* *
* PROCEDURE *
* INCLUDE DECLARATIONS. *
* INCLUDE DSN8MP1. *
* INCLUDE ERROR HANDLER. *
* *
* CP1EXIT: (REFERRED BY DSN8MP1)
* EXEC CICS RETURN. *
* *
* CP1CALL: (REFERRED BY DSN8MP1)
* EXEC CICS LINK PROGRAM('DSN8CP2') COMMAREA(COMMAREA) LENGTH(3000).
* GO TO MP1SAVE. (LABEL IN DSN8MP1)
* *
* INCLUDE VALIDATION MODULES.
* *
* END.
* *------------------------------------------------------------------*/

EXEC SQL WHENEVER SQLERROR GO TO DB_ERROR;
EXEC SQL WHENEVER SQLWARNING GO TO DB_ERROR;

/***************************************************************************/
/* ** DCLGENS AND INITIALIZATIONS */
/***************************************************************************/
DCL STRING BUILTIN;
DCL J FIXED BIN;
DCL SAVE_CONVID CHAR(16);
/* DECLARE CONTROL FLAGS */
DCL (SENDBIT, ENDBIT, NEXTBIT, ON, OFF) BIT(1);

/***************************************************************************/
/* ** FIELDS SENT TO MESSAGE ROUTINE */
/***************************************************************************/
DCL MODULE CHAR (07);
DCL OUTMSG CHAR (69);

DCL DSN8MPG EXTERNAL ENTRY;
EXEC SQL INCLUDE DSN8MPCA; /* INCLUDE COMMAREA */
DSN8_MODULE_NAME.MAJOR = 'DSN8CP1'; /* INITIALIZE MODULE NAME*/
EXEC SQL INCLUDE DSN8MPICS; /* INCLUDE PCONA */
EXEC SQL INCLUDE DSN8MPOV; /* INCLUDE POPVAL */
EXEC SQL INCLUDE DSN8MPVO; /* INCLUDE CURSOR */
EXEC SQL INCLUDE SQLCA; /* INCLUDE SQL COMMAREA*/
EXEC SQL INCLUDE DSN8MP1; /* INCLUDE SQL1 MAIN*/
EXEC SQL INCLUDE DSN8MPXX; /* INCLUDE ERRORHANDLER */

CP1EXIT:
EXEC CICS RETURN; /* STANDARD EXIT */

CP1CALL:
/* GO TO DSN8CP2 (SQL2) */
EXEC CICS LINK PROGRAM('DSN8CP2') COMMAREA(COMMAREA) LENGTH(3000);
GO TO MP1SAVE;
Related reference
“Sample applications in CICS” on page 1392
A set of Db2 sample applications run in the CICS environment.

DSN8CP2
ROUTER FOR SECONDARY SELECTION AND/OR DETAIL PROCESSING CALLS SECONDARY SELECTION MODULES DSN8MPA DSN8MPM CALLS DETAIL MODULES DSN8MPD DSN8MPF DSN8MPV DSN8MPW DSN8MPX DSN8MPZ CALLED BY DSN8MP1 (SQL1).

EXEC SQL INCLUDE DSN8MP3;  /* INCLUDE ACTION VALIDATION*/
EXEC SQL INCLUDE DSN8MP4;  /* INCLUDE OBJECT VALIDATION*/
EXEC SQL INCLUDE DSN8MP5;  /* INCLUDE SEARCH CRITERIA*/
END;  /* VALIDATION */
* DSN0872E-INVALID SELECTION ON SECONDARY SCREEN * 00630000
* EXTERNAL REFERENCES = NONE * 00650000
* ROUTINES/SERVICES = 10 MODULES LISTED ABOVE * 00668000
* DSN0MPG - ERROR MESSAGE ROUTINE * 00670000
* DSN0MPH - DATA MANAGEMENT ROUTINE * 00680000
* DATA AREAS = DSN0MPA - SECONDARY SELECTION FOR ORGANIZATION * 00690000
* DSN0MPAD - DECLARE ADMINISTRATIVE DETAIL * 00700000
* DSN0MPAL - CURSOR EMPLOYEE LIST * 00720000
* DSN0MPAL - CURSOR ADMINISTRATION LIST * 00730000
* DSN0MPA2 - DECLARE ADMINISTRATIVE DETAIL * 00740000
* DSN0MPCA - DECLARE SQL COMMON AREA * 00750000
* DSN0MPD - DEPARTMENT STRUCTURE DETAIL * 00760000
* DSN0MPDA - CURSOR ADMINISTRATION LIST * 00770000
* DSN0MPDH - CURSOR FOR DISPLAY TEXT FROM * 00780000
* TDSPTXT TABLE * 00790000
* DSN0MPDM - DECLARE DEPARTMENT MANAGER * 00800000
* DSN0MPDP - DECLARE DEPARTMENT * 00810000
* DSN0MPAT - DECLARE DISPLAY TEXT * 00820000
* DSN0MPAT - DEPARTMENT DETAIL * 00830000
* DSN0MPAT - DECLARE EMPLOYEE * 00840000
* DSN0MPAT - DECLARE EMPLOYEE-DEPARTMENT * 00850000
* DSN0MPAT - DECLARE EMPLOYEE DETAIL * 00860000
* DSN0MPAT - DECLARE OPTION VALIDATION * 00870000
* DSN0MPAT - ERROR HANDLER * 00880000
* TABLES = NONE * 00900000
* SQLCA - SQL COMMUNICATION AREA * 00910000
* SQLAT - SQL AREA TABLE * 00920000
* CHANGE ACTIVITY = NONE * 00930000
* PSEUDOCODE = 00940000
* THIS MODULE DETERMINES WHICH SECONDARY SELECTION AND/OR DETAIL MODULE(S) ARE TO BE CALLED IN THE CICS/PL/I ENVIRONMENT. * 00950000
* WHAT HAS HAPPENED SO FAR? ..................THE SUBSYSTEM * 01000000
* DEPENDENT MODULE (IMS,CICS,TSO) OR (SQL 0) HAS * 01050000
* READ THE INPUT SCREEN, FORMATTED THE INPUT AND PASSED CONTROL TO SQL 1. SQL 1 PERFORMS VALIDATION ON THE SYSTEM DEPENDENT FIELDS (MAJOR SYSTEM, ACTION, OBJECT, SEARCH CRITERIA). IF ALL SYSTEM FIELDS ARE VALID SQL 1 PASSED CONTROL TO THIS MODULE. PASSED PARAMETERS CONSIST ONLY OF A POINTER WHICH POINTS TO A COMMUNICATION CONTROL AREA USED TO COMMUNICATE BETWEEN SQL 0, SQL 1, SQL 2 AND THE SECONDARY SELECTION AND DETAIL MODULES. * 01100000
* WHAT IS INCLUDED IN THIS MODULE?............. * 01150000
* ALL SECONDARY SELECTION AND DETAIL MODULES ARE 'INCLUDED'. * 01160000
* ALL VARIABLES KNOWN IN THIS PROCEDURE ARE KNOWN IN THE SUB PROCEDURES. ALL SQL CURSOR DEFINITIONS AND SQL 'INCLUDES' ARE DONE IN THIS PROCEDURE. BECAUSE OF THE RESTRICTION THAT CURSOR HOST VARIABLES MUST BE DECLARED BEFORE THE CURSOR DEFINITION ALL CURSOR HOST VARIABLES ARE DECLARED IN THIS PROCEDURE. * 01170000
* PROCEDURE * 01200000
* IF ANSWER TO DETAIL SCREEN & DETAIL PROCESSOR * 01250000
* IS NOT WILLING TO ACCEPT AN ANSWER THEN * 01260000
* NEW REQUEST* * 01270000
* ELSE IF ANSWER TO A SECONDARY SELECTION THEN * 01300000
* DETERMINE IF NEW REQUEST. * 01310000
* CASE (NEW REQUEST) * 01320000
* SUBCASE ('ADD') * 01350000
* DETAIL PROCESSOR * 01360000
* RETURN TO SQL 1 * 01370000
* ENDSUB * 01390000
* SUBCASE ('ERASE','DISPLAY','UPDATE') * 01400000
* CALL SECONDARY SELECTION * 01410000
* IF # OF POSSIBLE CHOICES IS ^= 1 THEN * 01420000
* RETURN TO SQL 1 * 01430000

Chapter 12. Sample data and applications supplied with Db2 for z/OS 1417
ELSE
   CALL THE DETAIL PROCESSOR
   RETURN TO SQL 1
ENDCASE

   IF ANSWER TO SECONDARY SELECTION AND A SELECTION HAS
   ACTUALLY BEEN MADE THEN
      VALID SELECTION #?
      IF IT IS VALID THEN
         CALL DETAIL PROCESSOR
         RETURN TO SQL 1
      ELSE
         PRINT ERROR MSG
         RETURN TO SQL 1
   IF ANSWER TO SECONDARY SELECTION THEN
      CALL SECONDARY SELECTION
      RETURN TO SQL 1.
   IF ANSWER TO DETAIL THEN
      CALL DETAIL PROCESSOR
      RETURN TO SQL 1.
   END.
*EXAMPLE- A ROW IS SUCCESSFULLY ADDED, THE OPERATOR RECEIVES*
*THE SUCCESSFULLY ADDED MESSAGE AND JUST HITS ENTER.
/***********************************************************/
/* DETERMINES WHETHER NEW REQUEST OR NOT */
/* NEW REQUEST. */
IF PCONVSTA.PREV = ' ' THEN
COMPARM.NEWREQ = 'Y';

/* IF ANSWER TO SECONDARY SELECTION THEN DETERMINE IF REALLY A */
/* NEW REQUEST. IT WILL BE CONSIDERED A NEW REQUEST IF POSITIONS*/
/* 3 TO 60 ARE NOT ALL BLANK AND THE ENTERED DATA IF NOT 'NEXT' */
IF COMPARM.NEWREQ = 'N' & PCONVSTA.PREV = 'S' &
SUBSTR(COMPARM.DATA,3,58) ^= ' ' &
COMPARM.PFKIN ^= '08'
THEN COMPARM.NEWREQ = 'Y';

/* IF NEW REQUEST AND ACTION IS 'ADD' THEN */
/* CALL DETAIL PROCESSOR */
/* ELSE CALL SECONDARY SELECTION */
IF COMPARM.NEWREQ='Y' THEN
DO;
IF COMPARM.ACTION = 'A' THEN
DO;
CALL DETAIL; /*CALL DETAIL PROCESSOR */
GO TO EXIT; /* RETURN */
END;
CALL SECSEL; /*CALL SECONDARY SELECTION*/
IF MAXSEL = 1 THEN /* IF NO. OF CHOICES = 1 */
CALL DETAIL; /* CALL DETAIL PROCESSOR */
GO TO EXIT; /* RETURN */
END;

/* IF ANSWER TO SECONDARY SELECTION AND NOT A SCROLLING REQUEST */
/* (INPUT NOT EQUAL TO 'NEXT') AND THE POSITIONS */
/* 1 TO 2 IN INPUT DATA FIELD NOT EQUAL TO POSITIONS 1 TO 2 */
/* IN OUTPUT DATA FIELD THEN SEE IF VALID SELECTION. */
IF PCONVSTA.PREV ^= 'S' THEN GO TO IP201; /* TO SECONDARY SEL */
IF PCONVSTA.MAXSEL < 1  THEN GO TO IP201; /* NO VALID CHOICES */
IF COMPARM.PFKIN = '08' THEN GO TO IP201; /* SCROLL REQUEST */
IF SUBSTR(COMPARM.DATA,1,2) = SUBSTR(PCONVSTA.DATA,1,2)
THEN GO TO IP201; /* NO CHANGE ON INPUT SCREEN */
IF SUBSTR(COMPARM.DATA,2,1) = '' THEN /* SECOND CHAR BLANK */
IF VERIFY(SUBSTR(COMPARM.DATA,1,1),''123456789'' ) = 0 THEN
DO;
SUBSTR(COMPARM.DATA,2,1) = SUBSTR(COMPARM.DATA,1,1);
END;
IF VERIFY(SUBSTR(COMPARM.DATA,1,2),''0123456789'' ) = 0 &
SUBSTR(COMPARM.DATA,1,2) > '00' THEN
DO;
COMPARM.NEWREQ = 'Y'; /*TELL DETAIL PROCESSOR NEW REQ */
CALL DETAIL; /* CALL DETAIL PROCESSOR*/
GO TO EXIT; /* RETURN*/
END;
/*INVALID SELECTION NO.*/
/*PRINT ERROR MESSAGE */
CALL DSN8MPG (MODULE, '072E', OUTMSG);
PCONVSTA.MSG= OUTMSG;
GO TO EXIT; /* RETURN */

*************************************************************************/
/* DETERMINES WHETHER SECONDARY SELECTION OR DETAIL */
*************************************************************************/
/* MUST BE ANY ANSWER TO EITHER SEC SEL OR DETAIL */

IF PCONVSTA.PREV = 'S' THEN
DO;
   CALL SECSEL; /*SECONDARY SELECTION*/
   GO TO EXIT; /* RETURN */
END;

IF PCONVSTA.PREV = 'D' THEN
DO;
   CALL DETAIL; /* DETAIL PROCESSOR */
   GO TO EXIT; /* RETURN */
END;

/* LOGIC ERROR */
CALL DSN8MPG (MODULE, '066E', OUTMSG);
PCONVSTA.MSG= OUTMSG; /* PRINT ERROR MESSAGE */
GO TO EXIT;

EXEC SQL INCLUDE DSN8MPXX; /* HANDLES SQL ERRORS */
GO TO EXIT;

*************************************************************************/
/* CALLS SECONDARY SELECTION AND RETURNS TO SQL 1 */
/* NOTE - SAME SECONDARY SELECTION MODULE FOR DS, DE AND EM */
*************************************************************************/

SECSEL: PROC; /* CALL APPROPRIATE SECONDARY SEL */
PCONVSTA.LASTSCR = 'DSN8001'; /* NOTE GENERAL MAP */

IF COMPARM.OBJFLD='DS' THEN
DO; /* ADMINISTRATIVE */
   CALL DSN8MPA; /* DEPARTMENT STRUCTURE */
   RETURN;
END;

IF COMPARM.OBJFLD='DE' THEN
DO; /* INDIVIDUAL DEPARTMENT */
   CALL DSN8MPA; /* PROCESSING */
   RETURN;
END;

IF COMPARM.OBJFLD='EM' THEN
DO; /* INDIVIDUAL EMPLOYEE */
   CALL DSN8MPA; /* PROCESSING */
   RETURN;
END;

CALL DSN8MPG (MODULE, '062E', OUTMSG);
PCONVSTA.MSG= OUTMSG; /* PRINT ERROR MESSAGE */
GO TO EXIT; /* RETURN */

END SECSEL;

*************************************************************************/
/* CALLS DETAIL PROCESSOR AND RETURNS TO SQL 1 */
*************************************************************************/

DETAIL: PROC; /* CALL APPROPRIATE DETAIL MODULE */
PCONVSTA.LASTSCR = 'DSN8002'; /* NOTE DETAIL MAP */

SELECT (COMPARM.OBJFLD);
WHEN('DS') CALL DSN8MPD; /* DEPARTMENT STRUCTURE */
WHEN('DE') CALL DSN8MPE; /* DEPARTMENT */
WHEN('EM') CALL DSN8MPF; /* EMPLOYEE */

/* MISSING DETAIL MODULE */
A set of Db2 sample applications run in the CICS environment.

**Related reference**

“Sample applications in CICS” on page 1392

A set of Db2 sample applications run in the CICS environment.

**DSN8CP6**

THIS MODULE ISSUES A CICS RECEIVE MAP TO RETRIEVE INPUT, CALLS DSN8CP7, AND ISSUES A CICS SEND MAP AFTER RETURNING.

```
DSN8CP6 : PROC OPTIONS (MAIN); 00010000
/******************** ****************************************************** 00020000
 * * MODULE NAME = DSN8CP6 00021000
 * * DESCRIPTIVE NAME = DB2 SAMPLE APPLICATION 00030000
 * * SUBSYSTEM INTERFACE MODULE 00040000
 * * CICS 00050000
 * * PL/I 00060000
 * * PROJECT APPLICATION 00070000
 * * LICENSED MATERIALS - PROPERTY OF IBM 5605-DB2 00080000
 * * (C) COPYRIGHT 1982, 2010 IBM CORP. ALL RIGHTS RESERVED. 00090000
 * * STATUS = VERSION 10 00100000
 * * FUNCTION = THIS MODULE ISSUES A CICS RECEIVE MAP TO RETRIEVE INPUT, CALLS DSN8CP7, AND ISSUES A CICS SEND MAP AFTER RETURNING. 00110000
 * * NOTES = 00120000
 * * 1. INITIALIZES ITSELF WHEN TERMINAL OPERATOR ENTER INPUT 00130000
 * * AFTER VIEWING THE SCREEN SENT BY THE PREVIOUS 00140000
 * * ITERATION OF THE PROGRAM. 00150000
 * * DEPENDENCIES = TWO CICS MAPS(DSECTS) ARE REQUIRED : 00160000
 * * DSN8MCMCE AND DSN8MCMEF. 00170000
 * * MODULES DSN8CP7 IS REQUIRED. 00180000
 * * DCLGEN STRUCTURE DSN8MPIC IS REQUIRED. 00190000
 * * INCLUDED PLI STRUCTURE DSN8MPCA IS REQUIRED. 00200000
 * * RESTRICTIONS = NONE 00210000
 * * MODULE TYPE = PL/I PROC OPTIONS (MAIN) 00220000
 * * PROCESSOR = DB2 PRECOMPILER, CICS TRANSlator, PL/I OPTIMIZER 00230000
 * * MODULE SIZE = SEE LINK-EDIT 00240000
 * * ATTRIBUTES = REUSABLE 00250000
 * * ENTRY POINT = DSN8CP6 00260000
 * * PURPOSE = SEE FUNCTION 00270000
 * * LINKAGE = NONE 00280000
 * * INPUT = PARAMETERS EXPLICITLY PASSED TO THIS FUNCTION: 00290000
 * * SYMBOlic LABEL/NAME = NONE 00300000
 * * DESCRIPTION = NONE 00310000
 * * OUTPUT = PARAMETERS EXPLICITLY RETURNED: 00320000
 * * SYMBOlic LABEL/NAME = NONE 00330000
 * * DESCRIPTION = NONE 00340000
EXIT: EXEC CICS RETURN; 00350000
EXEC SQL INCLUDE DSN8MPF; /* DETAIL - EMPLOYEES */ 00360000
EXEC SQL INCLUDE DSN8MPE; /* DETAIL - DEPARTMENTS */ 00370000
EXEC SQL INCLUDE DSN8MPD; /* DETAIL - ADMIN STRUCTURE */ 00380000
EXEC SQL INCLUDE DSN8MPA; /* SEC SEL - ADMIN STRUCTURE */ 00390000
END DSN8CP2; 00400000
END DSN8CP2; 00410000
```

Chapter 12. Sample data and applications supplied with Db2 for z/OS 1421
EXEC SQL INCLUDE DSN8MPME; /* 2ND MAP, BUILT FROM DSN8CPE */
EXEC SQL INCLUDE DSN8MPMF; /* 1ST MAP, BUILT FROM DSN8CPF */
EXEC SQL INCLUDE DSN8MPCA; /* COMMAREA */

EXEC SQL INCLUDE DSN8MPMF; /* CICS/OS/VS PL/I MAP, PROJECTS */
EXEC SQL INCLUDE DSN8MPME; /* CICS/OS/VS PL/I MAP, PROJECTS */

CONTROL-BLOCKS = SQLCA - SQL COMMUNICATION AREA

TABLES = NONE

CHANGE-ACTIVITY = NONE

* PSEUODOCODE *

PROCEDURE

DECLARATIONS.

ALLOCATE PL/I WORK AREA FOR COMMAREA.

PUT MODULE NAME 'DSN8CP6' IN AREA USED BY ERROR-HANDLER.

PUT CICS EIBTRMID IN PCONVSTA.CONVID TO BE PASSED TO DSN8CP7

RETRIEVE LASTSCR FROM VCONA USING THE CONVID TO DETERMINE WHICH OF THE TWO BMS MAPS SHOULD BE USED TO MAP IN DATA.

IF RETRIEVAL IS SUCCESSFUL, THEN DO.

EXEC CICS RECEIVE MAP ACCORDING TO SPECIFIED LASTSCR

IF MAPFAIL CONDITION IS RAISED* THEN DO.

/* COMPARM.PFKIN = '00' */

GO TO CP6CP7

END

ELSE

PUT DATA FROM MAP INTO COMPARM **

ELSE

IT IS A NEW CONVERSATION,

AND NO EXEC CICS RECEIVE MAP IS ISSUED.

EXEC CICS LINK PROGRAM('DSN8CP7') COMMAREA(COMMAREA).

UPON RETURN FROM DSN8CP7, EXEC CICS SEND MAP ACCORDING TO THE TYPE SPECIFIED IN PCONVSTA.LASTSCR.

EXEC CICS RETURN TRANSID(DBPP).

END.

I.E. LAST CONVERSATION EXISTS, BUT OPERATOR HAD ENTERED DATA FROM A CLEARED SCREEN OR HAD ERASED ALL DATA ON SCREEN AND PRESSED ENTER.

** COMPARM.PFKIN = PF KEY ACTUALLY USED I.E. '01' FOR PF1...

SQL0 CICS (DSN8CP6)

EXEC SQL INCLUDE DSN8MPCA;
EXEC SQL INCLUDE DSN8MPMF; /* 1ST MAP, BUILT FROM DSN8CPF */
EXEC SQL INCLUDE DSN8MPME; /* 2ND MAP, BUILT FROM DSN8CPE */
EXEC SQL INCLUDE SQLCA; /* SQL COMMUNICATION AREA */
EXEC SQL INCLUDE DSN8MPCS; /* PCONA */

0/*****************************************************************************/
/* SUBMAP REDEFINES THE PL/I STRUCTURE ASSOCIATED WITH THE */
/* CICS MAP DSNBCPE. */
0/*****************************************************************************/
0DCL MAP1PTR PTR,
MAP2PTR PTR;
DCL IOAREA AREA(2048);
0DCL 1 SUBMAP(15) BASED (ADDR(DSN8CPEI.LINE1F1L)) UNALIGNED,
2 COL1LEN FIXED BIN (15,0),
2 COL1ATTR CHAR (1),
2 COL1DATA CHAR (37),
2 COL2LEN FIXED BIN (15,0),
2 COL2ATTR CHAR (1),
2 COL2DATA CHAR (40);

0/*****************************************************************************/
/* PFSTRG IS AN ARRAY OF 24 ELEMENTS REPRESENTING THE DIFFERENT */
/* PFKEYS AS THEY WOULD BE REPRESENTED IN EIBAID. */
0/*****************************************************************************/
0DCL CONVID CHAR(16); 
DCL PFSTRG CHAR(24) INIT ('123456789:#@ABCDEFGHI>.<');

0/*****************************************************************************/
/* PFK IS AN ARRAY OF 12 TWO-BYTE CHARS REPRESENTING THE PFKEYS */
/* ALLOWED AS INPUT TO DSN8CP7 AND DSN8CP8 ETC. */
0/*****************************************************************************/
PFK(12) CHAR(2) INIT ('01','02','03','04','05','06,'
'07','08','09','10','11','12'), N FIXED BIN;

0/*****************************************************************************/
/* ** DCLGENS AND INITIALIZATIONS */
0/*****************************************************************************/
DCL STRING BUILTIN;
DCL J FIXED BIN;
DCL SAVE_CONVID CHAR(16);
/* DECLARE CONTROL FLAGS */
DCL ( SENDBIT, ENDBIT, NEXTBIT, ON, OFF) BIT(1);

0/*****************************************************************************/
/*          **  FIELDS SENT TO MESSAGE ROUTINE          */
0/*****************************************************************************/
DCL  MODULE            CHAR (07);
DCL  OUTMSG            CHAR (69);

DCL DSN8MPG EXTERNAL ENTRY;

0/*****************************************************************************/
/*   SQL RETURN CODE HANDLING */
0/*****************************************************************************/
EXEC SQL WHENEVER SQLERROR GO TO DB_ERROR;
EXEC SQL WHENEVER SQLWARNING GO TO DB_ERROR;

0/*****************************************************************************/
/*   ALLOCATE PL/I WORK AREA / INITIALIZE VARIABLES */
0/*****************************************************************************/
0ALLOCATE COMMAREA SET(COMMPTR); /*ALLOCATE COMMON AREA*/
MAP1PTR = ADDR(IOAREA); /* SET THE POINTER FOR THE GENERAL MAP */
MAP2PTR = ADDR(IOAREA); /* SET THE POINTER FOR THE DETAIL MAP */
COMMAREA = ''; /*CLEAR COMMON AREA*/
DSN8_MODULE_NAME.MAJOR = 'DSN8CP6'; /*GET MODULE NAME*/

CONVID,PCONVSTA.CONVID = EIBTRMID || '            ' ;
OUTAREA.MAJSYS = 'P'; /*SET MAJOR SYSTEM TO P-PROJECT*/
EXITCODE = '0'; /*CLEAR EXIT CODE*/

EXEC CICS HANDLE CONDITION MAPFAIL(CP6SEND);

0/*****************************************************************************/
/*  ALLOCATE PL/I WORK AREA / INITIALIZE VARIABLES */
0/*****************************************************************************/
0ALLOCATE COMMAREA SET(COMMPTR); /*ALLOCATE COMMON AREA*/
MAP1PTR = ADDR(IOAREA); /* SET THE POINTER FOR THE GENERAL MAP */
MAP2PTR = ADDR(IOAREA); /* SET THE POINTER FOR THE DETAIL MAP */
COMMAREA = ''; /*CLEAR COMMON AREA*/

DSN8_MODULE_NAME.MAJOR = 'DSN8CP6';

EXEC CICS HANDLE CONDITION MAPFAIL(CP6SEND);

0/*****************************************************************************/
/*  ALLOCATE PL/I WORK AREA / INITIALIZE VARIABLES */
0/*****************************************************************************/
/* LAST SCREEN SPECIFIED TO RECEIVE INPUT FROM TERMINAL. */
EXEC SQL SELECT LASTSCR INTO :PCONA.LASTSCR
FROM VCONA
WHERE CONVID = :CONVID ;

IF SQLCODE = +100 THEN GO TO CP6SEND;

IF PCONA.LASTSCR = 'DSN8001 ' THEN
DO;
EXEC CICS RECEIVE MAP ('DSN8CPF') MAPSET ('DSN8CPF') ;
IF AMAJSYSL ^= 0 THEN COMPARM.MAJSYS = AMAJSYSI;
ELSE COMPARM.MAJSYS = 'P';
IF AACTIONL ^= 0 THEN COMPARM.ACTION = AACTIONI;
ELSE COMPARM.ACTION = ' '; IF AOBJECTL ^= 0 THEN COMPARM.OBJFLD = AOBJECTI;
ELSE COMPARM.OBJFLD = ' '; IF ASEARCHL ^= 0 THEN COMPARM.SEARCH = ASEARCHI;
ELSE COMPARM.SEARCH = ' '; IF ADATAL ^= 0 THEN COMPARM.DATA = ADATAI;
ELSE COMPARM.DATA = ' ';
END;
ELSE IF PCONA.LASTSCR = 'DSN8002 ' THEN
DO;
EXEC CICS RECEIVE MAP ('DSN8CPE') MAPSET('DSN8CPE') ;
IF BMAJSYSL ^= 0 THEN COMPARM.MAJSYS = BMAJSYSI;
ELSE COMPARM.MAJSYS = 'P';
IF BACTIONL ^= 0 THEN COMPARM.ACTION = BACTIONI;
ELSE COMPARM.ACTION = ' '; IF BOBJECTL ^= 0 THEN COMPARM.OBJFLD = BOBJECTI;
ELSE COMPARM.OBJFLD = ' '; IF BSEARCHL ^= 0 THEN COMPARM.SEARCH = BSEARCHI;
ELSE COMPARM.SEARCH = ' '; IF BDATAL ^= 0 THEN COMPARM.DATA = BDATAI;
ELSE COMPARM.DATA = ' ';
END;
DO I = 1 TO 15;
IF SUBMAP.COL2LEN(I) ^= 0 THEN COMPARM.TRANDATA(I) = SUBMAP.COL2DATA(I);
ELSE COMPARM.TRANDATA(I) = ' ';
END;
END;
ELSE /* WRONG LASTSCREEN NAME*/
DO;
EXEC CICS ABEND ABCODE ('MAPI');
END;

N = INDEX ( PFSTRG , EIBAID ) ;
IF N ^= 0 THEN /* IF PF KEY USED */
DO;
END;
ELSE
EXEC CICS ABEND ABCODE ('MAPI');
END;

/* CONVERT THE PFKEY INFO IN EIBAID TO THE FORM ACCEPTED */
/* BY DSN8CP7 AND DSN8CP8 ETC. EG. PF1 = '01' AND PF13 = '01'. */
DO I = 1 TO 15;
IF SUBMAP.COL2LEN(I) ^= 0 THEN COMPARM.TRANDATA(I) = SUBMAP.COL2DATA(I);
ELSE COMPARM.TRANDATA(I) = ' ';
END;
END;
**Chapter 12. Sample data and applications supplied with Db2 for z/OS**

```assembly
COMPARM.PFKIN = PFK(N); 02990000
END; 03000000
ELSE COMPARM.PFKIN = ' '; /* IF ENTER | PAKEYS */ 03020000
GO TO CP6CP7; 03030000

********************************************************************
/*                                                                   */
/*    GO TO DSN8CP7, GET DCLGEN STRUCTURES AND TABLE DCL             */
/*                                                                   */
********************************************************************
0 EXEC CICS LINK PROGRAM ('DSN8CP7') COMMAREA(COMMAREA) LENGTH(3000); 03210000

0 EXEC SQL INCLUDE DSN8MPXX; /*GET DCLGEN STRUCTURES*/ 03220000

0 IF PCONVSTA.LASTSCR = 'DSN8001' THEN /*MOVE DATA INTO */ 03350000
DO;
/INPUT FIELDS */ 03360000
ATITLEO = PCONVSTA.TITLE ; 03370000
AMJYSO= PCONVSTA.MAJSYS; 03380000
AACTIONO= PCONVSTA.ACTION; 03390000
AOBJECETO= PCONVSTA.OBJFLD; 03400000
ASEARCHO= PCONVSTA.SEARCH; 03410000
ADATAO = PCONVSTA.DATA ; 03420000
AMSGO = PCONVSTA.MSG ; 03430000
ADESCL2O= PCONVSTA.DESC2 ; 03440000
ADESCL3O= PCONVSTA.DESC3 ; 03450000
ADESCL4O= PCONVSTA.DESC4 ; 03460000
APFKEYO = PCONVSTA.PFKTEXT; 03470000

DO I = 1 TO 15; /*SEND MAP ACCORDING TO*/ 03490000
ALINEO(I) = PCONVSTA.OUTPUT.LINE(I); /*PREVIOUS SCREEN*/ 03500000
END; 03510000

0 IF CURSOR_VALUE = 0 THEN /*SEND OUTPUT MAP */ 03530000
EXEC CICS SEND MAP('DSN8CPF') MAPSET('DSN8CPF'); 03540000
ELSE /*ERASE CURSOR*/ 03550000
EXEC CICS SEND MAP('DSN8CPF') MAPSET('DSN8CPF') ERASE CURSOR(CURSOR_VALUE); 03560000

0 IF CURSOR_VALUE = 0 THEN /*SEND OUTPUT MAP */ 03570000
EXEC CICS SEND MAP('DSN8CPF') MAPSET('DSN8CPF'); 03580000
ELSE /*ERASE CURSOR*/ 03590000
EXEC CICS SEND MAP('DSN8CPF') MAPSET('DSN8CPF') ERASE CURSOR(CURSOR_VALUE); 03600000
```

---

// Chapter 12. Sample data and applications supplied with Db2 for z/OS 1425
IF EXITCODE = '1' THEN /*FINISHED ? */
EXEC CICS RETURN; /* RETURN, DON'T REINVOKE TRANSACTION*/
ELSE
EXEC CICS RETURN TRANSID('DBPP'); /* STANDARD RETURN */
END;

0/*********************************************************************/
/* MOVES DATA FROM OUTPUT MAP AREA TO */
/* RECEIVE MAP ACCORDING TO MAP SPECIFIED IN LASTSCR OF PCONVST */
/*********************************************************************/

0 ELSE IF PCONVSTA.LASTSCR = 'DSN8002 ' THEN /*FROM OUTPUT FIELDS*/
DO;
    BITITLEO = PCONVSTA.TITLE ;
    BMJYSYO= PCONVSTA.MJYSYS;
    BACTIOMO= PCONVSTA.ACTION;
    * BOBJECTO= PCONVSTA.OBJFLD; 
    BSEARCHO= PCONVSTA.SEARCH;
    BDATAO  = PCONVSTA.DATA ; 
    BMSGO = PCONVSTA.MSG ;
    BDESC1O= PCONVSTA.DESC1 ;
    BDESC3O= PCONVSTA.DESC3 ;
    BDESC4O= PCONVSTA.DESC4 ;
    BPFKEYO = PCONVSTA.PFKTEXT;

    DO I = 1 TO 15 ; /*RECEIVE MAP ACCORDING TO */
    SUBMAP.COL1DATA(I) = REOUT.FIELD1(I); /*PREVIOUS SCREEN */
END;

0 /*-----------------------------------------------------------------*/
/* MODULES DSN8MPE, DSN8MPF ETC. IN SQL2 HAVE PUT THE */
/* ATTRIBUTE BYTE AND CURSOR CONTROL INFO IN IMS MFS */
/* FORM - HEX'C0' FOR DYNAMIC CURSOR WITH 2 BYTES OF */
/* ATTRIBUTE INFORMATION TO FOLLOW. THIS PROGRAM CHECKS */
/* FOR THE HEX'C0' AND INSERTS -1 INTO */
/* THE LENGTH FIELD ASSOCIATED WITH THE DATA TO CONFORM */
/* WITH THE STANDARD WAY OF HANDLING DYNAMIC CURSORS IN */
/* CICS. SIMILARLY, ONLY THE SECOND OF THE TWO ATTRIBUTE */
/* BYTES IS MOVED INTO THE CICS ATTRIBUTE BYTE. THE */
/* FIRST TWO BITS OF THE ATTRIBUTE BYTE IS DIFFERENT */
/* BETWEEN IMS AND CICS STANDARD REPRESENTATIONS, HOWEVER */
/* 3270 MANUALS INDICATE THAT ON OUTPUT, THE FIRST */
/* TWO BITS ARE IGNORED. THUS THE SAME ATTRIBUTE BYTE */
/* IS USED BETWEEN IMS AND CICS MODULES. */
/* /*-----------------------------------------------------------------*/

0 IF UNSPEC(REOUT.ATTR1(I)) = '11000000'B /* X'C0' ATTR */
THEN SUBMAP.COL2LEN(I) = -1; 
SUBMAP.COL2ATTR(I) = REOUT.ATTR2(I); 
SUBMAP.COL2DATA(I) = REOUT.FIELD2(I); 
END;

0/*********************************************************************/
/* CREATES A DYNAMIC CURSOR */
/*********************************************************************/

0 IF CURSOR_VALUE = 0 THEN 
EXEC CICS SEND MAP('DSN8CPE') MAPSET('DSN8CPE');
ELSE
    CURSOR_VALUE = 0; /*CLEAR CURSOR*/
    IF BACTIOMO = ' ' THEN /*SET CURSOR POSITION */
        CURSOR_VALUE = 179; /*ACTION POSITION*/
    ELSE
        IF BOBJECTO = ' ' THEN /*SET CURSOR TO*/
            CURSOR_VALUE = 259; /*OBJECT POSITION*/
        ELSE
            IF BSEARCHO = ' ' THEN /*SET CURSOR TO*/
                CURSOR_VALUE = 339; /*SEARCH POSITION*/
            ELSE
                IF BDATAO = ' ' | /*SET CURSOR TO*/
                    (BACTIOMO = 'D' | /*DATA POSITION*/
                    BACTIOMO = 'U' | 
                    BACTIOMO = 'A' | 
                    BACTIOMO = 'E') THEN 
                    CURSOR_VALUE = 419;
            END;
        END;
    END;
    IF CURSOR_VALUE = 0 THEN /*SEND INPUT MAP */
EXEC CICS SEND MAP('DSN8CPE') MAPSET('DSN8CPE');
ELSE
    EXEC CICS SEND MAP('DSN8CPE') MAPSET('DSN8CPE');
END;
EXEC CICS SEND MAP('DSN8CPE') MAPSET('DSN8CPE') ERASE 04630000
CURSOR(CURSOR_VALUE); 04640000
04650000
04660000
IF EXITCODE = '1' THEN /*FINISHED ? */ 04670000
EXEC CICS RETURN; /* RETURN, DON'T REINVOKE TRANSACTION */ 04680000
ELSE 04690000
EXEC CICS RETURN TRANSID('D8PP'); /* STANDARD RETURN */ 04700000
END; 04710000
END; 04720000
0 /* SQL1 DID NOT PASS BACK VALID LASTSCREEN NAME */ 04730000
ELSE EXEC CICS ABEND ABCODE ('MAPO'); 04740000
END; 04750000

Related reference
“Sample applications in CICS” on page 1392
A set of Db2 sample applications run in the CICS environment.

DSN8CP7
THIS MODULE PERFORMS THE INCLUDES TO BRING IN THE SQL TABLE DCLS AND DCLGEN STRUCTURES AS WELL AS THE PARAMETER AREA.

DSN8CP7:PROC (COMMPTR) OPTIONS(MAIN);
    /*********************************************************************
    *                                                                    *
    *   MODULE NAME = DSN8CP7                                            *
    *                                                                    *
    *   DESCRIPTIVE NAME = DB2 SAMPLE APPLICATION                         *
    *   SQL 1 MAINLINE                                                    *
    *   CICS                                                               *
    *   PL/I                                                               *
    *   PROJECT APPLICATION                                               *
    *                                                                    *
    *    COPYRIGHT = 5740-XYR (C) COPYRIGHT IBM CORP 1982, 1985          *
    *    REFER TO COPYRIGHT INSTRUCTIONS FORM NUMBER G120-2083           *
    *                                                                    *
    *    STATUS = RELEASE 2, LEVEL 0                                      *
    *                                                                    *
    *    FUNCTION = THIS MODULE PERFORMS THE INCLUDES TO BRING IN THE     *
    *              SQL TABLE DCLS AND DCLGEN STRUCTURES AS WELL AS       *
    *              THE PARAMETER AREA.                                    *
    *    INCLUDE DSN8MP1.                                                 *
    *    CALL DSN8CP8.                                                    *
    *    RETURN TO DSN8CP6.                                               *
    *                                                                    *
    *    NOTES =                                                          *
    *       DEPENDENCIES = CALLED BY DSN8CP6, CALLS DSN8CP8 (CICS LINKS). *
    *       RESTRICTIONS = NONE                                          *
    *                                                                    *
    *    MODULE TYPE = PL/I PROC(COMMPTR) OPTIONS.                        *
    *       PROCESSOR = DB2 PRECOMPILED, CICS TRANSLATOR, PL/I OPTIMIZER *
    *       MODULE SIZE = SEE LINK-EDIT                                   *
    *       ATTRIBUTES = REUSABLE                                         *
    *                                                                    *
    *    ENTRY POINT = DSN8CP7                                           *
    *    PURPOSE = SEE FUNCTION                                          *
    *    LINKAGE = NONE                                                   *
    *                                                                    *
    *    INPUT = PARAMETERS EXPLICITLY PASSED TO THIS FUNCTION:          *
    *       SYMBOLIC LABEL/NAME = COMMPTR (POINTER TO COMMAREA)          *
    *       DESCRIPTION = NONE                                           *
    *                                                                    *
    *    OUTPUT = PARAMETERS EXPLICITLY RETURNED:                        *
    *       SYMBOLIC LABEL/NAME = NONE                                   *
    *       DESCRIPTION = NONE                                           *
    *                                                                    *
    *    EXIT-NORMAL = DSN8CP6                                           *
    *                                                                    *
    *    EXIT-ERROR = DSN8CP6                                            *
    *    RETURN CODE = NONE                                              *
    *    ABEND CODES = NONE                                              *
    *                                                                    *

Chapter 12. Sample data and applications supplied with Db2 for z/OS 1427
**ERROR-MESSAGES = NONE**

**EXTERNAL REFERENCES =**

**ROUTINES/SERVICES = DSNBCP8**

**DATA-AREAS =**

- **DSN8MPCA** - PLI STRUCTURE FOR COMMAREA
- **DSN8MPCS** - DECLARE CONVERSATION STATUS
- **DSN8MPV0** - DECLARE OPTION VALIDATION
- **DSN8MPVO** - FIND VALID OPTIONS FOR ACTION, OBJECT, SEARCH CRITERIA
- **DSN8MP1** - RETRIEVE LAST CONVERSATION, VALIDATE, CALL SQL2
- **DSN8MP3 -- DSN8MP5** - VALIDATION MODULES CALLED BY DSN8MP1
- **DSN8MPXX** - SQL ERROR HANDLER

**CONTROL-BLOCKS =**

- **SQLCA** - SQL COMMUNICATION AREA

**TABLES = NONE**

**CHANGE-ACTIVITY = NONE**

---

**PSEUDOCODE**

**PROCEDURE**

* INCLUDE DECLARATIONS. *
* INCLUDE DSN8MP1. *
* INCLUDE ERROR HANDLER. *
* CPIEXIT: (REFERRED BY DSN8MP1) *
  * EXEC CICS RETURN. *
* CP1CALL: (REFERRED BY DSN8MP1) *
  * EXEC CICS LINK PROGRAM('DSN8CP8') COMMAREA(COMMAREA) *
  * LENGTH(3000). *
  * GO TO MP1SAVE. (LABEL IN DSN8MP1) *
* * INCLUDE VALIDATION MODULES. *
* END. *

---

**SQL1 MAINLINE**

EXEC SQL WHENEVER SQLERROR GO TO DB_ERROR;
EXEC SQL WHENEVER SQLWARNING GO TO DB_ERROR;

-----------

**DECLGENS AND INITIALIZATIONS**

DCL STRING BUILTIN;
DCL J FIXED BIN;
DCL SAVE_CONVID CHAR(16);

/* DECLARE CONTROL FLAGS */
DCL (SENDBIT, ENDBIT, NEXTBIT, ON, OFF) BIT(1);

/* FIELDS SENT TO MESSAGE ROUTINE */
DCL MODULE CHAR (07);
DCL OUTMSG CHAR (69);

DCL DSN8MPG EXTERNAL ENTRY;
EXEC SQL INCLUDE DSN8MPCA; /* INCLUDE COMMAREA */
DSN8_MODULE_NAME.MAJOR = 'DSN8CP7'; /* INITIALIZE MODULE NAME */
EXEC SQL INCLUDE DSN8MPCS; /* INCLUDE PCONA */
EXEC SQL INCLUDE DSN8MPV0; /* INCLUDE POPTVAL */
EXEC SQL INCLUDE DSN8MPVO; /* INCLUDE CURSOR */
EXEC SQL INCLUDE SQLCA; /* INCLUDE SQL COMMAREA */
EXEC SQL INCLUDE DSN8MP1; /* INCLUDE SQL1 MAIN */
EXEC SQL INCLUDE DSN8MPXX; /* INCLUDE ERRORHANDLER */

CP1EXIT :
  EXEC CICS RETURN; /* STANDARD EXIT */
CP1CALL :
  EXEC CICS LINK PROGRAM('DSN8CP8') COMMAREA(COMMAREA) LENGTH(3000);
  GO TO MP1SAVE;
EXEC SQL INCLUDE DSN8MP3; /* INCLUDE ACTION VALIDATION */
EXEC SQL INCLUDE DSN8MP4; /* INCLUDE OBJECT VALIDATION */
EXEC SQL INCLUDE DSN8MP5; /* INCLUDE SEARCH CRITERIA */
END; /* VALIDATION */

Related reference
“Sample applications in CICS” on page 1392
A set of Db2 sample applications run in the CICS environment.

**DSN8CP8**

ROUTER FOR SECONDARY SELECTION AND/OR DETAIL PROCESSING CALLS SECONDARY SELECTION MODULES DSN88MPM CALLS DETAIL MODULES DSN88MPT DSN88MPV DSN88MPW DSN88MPX DSN88MPZ CALLED BY DSN88CP7 (SQL1).

---

Chapter 12. Sample data and applications supplied with Db2 for z/OS 1429
SUB PROCEDURES. ALL SQL CURSOR DEFINITIONS AND
ALL VARIABLES KNOWN IN THIS PROCEDURE ARE KNOWN IN THE
ALL SECONDARY SELECTION AND DETAIL MODULES ARE 'INCLUDED'.

WHAT IS INCLUDED IN THIS MODULE?.............

AND DETAIL MODULES.                               
BETWEEN SQL 0 , SQL 1, SQL 2 AND THE SECONDARY SELECTION
POINTS TO A COMMUNICATION CONTROL AREA USED TO COMMUNICATE
MODULE. PASSED PARAMETERS CONSIST ONLY OF A POINTER WHICH
FIELDS (MAJOR SYSTEM, ACTION, OBJECT, SEARCH CRITERIA). IF
TO SQL 1. SQL 1 PERFORMS VALIDATION ON THE SYSTEM DEPENDENT
DEPENDENT MODULE (IMS,CICS,TSO) OR (SQL 0) HAS
WHAT HAS HAPPENED SO FAR?.............THE SUBSYSTEM
CONTROL-BLOCKS =
SQLCA               - SQL COMMUNICATION AREA

DATA-AREAS =
DSNBMPAC - DCLGEN FOR ACTIVITY TYPES  * 00710000
DSNBMPAS - CURSOR SECONDARY SELECTION FOR  * 00720000
STAFF
DSNBMPCA - COMMUNICATION AREA BETWEEN MODULES  * 00730000
DSNBMPDH - CURSOR FOR DISPLAY TEXT FROM  * 00750000
TDSPTXT TABLE
DSNBMPDP - DCLGEN FOR DEPARTMENT  * 00770000
DSNBMPDT - DCLGEN FOR DISPLAY TABLE  * 00780000
DSNBMPM - DCLGEN FOR EMPLOYEE  * 00790000
DSNBMPPE - DCLGEN FOR PROJECT/STAFFING  * 00800000
DSNBMPES - CURSOR SECONDARY SELECTION FOR  * 00810000
ESTIMATES
DSNBMPOV - DCLGEN FOR OPTION VALIDATION  * 00830000
DSNBMPPA - DCLGEN FOR PROJECT/ACTIVITIES  * 00840000
DSNBMPPD - DCLGEN FOR PROJ STRUCTURE DETAIL  * 00850000
DSNBMPPE - CURSOR PROJECT DETAIL  * 00860000
DSNBMPPL - DCLGEN FOR PROJECTS  * 00870000
DSNBMPPE - CURSOR PROJECT LIST  * 00880000
DSNBMPPR - DCLGEN FOR PROJ/RESP EMPLOYEE  * 00890000
DSNBMPSA - DCLGEN FOR PROJ ACTIVITY LISTING  * 00900000
DSNBMPSL - CURSOR STAFFING LIST  * 00910000
DSNBMPSC - DCLGEN FOR PROJECT/STAFFING  * 00920000
DSNBMPXT - ACTIVITY ESTIMATE DETAIL  * 00940000
DSNBMPXT - DCLGEN FOR DEPARTMENT  * 00950000
DSNBMPM - SECONDARY SELECTION FOR PROJECTS  * 00960000
DSNBMPPT - PROJECT ACTIVITY LIST  * 00970000
DSNBMPX - PROJECT STRUCTURE DETAIL  * 00980000
DSNBMPX - DCLGEN FOR PROJECT/ACTIVITIES  * 00990000
DSNBMPZ2 - DCLGEN FOR PROJ STRUCTURE DETAIL  * 01000000
DSNBMPZ - PROJECT DETAIL  * 01010000

CONTROL-BLOCKS =

SQLCA               - SQL COMMUNICATION AREA

DATA-AREAS =
DSNBMPAC - DCLGEN FOR ACTIVITY TYPES  * 00710000
DSNBMPAS - CURSOR SECONDARY SELECTION FOR  * 00720000
STAFF
DSNBMPCA - COMMUNICATION AREA BETWEEN MODULES  * 00730000
DSNBMPDH - CURSOR FOR DISPLAY TEXT FROM  * 00750000
TDSPTXT TABLE
DSNBMPDP - DCLGEN FOR DEPARTMENT  * 00770000
DSNBMPDT - DCLGEN FOR DISPLAY TABLE  * 00780000
DSNBMPM - DCLGEN FOR EMPLOYEE  * 00790000
DSNBMPPE - DCLGEN FOR PROJECT/STAFFING  * 00800000
DSNBMPES - CURSOR SECONDARY SELECTION FOR  * 00810000
ESTIMATES
DSNBMPOV - DCLGEN FOR OPTION VALIDATION  * 00830000
DSNBMPPA - DCLGEN FOR PROJECT/ACTIVITIES  * 00840000
DSNBMPPD - DCLGEN FOR PROJ STRUCTURE DETAIL  * 00850000
DSNBMPPE - CURSOR PROJECT DETAIL  * 00860000
DSNBMPPL - DCLGEN FOR PROJECTS  * 00870000
DSNBMPPE - CURSOR PROJECT LIST  * 00880000
DSNBMPPR - DCLGEN FOR PROJ/RESP EMPLOYEE  * 00890000
DSNBMPSA - DCLGEN FOR PROJ ACTIVITY LISTING  * 00900000
DSNBMPSL - CURSOR STAFFING LIST  * 00910000
DSNBMPSC - DCLGEN FOR PROJECT/STAFFING  * 00920000
DSNBMPXT - ACTIVITY ESTIMATE DETAIL  * 00940000
DSNBMPXT - DCLGEN FOR DEPARTMENT  * 00950000
DSNBMPM - SECONDARY SELECTION FOR PROJECTS  * 00960000
DSNBMPPT - PROJECT ACTIVITY LIST  * 00970000
DSNBMPX - PROJECT STRUCTURE DETAIL  * 00980000
DSNBMPZ2 - DCLGEN FOR PROJ STRUCTURE DETAIL  * 00990000
DSNBMPZ - PROJECT DETAIL  * 01000000

CONTROL-BLOCKS =

SQLCA               - SQL COMMUNICATION AREA

DATA-AREAS =
DSNBMPAC - DCLGEN FOR ACTIVITY TYPES  * 00710000
DSNBMPAS - CURSOR SECONDARY SELECTION FOR  * 00720000
STAFF
DSNBMPCA - COMMUNICATION AREA BETWEEN MODULES  * 00730000
DSNBMPDH - CURSOR FOR DISPLAY TEXT FROM  * 00750000
TDSPTXT TABLE
DSNBMPDP - DCLGEN FOR DEPARTMENT  * 00770000
DSNBMPDT - DCLGEN FOR DISPLAY TABLE  * 00780000
DSNBMPM - DCLGEN FOR EMPLOYEE  * 00790000
DSNBMPPE - DCLGEN FOR PROJECT/STAFFING  * 00800000
DSNBMPES - CURSOR SECONDARY SELECTION FOR  * 00810000
ESTIMATES
DSNBMPOV - DCLGEN FOR OPTION VALIDATION  * 00830000
DSNBMPPA - DCLGEN FOR PROJECT/ACTIVITIES  * 00840000
DSNBMPPD - DCLGEN FOR PROJ STRUCTURE DETAIL  * 00850000
DSNBMPPE - CURSOR PROJECT DETAIL  * 00860000
DSNBMPPL - DCLGEN FOR PROJECTS  * 00870000
DSNBMPPE - CURSOR PROJECT LIST  * 00880000
DSNBMPPR - DCLGEN FOR PROJ/RESP EMPLOYEE  * 00890000
DSNBMPSA - DCLGEN FOR PROJ ACTIVITY LISTING  * 00900000
DSNBMPSL - CURSOR STAFFING LIST  * 00910000
DSNBMPSC - DCLGEN FOR PROJECT/STAFFING  * 00920000
DSNBMPXT - ACTIVITY ESTIMATE DETAIL  * 00940000
DSNBMPXT - DCLGEN FOR DEPARTMENT  * 00950000
DSNBMPM - SECONDARY SELECTION FOR PROJECTS  * 00960000
DSNBMPPT - PROJECT ACTIVITY LIST  * 00970000
DSNBMPX - PROJECT STRUCTURE DETAIL  * 00980000
DSNBMPZ2 - DCLGEN FOR PROJ STRUCTURE DETAIL  * 00990000
DSNBMPZ - PROJECT DETAIL  * 01000000

WHAT HAS HAPPENED SO FAR?.............THE SUBSYSTEM
DEPENDENT MODULE (IMS,CICS,TSO) OR (SQL 0) HAS
READ THE INPUT SCREEN, FORMATTED THE INPUT AND PASSED CONTROL
TO SQL 1. SQL 1 PERFORMS VALIDATION ON THE SYSTEM DEPENDENT
FIELDS (MAJOR SYSTEM, ACTION, OBJECT, SEARCH CRITERIA). IF
ALL SYSTEM FIELDS ARE VALID SQL 1 PASSED CONTROL TO THIS
MODULE. PASSED PARAMETERS CONSIST ONLY OF A POINTER WHICH
POINTS TO A COMMUNICATION CONTROL AREA USED TO COMMUNICATE
BETWEEN SQL 0 , SQL 1, SQL 2 AND THE SECONDARY SELECTION
AND DETAIL MODULES.

WHAT IS INCLUDED IN THIS MODULE?.............
ALL SECONDARY SELECTION AND DETAIL MODULES ARE 'INCLUDED'.
ALL VARIABLES KNOWN IN THIS PROCEDURE ARE KNOWN IN THE
SUB PROCEDURES. ALL SQL CURSOR DEFINITIONS AND
SQL 'INCLUDES' ARE DONE IN THIS PROCEDURE. BECAUSE OF THE
Chapter 12. Sample data and applications supplied with Db2 for z/OS 1431
EXEC SQL INCLUDE DSN8MPPE; /* MAJSYS P - DETAIL FOR PS */ 02110000
EXEC SQL INCLUDE DSN8MPPSL; /* MAJSYS P - DETAIL FOR AL */ 02120000
EXEC SQL INCLUDE DSN8MPDH; /* PROG TABLES - DISPLAY HEADINGS */ 02130000
02140000
EXEC SQL WHENEVER SQLERROR GO TO DB_ERROR;                         02150000
EXEC SQL WHENEVER SQLWARNING GO TO DB_ERROR;                         02160000
02170000
DCL UNSPEC BUILTIN;                                                  02180000
DCL VERIFY BUILTIN;                                                  02190000
02200000
DCL LENGTH BUILTIN;                                                  02210000
DCL DSN8MPG EXTERNAL ENTRY;                                          02220000
02230000
DCL STRING BUILTIN;                                                    02240000
DCL J FIXED BIN;                                                       02250000
DCL SAVE_CONVID CHAR(16);                                              02260000
/* DECLARE CONTROL FLAGS */    02270000
DCL ( SENDBIT, ENDBIT, NEXTBIT, ON, OFF) BIT(1);                       02280000
02290000
DCL MODULE        CHAR (07) INIT('DSN8CP8');                        02300000
DCL OUTMSG        CHAR (69);                                        02310000
02320000
DSN8_MODULE_NAME.MAJOR='DSN8CP8';                                    02330000
DSN8_MODULE_NAME.MINOR=' ';                                          02340000
02350000
IF PCONVSTA.PREV = ' ' THEN                                          02360000
COMPARM.NEWREQ = 'Y';                                              02370000
02380000
IF COMPARM.NEWREQ = 'N' & PCONVSTA.PREV = 'S' &                      02390000
SUBSTR(COMPARM.DATA,3,58) ^= ' ' &                                02400000
COMPARM.DATA ^= 'NEXT'                                            02410000
THEN COMPARM.NEWREQ = 'Y';                                        02420000
02430000
IF COMPARM.NEWREQ='Y' THEN                                           02440000
DO;                                                                02450000
    IF COMPARM.ACTION = 'A' THEN                                     02460000
        DO;                                                            02470000
            CALL DETAIL;                      /* CALL DETAIL PROCESSOR*/ 02480000
            GO TO EXIT;                       /* RETURN */               02490000
        END;                                                           02500000
        CALL SECSEL;                 /* CALL SECONDARY SELECTION */      02510000
        IF MAXSEL = 1 THEN           /* IF NO. OF CHOICES = 1    */      02520000
            CALL DETAIL;               /* CALL DETAIL PROCESSOR    */      02530000
            GO TO EXIT;                  /* RETURN                   */      02540000
        END;                                                               02550000
        CALL SECSEL;                 /* CALL SECONDARY SELECTION */      02560000
    END;                                                           02570000
    IF COMPARM.ACTION = 'A' THEN                                     02580000
        DO;                                                            02590000
            CALL DETAIL;                      /* CALL DETAIL PROCESSOR*/ 02600000
            GO TO EXIT;                       /* RETURN */               02610000
        END;                                                           02620000
        CALL SECSEL;                 /* CALL SECONDARY SELECTION */      02630000
    END;                                                               02640000
END;                                                               02650000
/* IF ANSWER TO SECONDARY SELECTION AND NOT A SCROLLING REQUEST */    02660000
02670000
/* (INPUT NOT EQUAL TO 'NEXT') AND THE POSITIONS */
/* 1 TO 2 IN INPUT DATA FIELD NOT EQUAL TO POSITIONS 1 TO 2 */
/* IN OUTPUT DATA FIELD THEN SEE IF VALID SELECTION. */

/***********************************************************/
/* DETERMINES IF VALID SELECTION NUMBER */
/***********************************************************/

IF PCONVSTA.PREV ^= 'S' THEN GO TO IP201; /* TO SECONDARY SEL */
IF PCONVSTA.MAXSEL < 1 THEN GO TO IP201; /* NO VALID CHOICES */
IF COMPARM.DATA = 'NEXT' THEN GO TO IP201; /* SCROLL REQUEST */
IF SUBSTR(COMPARM.DATA,1,2) = SUBSTR(PCONVSTA.DATA,1,2)
   THEN GO TO IP201;  /* NO CHANGE ON INPUT SCREEN */
IF SUBSTR(COMPARM.DATA,2,1) = ' ' THEN     /* SECOND CHAR BLANK */
   DO;
      IF VERIFY(SUBSTR(COMPARM.DATA,1,1),'123456789') = 0 THEN
         DO;
            SUBSTR(COMPARM.DATA,2,1) = SUBSTR(COMPARM.DATA,1,1);
            SUBSTR(COMPARM.DATA,1,1) = '0';
         END;
      END;
   END;
   IF VERIFY(SUBSTR(COMPARM.DATA,1,2),'0123456789') = 0 &
      SUBSTR(COMPARM.DATA,1,2) > '00' THEN
      IF DATAP <= PCONVSTA.MAXSEL THEN
         DO;
            COMPARM.NEWREQ = 'Y'; /* TELL DETAIL PROCESSOR NEW REQ */
            CALL DETAIL;          /* CALL DETAIL PROCESSOR */
            GO TO EXIT;           /*      RETURN                   */
         END;
      END;
   END;
   IF IF PCONVSTA.PREV = 'S' THEN
      DO;
         CALL SECSEL;                     /* CALL SECONDARY SELECTION*/
         GO TO EXIT;                      /* RETURN */
      END;
   END;
   IF PCONVSTA.PREV = 'D' THEN
      DO;
         CALL DETAIL;                     /* CALL DETAIL PROCESSOR */
         GO TO EXIT;                      /* RETURN */
      END;
   END;
   /* LOGIC ERROR */               /* PRINT ERROR MESSAGE */
   CALL DSN8MPG (MODULE, '066E', OUTMSG);
   PCONVSTA.MSG= OUTMSG;
   PCONVSTA.PREV = ' ';                /* NOT SEC SELECTION, ERROR */
   GO TO EXIT;                         /* RETURN */
/* CALLS SECONDARY SELECTION AND RETURNS TO SQL 1 */
/* NOTE - SAME SECONDARY SELECTION MODULE FOR DS, DE AND EM */
SECSEL: PROC; /* CALL APPROPRIATE SECONDARY SELECTION MODULE */
   PCONVSTA.LASTSCR = 'DSN8001';  /* SET FOR GENERAL MAP */
   IF COMPARM.OBJFLD='AE' |      /* ACTIVITY ESTIMATE */
      COMPARM.OBJFLD='AL' |      /* PROJECT ACTIVITY LISTING */
      COMPARM.OBJFLD='AS' |      /* INDIVIDUAL PROJECT STAFFING*/
      COMPARM.OBJFLD='PR' |      /* INDIVIDUAL PROJECT PROCESSING*/
      COMPARM.OBJFLD='PS' THEN  /* PROJECT STRUCTURE */
      DO;
/* EXEC SQL INCLUDE DSN8MPXX;*/
/* HANDLES SQL ERRORS */
   EXEC SQL INCLUDE DSN8MPXX;
   /***************************************************/
   /* CALLS SECONDARY SELECTION AND RETURNS TO SQL 1 */
   /***************************************************/
   IF IF PCONVSTA.PREV = 'S' THEN
      DO;
         CALL SECSEL;                     /* CALL SECONDARY SELECTION*/
         GO TO EXIT;                      /* RETURN */
      END;
   END;
   IF PCONVSTA.PREV = 'D' THEN
      DO;
         CALL DETAIL;                     /* CALL DETAIL PROCESSOR */
         GO TO EXIT;                      /* RETURN */
      END;
   END;
   /* LOGIC ERROR */               /* PRINT ERROR MESSAGE */
   CALL DSN8MPG (MODULE, '066E', OUTMSG);
   PCONVSTA.MSG= OUTMSG;
   PCONVSTA.PREV = ' ';                /* NOT SEC SELECTION, ERROR */
   GO TO EXIT;                         /* RETURN */
/* EXEC SQL INCLUDE DSN8MPXX;*/
/* HANDLES SQL ERRORS */
   EXEC SQL INCLUDE DSN8MPXX;
   /***************************************************/
   /* CALLS SECONDARY SELECTION AND RETURNS TO SQL 1 */
   /***************************************************/
   IF IF PCONVSTA.PREV = 'S' THEN
      DO;
         CALL SECSEL;                     /* CALL SECONDARY SELECTION*/
         GO TO EXIT;                      /* RETURN */
      END;
   END;
   IF PCONVSTA.PREV = 'D' THEN
      DO;
         CALL DETAIL;                     /* CALL DETAIL PROCESSOR */
         GO TO EXIT;                      /* RETURN */
      END;
   END;
   /* LOGIC ERROR */               /* PRINT ERROR MESSAGE */
   CALL DSN8MPG (MODULE, '066E', OUTMSG);
   PCONVSTA.MSG= OUTMSG;
   PCONVSTA.PREV = ' ';                /* NOT SEC SELECTION, ERROR */
   GO TO EXIT;                         /* RETURN */
CALL DSN8MPM; /*SECONDARY SELECTION FOR PROJECTS*/ 03750000 RETURN; /*RETURN */ 03760000 END; 03770000 /*MISSING SECONDARY SEL*/ 03780000 CALL DSN8MPG (MODULE, '062E', OUTMSG); 03790000 PCONVSTA.MSG= OUTMSG; 03800000 PCONVSTA.PREV = ' '; /* NOT SEC SELECTION, ERROR */ 03810000 GO TO EXIT; /*RETURN */ 03820000 END SECSEL; 03830000 /*MISSING DETAIL MODULE*/ 03840000 /*PRINT ERROR MESSAGE*/ 03850000 CALL DSN8MPG (MODULE, '062E', OUTMSG); 03860000 PCONVSTA.MSG= OUTMSG; 03870000 PCONVSTA.PREV = ' '; /* NOT SEC SELECTION, ERROR */ 03880000 GO TO EXIT; /*RETURN */ 03890000 END DETAIL; 03900000 /*RETURNS TO SQL 1*/ 03910000 EXIT: EXEC CICS RETURN; /* PROJECTS */ 03920000 EXEC SQL INCLUDE DSN8MPM; /* SEC SEL - PROJECTS */ 03930000 EXEC SQL INCLUDE DSN8MPT; /* DETAIL - PROJ ACT LISTING*/ 03940000 EXEC SQL INCLUDE DSN8MPV; /* DETAIL - PROJ STRUCTURE */ 03950000 EXEC SQL INCLUDE DSN8MPW; /* DETAIL - INDIVID STAFFING*/ 03960000 EXEC SQL INCLUDE DSN8MPX; /* DETAIL - INDIVID ACTIVITY*/ 03970000 EXEC SQL INCLUDE DSN8MPZ; /* DETAIL - INDIVIDUAL PROJ */ 03980000 END DSN8CP8; 03990000 Related reference
"Sample applications in CICS" on page 1392
A set of Db2 sample applications run in the CICS environment.

**DSN8CP3**

THIS MODULE LISTS EMPLOYEE PHONE NUMBERS AND UPDATES THEM IF DESIRED.
Chapter 12. Sample data and applications supplied with Db2 for z/OS 1435
* IN_MESSAGE - VIA BMS, SEE INPUT PARAMETERS *
* OUT_MESSAGE - VIA BMS, SEE OUTPUT PARAMETERS *
* DSN8MPML - DECLARE FOR DSN8CPCL CICS MAP *
* DSN8MPMN - DECLARE FOR DSN8CPN CICS MAP *
* DSN8MPMU - DECLARE FOR DSN8CPU CICS MAP *
* CONTROL-BLOCKS = *
  * SQLCA - SQL COMMUNICATION AREA *
* TABLES = NONE *
* *
* CHANGE-ACTIVITY = *
* PQ92146 09/07/04 CHANGE DECLARED LENGTH OF BMS_IO FROM 32767 @01*
  * TO 1408 TO STOP IBM2402I COMPIL-TIME ERROR  @01 *
* *
* *PSEUDOCODE* *
* *
* PROCEDURE *
* GET FIRST INPUT *
* DO WHILE MORE INPUT *
* GET REPORT HEADING *
* *
* CASE (ACTION) *
* *
  SUBCASE ('L') *
  IF LASTNAME IS '*' *
  LIST ALL EMPLOYEES *
  ELSE *
  IF LASTNAME CONTAINS '%' *
  LIST EMPLOYEES GENERIC *
  ELSE *
  LIST EMPLOYEES SPECIFIC *
  ENDSUB *
* *
  SUBCASE ('U') *
  UPDATE PHONENUMBER FOR EMPLOYEE *
  WRITE CONFIRMATION MESSAGE *
  OTHERWISE *
  INVALID REQUEST *
  ENDSUB *
* *
  GET NEXT INPUT *
* ENDCASE *
* *
  IF SQL ERROR OCCURS THEN *
  FORMAT ERROR MESSAGE *
  ROLLBACK *
  END *
* END. *
*-------------------------------------------------------------------*
1/*********************************************************************/
/*      DECLARATION FOR INPUT / OUTPUT                              */
/*********************************************************************/
EXEC SQL INCLUDE DSN8MPMN ;
EXEC SQL INCLUDE DSN8MPML ;
EXEC SQL INCLUDE DSN8MPMU ;
ODCL 1 SUBMAPI(15)  UNALIGNED BASED(ADDR(DSN8CU2I.NEWNO1L)),
  2 NEWNOL FIXED BIN(15,0),
  2 NEWNOA CHAR(1),
  2 NEWNOD CHAR(4),
  2 ENOL FIXED BIN(15,0),
  2 ENOA CHAR(1),
  2 ENOD CHAR(6);
ODCL 1 SUBMAPO(15) UNALIGNED BASED(ADDR(DSN8CL2I.FNAME1L)),
  2 FNAMEL FIXED BIN(15,0),
  2 FNAMOA CHAR(1),
  2 FNAMED CHAR(12),
  2 MINITL FIXED BIN(15,0),
  2 MINITA CHAR(1),
  2 MINITD CHAR(1),
  2 LNAMEL FIXED BIN(15,0),
  2 LNAMEA CHAR(1),
  2 LNAMED CHAR(15),
2 PNOL FIXED BIN(15,0),
2 PNOA CHAR(1),
2 PNOD CHAR(4),
2 ENOL FIXED BIN(15,0),
2 ENOA CHAR(1),
2 ENOD CHAR(6),
2 WDEPTL FIXED BIN(15,0),
2 WDEPTA CHAR(1),
2 WDEPTD CHAR(3),
2 WNAMEL FIXED BIN(15,0),
2 WNAMAEA CHAR(1),
2 WNAME(31);

/***** HOLDS BYTE-COUNT OF STORAGE ALLOCATED TO BMS OUTPUT AREA *****/
DCL BMS_LL BIN FIXED( 31 ) INIT( STG(DSN8CL2I) ); /*!<EDVG*/

/******* MASK/OVERLAY OF STORAGE ALLOCATED TO BMS OUTPUT AREA *******/
DCL BMS_IO CHAR( 1408 ) BASED( ADDR(DSN8CL2I) ); /*@01*/

1/*********************************************************************/
/*       DECLARATION FOR PGM-LOGIC                                  */
1/*********************************************************************/
DCL FIRST BIT(1);
DCL PAGING BIT(1);
DCL OFLOW BIT(1);
DCL EMPLOYEE_NO CHAR (6);
DCL PHONE_NO CHAR (4);
DCL CHAR_SQLCODE CHAR (14);
DCL 1 CHAR_SQLSTR BASED(ADDR(CHAR_SQLCODE)),
2 CHAR_BLNK CHAR(4),
2 CHAR_SQLCOD CHAR(10);

1//****************************************************************************
/*  FIELDS SENT TO MESSAGE ROUTINE                                     */
1//****************************************************************************
DCL MODULE CHAR (7) INIT('DSN8CP3');
DCL OUTMSG CHAR (69);
DCL DSN8MPG EXTERNAL ENTRY;

1/*********************************************************************/
/*                       DECLARATION FOR SQL                         */
1/*********************************************************************/
EXEC SQL INCLUDE SQLCA; /* SQL COMMUNICATION AREA          */
/* SQL DECLARATION FOR VIEW PHONE */
EXEC SQL DECLARE VPHONE TABLE
(LASTNAME VARCHAR(15) ,
FIRSTNAME VARCHAR(12) ,
MIDDLEINITIAL CHAR(1) ,
PHONENUMBER CHAR(4) ,
EMPLOYEENUMBER CHAR(6) ,
DEPTNUMBER CHAR(3) NOT NULL,
DEPTNAME VARCHAR(36) NOT NULL);
/* STRUCTURE FOR PHONE RECORD */
DCL 1 PPHONE,
2 LASTNAME CHAR (15) VAR,
2 FIRSTNAME CHAR (12) VAR,
2 MIDDLEINITIAL CHAR (1),
2 PHONENUMBER CHAR (4),
2 EMPLOYEENUMBER CHAR (6),
2 DEPTNUMBER CHAR (3),
2 DEPTNAME CHAR (36) VAR;
/* SQL DECLARATION FOR VIEW VEMPLP*/
EXEC SQL DECLARE VEMPLP TABLE
(EMPLOYEENUMBER CHAR(6) ,
PHONENUMBER CHAR(4));
/* STRUCTURE FOR VEMPLP RECORD */
DCL 1 PEMP,
2 EMPLOYEENUMBER CHAR (6),
2 PHONENUMBER CHAR (4);

1/***************************************************************************/
/* SQL CURSORS */
1/***************************************************************************/
EXEC SQL DECLARE TELE1 CURSOR FOR
SELECT *
FROM VPHONE;
EXEC SQL DECLARE TELE2 CURSOR FOR
SELECT *
FROM VPHONE
WHERE LASTNAME LIKE :DSN8CN2I.LNAMEI

Chapter 12. Sample data and applications supplied with Db2 for z/OS 1437
AND FIRSTNAME LIKE :DSN8CN2I.FNAMEI;
EXEC SQL DECLARE TELE3 CURSOR FOR
SELECT *
FROM VPHONE
WHERE LASTNAME  = :DSN8CN2I.LNAMEI
AND FIRSTNAME LIKE :DSN8CN2I.FNAMEI;

EXEC SQL WHENEVER SQLERROR   GOTO P3_DBERROR;
EXEC SQL WHENEVER SQLWARNING GOTO P3_DBERROR;
EXEC SQL WHENEVER NOT FOUND  CONTINUE;

EXEC CICS RECEIVE MAP('DSN8CN2') MAPSET('DSN8CPN');
1/********************************************************************/ /*               SQL RETURN CODE HANDLING                           */ /********************************************************************/ EXEC SQL WHENEVER SQLERROR   GOTO P3_DBERROR;
EXEC SQL WHENEVER SQLWARNING GOTO P3_DBERROR;
EXEC SQL WHENEVER NOT FOUND  CONTINUE;
1/********************************************************************/ /*                 MAIN PROGRAM ROUTINE                             */ /********************************************************************/ /* SET HANDLE CONDITIONS  */ EXEC CICS HANDLE CONDITION MAPFAIL (P3_MAPFAIL);
EXEC CICS HANDLE AID CLEAR (P3_CLEAR);

SUBSTR(BMS_IO,1,BMS_LL) = LOW(BMS_LL) ; /*@EDVG*/
P3_START:
FIRST = '1'B;                            /*INITIALIZE FIRST BIT */
OFLOW = '0'B;                          /*INITIALIZE OVERFLOW BIT*/

SELECT (EIBTRNID);                      /* SELECT ACTION */
WHEN ('D8PT') DO;                     /* LIST EMPLOYEES */
/* GET INPUT FROM SCREEN */
EXEC CICS RECEIVE MAP('DSN8CN2') MAPSET('DSN8CPN');
1/********************************************************************/ /*                 LIST ALL EMPLOYEES                               */ /********************************************************************/ IF DSN8CN2I.LNAMEI = '*' THEN      /*LIST ALL EMPLOYEES */
DO;
EXEC SQL OPEN TELE1;                /* OPEN CURSOR    */
EXEC SQL FETCH TELE1             /* GET FIRST RECORD  */
INTO :PPHONE;
I = 0;                          /* INITIALIZE COUNTER */
IF SQLCODE = 100 THEN            /* NO EMPLOYEE FOUND */
DO;                           /* PRINT ERROR MESSAGE */
call DSN8MPG (MODULE, '008I', OUTMSG);
DSN8CN3I.EMSGI = OUTMSG;
EXEC CICS SEND MAP('DSN8CN3') MAPSET('DSN8CPN') ERASE;
EXEC CICS SEND MAP('DSN8CN2') MAPSET('DSN8CPN');
END;
DO WHILE (SQLCODE = 0);                 /*LIST EMPLOYEES*/
I = I + 1;                        /* INCREMENT COUNTER*/
SUBMAPO.FNAMED(I) = PPHONE.FIRSTNAME;
SUBMAPO.MINITD(I) = PPHONE.MIDDLEINITIAL;
SUBMAPO.LNAMED(I) = PPHONE.LASTNAME;
SUBMAPO.PNOD(I) = PPHONE.PHONENUMBER;
SUBMAPO.ENOD(I) = PPHONE.EMPLOYEENUMBER;
SUBMAPO.WDEPTD(I) = PPHONE.DEPTNUMBER;
SUBMAPO.WNAMED(I) = PPHONE.DEPTNAME;
IF I = 15 THEN                 /*POSSIBLE OVERFLOW */
DO;                           /* PRINT ERROR MESSAGE*/
OFLOW = '1'B;
call DSN8MPG (MODULE, '057I', OUTMSG);
DSN8CL3O.EMSGO = OUTMSG;
END;
IF I = 15 THEN LEAVE;           /* SCREEN IS FILLED */
EXEC SQL FETCH TELE1           /* GET NEXT RECORD   */
INTO :PPHONE;
END;                             /* END OF WHILE      */
EXEC SQL CLOSE TELE1; /* CLOSE CURSOR */
END;
/* END OF IF */
1/**************************************************************************/

/* LIST GENERIC EMPLOYEES */
**************************************************************************/
ELSE
/* SELECT EMPLOYEES BY NAME*/
DO;
/* SEARCH ON PART OF NAME? */
IF DSN8CN2I.LNAMEL = 0 THEN /* IS LAST NAME BLANK? */
DSN8CN2I.LNAMEI = '%%%%%%%%%%%%%%%%'; /* YES, ANYTHING */
IF INDEX(DSN8CN2I.LNAMEI,'%') > 0 THEN /* IS IT A PATTERN*/
DO; /* YES, SEARCH ON */
/* PART OF LAST NAME */
DSN8CN2I.LNAMEI = TRANSLATE(DSN8CN2I.LNAMEI,'%',' ');
/*AND OPTIONALLY FIRST NAME*/
IF DSN8CN2I.FNAMEL = 0 THEN
DSN8CN2I.FNAMEI = '%%%%%%%%%%%%%%%%';
ELSE
DSN8CN2I.FNAMEI = TRANSLATE(DSN8CN2I.FNAMEI,'%',' ');
EXEC SQL OPEN TELE2; /* OPEN CURSOR */
EXEC SQL FETCH TELE2 /* GET FIRST RECORD */
INTO :PPHONE;
I = 0; /* INITIALIZE COUNTER */
IF SQLCODE = 100 THEN /* EMPLOYEE NOT FOUND */
DO; /* PRINT ERROR MESSAGE */
CALL DSN8MPG (MODULE, '008I', OUTMSG);
DSN8CN3I.EMSGI = OUTMSG;
EXEC CICS SEND MAP('DSN8CN3') MAPSET('DSN8CPN') ERASE;
EXEC CICS SEND MAP('DSN8CN2') MAPSET('DSN8CPN');
END;
DO WHILE (SQLCODE = 0); /* LIST EMPLOYEES */
I = I + 1; /* INCREMENT COUNTER */
PAGING = '1'B;
SUBMAPO.FNAMED(I) = PPHONE.FIRSTNAME;
SUBMAPO.MINITD(I) = PPHONE.MIDDLEINITIAL;
SUBMAPO.LNAMED(I) = PPHONE.LASTNAME;
SUBMAPO.PNOD(I) = PPHONE.PHONENUMBER;
SUBMAPO.ENOD(I) = PPHONE.EMPLOYEE_NUMBER;
SUBMAPO.WDEPTD(I) = PPHONE.DEPT_NUMBER;
SUBMAPO.WNAMED(I) = PPHONE.DEPT_NAME;
IF I = 15 THEN /*POSSIBLE OVERFLOW*/
DO; /* PRINT ERROR MESSAGE*/
OFLOW = '1'B;
CALL DSN8MPG (MODULE, '057I', OUTMSG);
DSN8CL3O.EMSGO = OUTMSG;
END;
IF I = 15 THEN LEAVE; /* SCREEN IS FILLED */
EXEC SQL FETCH TELE2 /* GET NEXT RECORD */
INTO :PPHONE;
END; /* END OF DO WHILE */
EXEC SQL CLOSE TELE2; /* CLOSE CURSOR */
END; /* END OF IF */
1/**************************************************************************/

/* LIST SPECIFIC EMPLOYEE(S) */
**************************************************************************/
ELSE
/* SEARCH ON LAST NAME */
DO; /*AND OPTIONALLY FIRST NAME*/
IF DSN8CN2I.FNAMEL = 0 THEN
DSN8CN2I.FNAMEI = '%%%%%%%%%%%%%%%%';
ELSE
DSN8CN2I.FNAMEI = TRANSLATE(DSN8CN2I.FNAMEI,'%',' ');
EXEC SQL OPEN TELE3; /* OPEN CURSOR */
EXEC SQL FETCH TELE3 /* GET FIRST RECORD */
INTO :PPHONE;
I = 0; /* INITIALIZE COUNTER */
IF SQLCODE = 100 THEN /* EMPLOYEE NOT FOUND */
DO;                            /* PRINT ERROR MESSAGE */
CALL DSN8MPG (MODULE, '008I', OUTMSG);
DSNBCN3I.EMSGI = OUTMSG;
EXEC CICS SEND MAP('DSNBCN3') MAPSET('DSNBCP') ERASE;
EXEC CICS SEND MAP('DSNBCN2') MAPSET('DSNBCP');
END;
DO WHILE (SQLCODE = 0);       /* LIST EMPLOYEE(S) */
  I = I + 1;                 /* INCREMENT COUNTER */
  PAGING = '1'B;
  SUBMAPO.FNAMED(I) = PPHONE.FIRSTNAME;
  SUBMAPO.MINITD(I) = PPHONE.MIDDLEINITIAL;
  SUBMAPO.LNAMED(I) = PPHONE.LASTNAME;
  SUBMAPO.PNOD(I) = PPHONE.PHONENUMBER;
  SUBMAPO.ENOD(I) = PPHONE.EMPLOYEEENUMBER;
  SUBMAPO.WDEPTD(I) = PPHONE.DEPTNUMBER;
  SUBMAPO.WNAMED(I) = PPHONE.DEPTNAME;
IF I = 15 THEN                 /*POSSIBLE OVERFLOW */
  DO;                           /* PRINT ERROR MESSAGE*/
    OFLOW = '1'B;
    CALL DSN8MPG (MODULE, '057I', OUTMSG);
    DSN8CL3O.EMSGO = OUTMSG;
  END;
IF I = 15 THEN LEAVE;     /* SCREEN IS FILLED       */
EXEC SQL FETCH TELE3     /*   GET NEXT RECORD       */
  INTO :PPHONE;
END;                        /* END OF DO WHILE         */
EXEC SQL CLOSE TELE3;       /* CLOSE CURSOR        */
END;                          /* END OF ELSE         */
END;                            /* END OF IF           */
IF PAGING THEN
  DO;
    PAGING = '0'B;
    EXEC CICS SEND MAP ('DSNBC1') MAPSET('DSNBCP') ERASE
      ACCUM PAGING;
    EXEC CICS SEND MAP ('DSNBC2') MAPSET('DSNBCP')
      ACCUM PAGING;
  IF OFLOW THEN
    DO;
      OFLOW = '0'B;
      EXEC CICS SEND MAP ('DSNBC3') MAPSET('DSNBCP')
        ACCUM PAGING;
  END;
  EXEC CICS SEND PAGE;
  EXEC CICS RETURN TRANSID('D8PU');
END;                            /* END OF IF    */
ELSE EXEC CICS RETURN TRANSID ('D8PT');
END;                           /* END OF WHEN  */
/* CHANGE ERROR HANDLING   */
/* FOR UPDATE              */
EXEC SQL WHENEVER SQLERROR CONTINUE;
EXEC SQL WHENEVER SQLWARNING CONTINUE;
1/********************************************************************/
/*             UPDATES PHONE NUMBERS FOR EMPLOYEES                  */
/********************************************************************)
WHEN ('D8PU') DO;                   /* TELEPHONE UPDATE        */
/* GET UPDATED DATA */
EXEC CICS RECEIVE MAP('DSNBCU2') MAPSET('DSNBCP');
/* FIND WHICH NUMBERS HAVE */
/* BEEN UPDATED */
DSNBCN3I.EMSGI = '';              /* SET IN CASE NO UPDATES  */
DO I = 1 TO 15;
  IF SUBMAPI.NEWNOL(I) = 0 THEN;  /* NO UPDATE ON THIS LINE */
ELSE
  DO;
    EMPLOYEE_NO = SUBMAPI.ENOD(I);
    PHONE_NO = SUBMAPI.NEWNOD(I);
EXEC SQL UPDATE VEMPLP         /*     PERFORM UPDATE     */
    SET PHONENUMBER = :PHONE_NO
      WHERE EMPLOYEEENUMBER = :EMPLOYEE_NO;
IF SQLCODE ^= 0 THEN
   /* UPDATE FAILED */
   CALL DSN8MPG (MODULE, '007E', OUTMSG);
   DSN8CU3I.EMSGI = OUTMSG;
   EXEC CICS SEND MAP('DSN8CU3') MAPSET('DSN8CPU');
   GOTO P3_DBERROR2;
END;
   /* UPDATE SUCCESSFUL*/
   /* PRINT CONFIRMATION */
   DO;
   CALL DSN8MPG (MODULE, '004I', OUTMSG);
   DSN8CN3I.EMSGI = OUTMSG;
   END;
   /* END ELSE */
   END;
   /* END FOR */
EXEC CICS SEND MAP('DSN8CN3') MAPSET('DSN8CPN') ERASE;
EXEC CICS SEND MAP('DSN8CN2') MAPSET('DSN8CPN');
EXEC CICS RETURN TRANSID('D8PT');
END;
/* UPDATE SUCCESSFUL*/
/* PRINT CONFIRMATION */
ELSE
   /* MESSAGE */
   DO;
   CALL DSN8MPG (MODULE, '004I', OUTMSG);
   DSN8CN3I.EMSGI = OUTMSG;
   END;
   /* END ELSE */
   END;
   /* END FOR */
EXEC CICS SEND MAP('DSN8CN3') MAPSET('DSN8CPN') ERASE;
EXEC CICS SEND MAP('DSN8CN2') MAPSET('DSN8CPN');
EXEC CICS RETURN TRANSID('D8PT');
P3_MAPFAIL:
   / DBPT FROM UNFORMATTED /*
   / SCREEN */
   / MAP ONLY */
EXEC CICS SEND MAP('DSN8CN2') MAPONLY MAPSET('DSN8CPN') ERASE;
EXEC CICS RETURN TRANSID('D8PT');
P3_DBERROR:
   /* SQL ERROR HANDLING */
   CALL DSN8MPG (MODULE, '060E', OUTMSG);
   CHAR_SQLCODE = SQLCODE;
   DSN8CN3I.EMSGI = OUTMSG||CHAR_SQLCODE;
EXEC CICS SEND MAP('DSN8CN3') MAPSET('DSN8CPN') ERASE;
EXEC CICS RETURN TRANSID('D8PT');
P3_DBERROR2:
   EXEC CICS SEND PAGE ; /* PERFORM ROLLBACK */
   EXEC CICS SYNCPPOINT ROLLBACK;
   EXEC CICS RETURN;
P3_CLEAR:
   / CLEAR SCREEN */
   EXEC CICS SEND CONTROL FREEKB ;
   EXEC CICS RETURN;
P3_END:
   /* PROGRAM END */
   END DSN8CP3;
END;

Related reference
“Sample applications in CICS” on page 1392
A set of Db2 sample applications run in the CICS environment.

DSNTEJ5C
THIS JCL PERFORMS THE PHASE 5 SETUP FOR THE SAMPLE APPLICATIONS AT SITES WITH COBOL.

********************************************************************
/* SQL ERROR HANDLING */
P3_DBERROR: /* SQL ERROR HANDLING */
   CALL DSN8MPG (MODULE, '060E', OUTMSG);
   CHAR_SQLCODE = SQLCODE;
   DSN8CN3I.EMSGI = OUTMSG||CHAR_SQLCODE;
EXEC CICS SEND MAP('DSN8CN3') MAPSET('DSN8CPN');
P3_DBERROR2: /* PERFORM ROLLBACK */
   EXEC CICS SEND PAGE ;
   EXEC CICS SYNCPPOINT ROLLBACK;
   EXEC CICS RETURN;
P3_CLEAR: /* CLEAR SCREEN */
   EXEC CICS SEND CONTROL FREEKB ;
   EXEC CICS RETURN;
P3_END: /* PROGRAM END */
   END DSN8CP3;

Chapter 12. Sample data and applications supplied with Db2 for z/OS
//* 08/18/2014 Single-phase migration s21938_inst1 s21938
//*
FLICTE ADTECTION

//JOBLIB DD DSN=DSN!:0.SDSNXEIT,DISP=SHR
// DD DSN=DSN!:0.SDSNLLOAD,DISP=SHR
// DD DSN=CICS.SDFHLOAD,DISP=SHR
/**
 * STEP 1: CREATE CICS LOGICAL MAP
*/
//MAPG EXEC DFHASMVS,PARM='DECK,NOOBJECT,SYSPARM(DSECT)',
//OUTC='*'
*/
//SYSPUNCH DD DSN=DSN!:0.SRCLIB.DATA(DSN8MCMG),
//DISP=OLD
//SYSIN DD DSN=DSN!:0.SDSNSAMP(DSNBCCG),
//DISP=SHR
/**
 * STEP 2: CREATE CICS LOGICAL MAP
*/
//MAPD EXEC DFHASMVS,PARM='DECK,NOOBJECT,SYSPARM(DSECT)',
//COND=(4,LT),OUTC='*'
*/
//SYSPUNCH DD DSN=DSN!:0.SRCLIB.DATA(DSN8MCMMD),
//DISP=OLD
//SYSIN DD DSN=DSN!:0.SDSNSAMP(DSNBCCD),
//DISP=SHR
/**
 * STEP 3: PREPARE CICS COBOL PROGRAMS
*/
//DSNH EXEC PGM=IKJEFT01,COND=(4,LT),DYNAMNBR=50
//SYSTSPRT DD SYSOUT=* 
//SYSTERM DD SYSOUT=* 
//SYSUDUMP DD SYSOUT=* 
//SYSOUT DD SYSOUT=* 
//SYSPROC DD DSN=DSN!:0.SDSNCLST,DISP=SHR
//SYSSIN DD * 
TIM%DSNH INPUT('DSN!:0.SDSNSAMP(DSNBCCD)'),
PLIB('DSN!:0.SRCLIB.DATA'),
P2LIB('DSN!:0.SDSNSAMP'),
TERM(LEAVE) PRINT(LEAVE) SOURCE(NO) XREF(NO) +
HOST(IBMCOB) RUN(CICS) BIND(NO) +
DELIM(APOST) SQLDELIM(APOSTSQL) +
DBRMLIB('DSN!:0.DBRMLIB.DATA') +
PRELINK(YES) +
LLIB('DSN!:0.RUNLIB.LOAD') +
COBICOMP('IGY.V!R!M!.SIGYCOMP(IGYCRCTL)') +
COPTION('NOSEQUENCE,QUOTE,RENT,PGMNAME(LONGUPPER)') +
LOPTION('LIST,XREF,MAP,RENT') +
STDSQL(NO) +
XLIB('DSN!:0.SDSNLLOAD') +
LOAD('DSN!:0.RUNLIB.LOAD')
%DSNH INPUT('DSN!:0.SDSNSAMP(DSNBCC1)'),
PLIB('DSN!:0.SRCLIB.DATA'),
P2LIB('DSN!:0.SDSNSAMP'),
TERM(LEAVE) PRINT(LEAVE) SOURCE(NO) XREF(NO) +
HOST(IBMCOB) RUN(CICS) BIND(NO) +
DELIM(APOST) SQLDELIM(APOSTSQL) +
DBRMLIB('DSN!:0.DBRMLIB.DATA') +
PRELINK(YES) +
LLIB('DSN!:0.RUNLIB.LOAD') +
COBICOMP('IGY.V!R!M!.SIGYCOMP(IGYCRCTL)') +
COPTION('NOSEQUENCE,QUOTE,RENT,PGMNAME(LONGUPPER)') +
LOPTION('LIST,XREF,MAP,RENT') +
STDSQL(NO) +
XLIB('DSN!:0.SDSNLLOAD') +
LOAD('DSN!:0.RUNLIB.LOAD')
%DSNH INPUT('DSN!:0.SDSNSAMP(DSNBCC2)'),
PLIB('DSN!:0.SRCLIB.DATA'),
P2LIB('DSN!:0.SDSNSAMP'),
TERM(LEAVE) PRINT(LEAVE) SOURCE(NO) XREF(NO) +
HOST(IBMCOB) RUN(CICS) BIND(NO) +
DELIM(APOST) SQLDELIM(APOSTSQL) +
DBRMLIB('DSN!:0.DBRMLIB.DATA') +
PRELINK(YES) +
LLIB('DSN!:0.RUNLIB.LOAD') +
COBICOMP('IGY.V!R!M!.SIGYCOMP(IGYCRCTL)') +
COPTION('NOSEQUENCE,QUOTE,RENT,PGMNAME(LONGUPPER)') +
LOPTION('LIST,XREF,MAP,RENT') +
STDSQL(NO) +
XLIB('DSN!:0.SDSNLLOAD') +
LOAD('DSN!:0.RUNLIB.LOAD')
/*
 * STEP 4: BIND THE PROGRAM
*/
//BIND EXEC PGM=IKJEFT01,DYNAMNBR=20,COND=(4,LT)
//DBRMLIB DD DSN=DSN!:0.DBRMLIB.DATA,DISP=SHR
//SYSDUMP DD SYSOUT=*
//SYSTSPRT DD SYSOUT=*  
//SYSPRINT DD SYSOUT=*  
//SYSin DD *
//   SET CURRENT SQLID = 'SYSADM';
//   GRANT BIND, EXECUTE ON PLAN DSN8CC0
//   TO PUBLIC;
//SYSSIN DD *
DSN SYSTEM(DSN)
BIND PACKAGE (DSN8CC!!) MEMBER(DSN8CC0) APPLCOMPAT(V!!R1) +
   ACT(REP) ISO(CS) CURRENTDATA(YES) ENCODING(EBCDIC)
BIND PACKAGE (DSN8CC!!) MEMBER(DSN8CC1) APPLCOMPAT(V!!R1) +
   ACT(REP) ISO(CS) CURRENTDATA(YES) ENCODING(EBCDIC)
BIND PACKAGE (DSN8CC!!) MEMBER(DSN8CC2) APPLCOMPAT(V!!R1) +
   ACT(REP) ISO(CS) CURRENTDATA(YES) ENCODING(EBCDIC)
BIND PLAN(DSN8CC0) PKLIST(DSN8CC!!.*) -
   ACTION(REPLACE) RETAIN +
   ISO(CS) CURRENTDATA(YES) ENCODING(EBCDIC)
RUN PROGRAM(DSNTIAD) PLAN(DSNTIA!!) -
   LIB('DSN!!0.RUNLIB.LOAD')
END  
/*
/*        STEP  5: CREATE THE CICS BMS PHYSICAL MAP
/*MAPGP  EXEC DFHASMVS,COND=(4,LT),OUTC='*
//SYSPUNCH DD DSN=&TEMP,
//   DISP=(,PASS),
//   UNIT=SYSDA,SPACE=(1024,(100,10)),
//   DCB=(RECFM=F,BLKSIZE=80)
//SYSin DD DSN=DSN!!0.SDSNSAMP(DSN8CCG),
//   DISP=SHR
/*
/*        STEP  6: LINKEDIT THE CICS BMS PHYSICAL MAP
/*MAPGL  EXEC PGM=IEWL,PARM='LIST,LET,XREF',COND=(4,LT)
//SYSU1 DD UNIT=SYSDA,SPACE=(1024,(100,10))
//SYSLMOD DD DSN=DSN!!0.RUNLIB.LOAD,DISP=SHR
//SYSPRINT DD SYSOUT=*  
//SYSSDUMP DD SYSOUT=*  
//SYSLIN DD DSN=&TEMP,DISP=(OLD,DELETE)
//  DD *
//NAME DSN8CCG(R)
/*
/*        STEP  7: CREATE THE CICS BMS PHYSICAL MAP
/*MAPDP  EXEC DFHASMVS,COND=(4,LT),OUTC='*
//SYSPUNCH DD DSN=&TEMP,
//   DISP=(,PASS),
//   UNIT=SYSDA,SPACE=(1024,(100,10)),
//   DCB=(RECFM=F,BLKSIZE=80)
//SYSin DD DSN=DSN!!0.SDSNSAMP(DSN8CCD),
//   DISP=SHR
/*
/*        STEP  8: LINKEDIT THE CICS BMS PHYSICAL MAP
/*MAPDL  EXEC PGM=IEWL,PARM='LIST,LET,XREF',COND=(4,LT)
//SYSU1 DD UNIT=SYSDA,SPACE=(1024,(100,10))
//SYSLMOD DD DSN=DSN!!0.RUNLIB.LOAD,DISP=SHR
//SYSPRINT DD SYSOUT=*  
//SYSSDUMP DD SYSOUT=*  
//SYSLIN DD DSN=&TEMP,DISP=(OLD,DELETE)
//  DD *
//NAME DSN8CCD(R)

Related reference
“Sample applications in CICS” on page 1392
A set of Db2 sample applications run in the CICS environment.

DSNTEJ5P

THIS JCL PERFORMS THE PHASE 5 SETUP FOR THE SAMPLE APPLICATIONS AT SITES WITH PL/I.
STATUS = Version 12

FUNCTION = THIS JCL PERFORMS THE PHASE 5 SETUP FOR THE SAMPLE APPLICATIONS AT SITES WITH PL/I. IT PREPARES THE PL/I CICS PROGRAM.

RUN THIS JOB ANYTIME AFTER PHASE 2.

CHANGE ACTIVITY = 08/18/2014 Single-phase migration s21938 inst1 s21938

*********************************************************************

//JOBLIB DD DISP=SHR,DSN=CICSTS.SDFHLOAD
// DD DISP=SHR,DSN=DSN!!0.SDSNLOAD

//STEP  1: CREATE CICS BMS LOGICAL MAPS
//PH05PS01 EXEC DFHASMVS,PARM='DECK,NOOBJECT,SYSPARM(DSECT)',
//           OUTC='*'
//SYSPUNCH DD DSN=DSN!!0.SRCLIB.DATA(DSN8MPMG),
//           DISP=OLD
//SYSPUNCH DD DSN=DSN!!0.SDSNAMP(DSN8BCPG),
//           DISP=SHR

//STEP  2: CREATE CICS BMS LOGICAL MAPS
//PH05PS02 EXEC DFHASMVS,PARM='DECK,NOOBJECT,SYSPARM(DSECT)',
//           COND=(4,LT),OUTC='*'
//SYSPUNCH DD DSN=DSN!!0.SRCLIB.DATA(DSN8MPMD),
//           DISP=OLD
//SYSPUNCH DD DSN=DSN!!0.SDSNAMP(DSN8BCPD),
//           DISP=SHR

//STEP  3: CREATE CICS BMS LOGICAL MAPS
//PH05PS03 EXEC DFHASMVS,PARM='DECK,NOOBJECT,SYSPARM(DSECT)',
//           COND=(4,LT),OUTC='*'
//SYSPUNCH DD DSN=DSN!!0.SRCLIB.DATA(DSN8MPMN),
//           DISP=OLD
//SYSPUNCH DD DSN=DSN!!0.SDSNAMP(DSN8BCPN),
//           DISP=SHR

//STEP  4: CREATE CICS BMS LOGICAL MAPS
//PH05PS04 EXEC DFHASMVS,PARM='DECK,NOOBJECT,SYSPARM(DSECT)',
//           COND=(4,LT),OUTC='*'
//SYSPUNCH DD DSN=DSN!!0.SRCLIB.DATA(DSN8MPML),
//           DISP=OLD
//SYSPUNCH DD DSN=DSN!!0.SDSNAMP(DSN8BCPL),
//           DISP=SHR

//STEP  5: CREATE CICS BMS LOGICAL MAPS
//PH05PS05 EXEC DFHASMVS,PARM='DECK,NOOBJECT,SYSPARM(DSECT)',
//           COND=(4,LT),OUTC='*'
//SYSPUNCH DD DSN=DSN!!0.SRCLIB.DATA(DSN8MPMU),
//           DISP=OLD
//SYSPUNCH DD DSN=DSN!!0.SDSNAMP(DSN8BCPU),
//           DISP=SHR

//STEP  6: CICS TRANSLATE FOR SQL 0 PART
//PH05PS06 EXEC PGM=DFHHEPP1$,COND=(4,LT)
//SYSPRINT DD SYSOUT=*
//SYSPRINT DD SYSDUMP DD SYSOUT=*
//SYSPUNCH DD DSN=DSN!!0.SRCLIB.DATA,DISP=(NEW,PASS),
//           UNIT=SYSDA,SPACE=(400,(100,100)),
//           DCB=BLKSIZE=400
//SYSPUNCH DD DSN=DSN!!0.SDSNAMP(DSN8BCP0),
//           DISP=SHR

//STEP  7: PREPARE SQL 0 PART
//PH05PS07 EXEC DSNHPLI,MEM=DSN8BCP0,
//           COND=(4,LT),
//           PARM.PPLI='MACRO, NOSYNTAX, MDECK, NOINSOURCE, NOSOURCE',
//           PARM.PC='HOST(PLI), CCSID(37), NOGRAPHIC, STDSQL(NO)',
//           PARM.PLI=('NOPT,SOURCE,OBJECT,MARGINS(2,72,0)',
//             'LIMITS(EXTNAME(7)),OPTIONS', 'SYSTEM(CICS)'),
//           PARM.LKED='NCAL'
//PPLI.SYSPIN DD DSN=DSN!!0.SRCLIB.OUT0,DISP=(OLD,DELETE)
//PPLI.SYSLIB DD DSN=CICSTS.SDFHPLI,
//             DISP=SHR
//PC.DBRMLIB DD DSN=DSN!!0.DBRMLIB.DATA(DSN8BCP0),
//             DISP=SHR
//PC.SYSCIN DD DSN=DSN!!0.DSRMLIB.DATA,
Chapter 12. Sample data and applications supplied with Db2 for z/OS 1445
/*        STEP 13: PREPARE TELEPHONE APPLICATION
PH05PS13 EXEC DSNHPLI, MEM=DSN8CP3,
    COND=(4,LT),
    PARM.PPLI='MACRO,NOSYNTAX,MDECK,NOINSOURCE,NOSOURCE',
    PARM.PC='HOST(PLI),CCSID(37),NOGRAPHIC,STDSQL(NO)',
    PARM.PL='('NOPT,SOURCE,OBJECT,MARGINS(2,72,0)',
      'LIMITS(EXTNAME(7)),OPTIONS','SYSTEM(CICS)'),
    PARM.LKED='NCAL'
PPLI.SYSPRINT DD DSN=&&CICSOUT3,DISP=(OLD,DELETE)
PPLI.SYSLIB DD DSN=CICSTS.SDFHPL1,
    DISP=SHR
/PC.DBRMLIB DD DSN=DSN!!0.DBRMLIB.DATA(DSN8CP3),
    DISP=SHR
/PC.SYSCIN DD DSN=&&DSNHOUT3
/PC.SYSLIB DD DSN=DSN!!0.SRCLIB.DATA,
    DISP=SHR
    DD DSN=DSN!!0.SDSNSAMP,
    DISP=SHR
/PLI.SYSLIB DD DSN=DSN!!0.SRCLIB.DATA,
    DISP=SHR
/PLI.SYSIN DD DSN=&&DSNHOUT6
/LKED.SYSLMOD DD DSN=DSN!!0.RUNLIB.LOAD(DSN8CP3),
    DISP=SHR
/LKED.SYSLMOD DD DUMMY
*/
/*        STEP 14: CREATE CICS BMS LOGICAL MAPS
PH05PS14 EXEC DFHASMVS,PARM='DECK,NOOBJECT,SYSPARM(DSECT)',
    COND=(4,LT),OUTC='*'
/SYSPUNCH DD DSN=DSN!!0.SRCLIB.DATA(DSN8MPMF),
    DISP=OLD
/SYSSIN DD DSN=DSN!!0.SDSNSAMP(DSN8CPF),
    DISP=SHR
/*        STEP 15: CREATE CICS BMS LOGICAL MAPS
PH05PS15 EXEC DFHASMVS,PARM='DECK,NOOBJECT,SYSPARM(DSECT)',
    COND=(4,LT),OUTC='*'
/SYSPUNCH DD DSN=DSN!!0.SRCLIB.DATA(DSN8MPME),
    DISP=OLD
/SYSSIN DD DSN=DSN!!0.SDSNSAMP(DSN8CPE),
    DISP=SHR
/*        STEP 16: CICS TRANSLATE FOR SQL 0 PART
PH05PS16 EXEC PGM=DFHEPP1$,COND=(4,LT)
/SYSPRINT DD SYSOUT=*
/SYSUDUMP DD SYSOUT=* 
/SYSPUNCH DD DSN=&&CICSOUT6,
    DISP=(NEW,PASS),
    UNIT=SYSDA,SPACE=(400,(100,100)),
    DCB=BLKSIZE=400
/SYSSIN DD DSN=DSN!!0.SDSNSAMP(DSN8CP6),
    DISP=SHR
/*        STEP 17: PREPARE SQL 0 PART
PH05PS17 EXEC DSNHPLI, MEM=DSN8CP6,
    COND=(4,LT),
    PARM.PPLI='MACRO,NOSYNTAX,MDECK,NOINSOURCE,NOSOURCE',
    PARM.PC='HOST(PLI),CCSID(37),NOGRAPHIC,STDSQL(NO)',
    PARM.PL='('NOPT,SOURCE,OBJECT,MARGINS(2,72,0)',
      'LIMITS(EXTNAME(7)),OPTIONS','SYSTEM(CICS)'),
    PARM.LKED='NCAL'
/PPLI.SYSPRINT DD DSN=&&CICSOUT6,DISP=(OLD,DELETE)
PPLI.SYSLIB DD DSN=CICSTS.SDFHPL1,
    DISP=SHR
/PC.DBRMLIB DD DSN=DSN!!0.DBRMLIB.DATA(DSN8CP6),
    DISP=SHR
/PC.SYSCIN DD DSN=&&DSNHOUT6
/PC.SYSLIB DD DSN=DSN!!0.SRCLIB.DATA,
    DISP=SHR
    DD DSN=DSN!!0.SDSNSAMP,
    DISP=SHR
/PLI.SYSLIB DD DSN=&&DSNHOUT6
/LKED.SYSLMOD DD DSN=DSN!!0.RUNLIB.LOAD(DSN8CP6),
    DISP=SHR
/LKED.SYSLMOD DD DUMMY
*/
/*        STEP 18: CICS TRANSLATE FOR SQL 1 PART
PH05PS18 EXEC PGM=DFHEPP1$,COND=(4,LT)
/SYSPRINT DD SYSOUT=*
/SYSUDUMP DD SYSOUT=*

1446   Db2 12 for z/OS: Application Programming and SQL Guide
Chapter 12. Sample data and applications supplied with Db2 for z/OS 1447
INCLUDE SYSLIB(DFHELII)
INCLUDE SYSLIB(DSNCLLI)
REPLACE PLISTART
CHANGE PLIMAIN(CEEMAIN)
INCLUDE SYSLMOD(DSNBCP0)
INCLUDE SYSLMOD(DSN8MPG)
ORDER CEESTART
ENTRY CEESTART
NAME DSNBCP0(R)
INCLUDE SYSLIB(CEESTART)
INCLUDE SYSLIB(CEESG010)
INCLUDE SYSLIB(DFHELII)
INCLUDE SYSLIB(DSNCLLI)
REPLACE PLISTART
CHANGE PLIMAIN(CEEMAIN)
INCLUDE SYSLMOD(DSN8CP1)
INCLUDE SYSLMOD(DSN8MPG)
ORDER CEESTART
ENTRY CEESTART
NAME DSNBCP1(R)
INCLUDE SYSLIB(CEESTART)
INCLUDE SYSLIB(CEESG010)
INCLUDE SYSLIB(DFHELII)
INCLUDE SYSLIB(DSNCLLI)
REPLACE PLISTART
CHANGE PLIMAIN(CEEMAIN)
INCLUDE SYSLMOD(DSNBCP2)
INCLUDE SYSLMOD(DSN8MPG)
ORDER CEESTART
ENTRY CEESTART
NAME DSNBCP2(R)
INCLUDE SYSLIB(CEESTART)
INCLUDE SYSLIB(CEESG010)
INCLUDE SYSLIB(DFHELII)
INCLUDE SYSLIB(DSNCLLI)
REPLACE PLISTART
CHANGE PLIMAIN(CEEMAIN)
INCLUDE SYSLMOD(DSNBCP3)
INCLUDE SYSLMOD(DSN8MPG)
ORDER CEESTART
ENTRY CEESTART
NAME DSNBCP3(R)
INCLUDE SYSLIB(CEESTART)
INCLUDE SYSLIB(CEESG010)
INCLUDE SYSLIB(DFHELII)
INCLUDE SYSLIB(DSNCLLI)
REPLACE PLISTART
CHANGE PLIMAIN(CEEMAIN)
INCLUDE SYSLMOD(DSNBCP6)
INCLUDE SYSLMOD(DSN8MPG)
ORDER CEESTART
ENTRY CEESTART
NAME DSNBCP6(R)
INCLUDE SYSLIB(CEESTART)
INCLUDE SYSLIB(CEESG010)
INCLUDE SYSLIB(DFHELII)
INCLUDE SYSLIB(DSNCLLI)
REPLACE PLISTART
CHANGE PLIMAIN(CEEMAIN)
INCLUDE SYSLMOD(DSNBCP7)
INCLUDE SYSLMOD(DSN8MPG)
ORDER CEESTART
ENTRY CEESTART
NAME DSNBCP7(R)
INCLUDE SYSLIB(CEESTART)
INCLUDE SYSLIB(CEESG010)
INCLUDE SYSLIB(DFHELII)
INCLUDE SYSLIB(DSNCLLI)
REPLACE PLISTART
CHANGE PLIMAIN(CEEMAIN)
INCLUDE SYSLMOD(DSNBCP8)
INCLUDE SYSLMOD(DSN8MPG)
ORDER CEESTART
ENTRY CEESTART
NAME DSNBCP8(R)

//**
//**       STEP 23: BIND PROGRAMS
//**PH05PS23 EXEC PGM=IKJEFT01,DYNAMNBR=20,COND=(4,LT)
//DBRMLIB DD DISP=SHR,DSN=DSN!10.DBRMLIB.DATA
//SYSUDUMP DD SYSOUT=* 
//SYSTSPRT DD SYSOUT=* 
//SYSPRINT DD SYSOUT=* 

1448 Db2 12 for z/OS: Application Programming and SQL Guide
SET CURRENT SQLID = 'SYSADM';
GRANT BIND, EXECUTE ON PLAN DSN8CP0, DSN8CQ0, DSN8CH0
TO PUBLIC;

DSN SYSTEM(DSN)
BIND PACKAGE(DSN8CP!) MEMBER(DSN8CP0) APPLCOMPAT(V11R1) +
   ACT(REP) ISO(CS) CURRENTDATA(YES) ENCODING(EBCDIC)
BIND PACKAGE(DSN8CP!) MEMBER(DSN8CP1) APPLCOMPAT(V11R1) +
   ACT(REP) ISO(CS) CURRENTDATA(YES) ENCODING(EBCDIC)
BIND PACKAGE(DSN8CP!) MEMBER(DSN8CP2) APPLCOMPAT(V11R1) +
   ACT(REP) ISO(CS) CURRENTDATA(YES) ENCODING(EBCDIC)
BIND PACKAGE(DSN8CP!) MEMBER(DSN8CP3) APPLCOMPAT(V11R1) +
   ACT(REP) ISO(CS) CURRENTDATA(YES) ENCODING(EBCDIC)
BIND PACKAGE(DSN8CP!) MEMBER(DSN8CP6) APPLCOMPAT(V11R1) +
   ACT(REP) ISO(CS) CURRENTDATA(YES) ENCODING(EBCDIC)
BIND PACKAGE(DSN8CP!) MEMBER(DSN8CP7) APPLCOMPAT(V11R1) +
   ACT(REP) ISO(CS) CURRENTDATA(YES) ENCODING(EBCDIC)
BIND PACKAGE(DSN8CP!) MEMBER(DSN8CP8) APPLCOMPAT(V11R1) +
   ACT(REP) ISO(CS) CURRENTDATA(YES) ENCODING(EBCDIC)
BIND PLAN(DSN8CP0) +
   PKLIST(DSN8CP!.DSN8CP0, +
   DSN8CP!.DSN8CP1, +
   DSN8CP!.DSN8CP2) +
   ACTION(REPLACE) RETAIN +
   ISO(CS) CURRENTDATA(YES) ENCODING(EBCDIC)
BIND PLAN(DSN8CQ0) +
   PKLIST(DSN8CP!.DSN8CP6, +
   DSN8CP!.DSN8CP7, +
   DSN8CP!.DSN8CP8) +
   ACTION(REPLACE) RETAIN +
   ISO(CS) CURRENTDATA(YES) ENCODING(EBCDIC)
BIND PLAN(DSN8CH0) +
   PKLIST(DSN8CP!.DSN8CP3) +
   ACTION(REPLACE) RETAIN +
   ISO(CS) CURRENTDATA(YES) ENCODING(EBCDIC)
RUN PROGRAM(DSNTIAD) PLAN(DSNTIA!!) -
   LIB('DSN!0.RUNLIB.LOAD')
END

** Chapter 12. Sample data and applications supplied with Db2 for z/OS **
//SYSIN DD DSN=DSN8CPN.SDSNSAMP(DSN8CPN),
   DISP=SHR
//**
//**  STEP 29: LINKEDIT CICS BMS PHYSICAL MAPS
//PH05PS29 EXEC PGM=IEWL,PARM='LIST,LET,XREF',COND=(4,LT)
//SYSUT1 DD UNIT=SYSDA,SPACE=(1024,(100,10))
//SYSLMOD DD DISP=SHR,DNS=DSN8CPN.SDSNSAMP(DSN8CPN)
//SYSPRINT DD SYSPRINT*
//SYSDUMP DD SYSDUMP*
//SYSLIN DD DNS=&&TEMP,DISP=(OLD,DELETE)
// DD *
// NAME DSN8CPN(R)
//**
//**  STEP 30: CREATE CICS BMS PHYSICAL MAPS
//PH05PS30 EXEC DPHASMSV,COND=(4,LT),OUTC='*
//SYSPUNCH DD DNS=&&TEMP,
   DISP=(NEW,PASS),
   UNIT=SYSDA,SPACE=(1024,(100,10)),
   DCB=(RECFM=F,BLKSIZE=80)
//SYSIN DD DNS=DSN8CPN.SDSNSAMP(DSN8CPN)
   DISP=SHR
//**
//**  STEP 31: LINKEDIT CICS BMS PHYSICAL MAPS
//PH05PS31 EXEC PGM=IEWL,PARM='LIST,LET,XREF',COND=(4,LT)
//SYSUT1 DD UNIT=SYSDA,SPACE=(1024,(100,10))
//SYSLMOD DD DNS=DSN8CPN.SDSNSAMP(DSN8CPN)
//SYSPRINT DD SYSPRINT*
//SYSDUMP DD SYSDUMP*
//SYSLIN DD DNS=&&TEMP,DISP=(OLD,DELETE)
// DD *
// NAME DSN8CPN(R)
//**
//**  STEP 32: CREATE CICS BMS PHYSICAL MAPS
//PH05PS32 EXEC DPHASMSV,COND=(4,LT),OUTC='*
//SYSPUNCH DD DNS=&&TEMP,
   DISP=(NEW,PASS),
   UNIT=SYSDA,SPACE=(1024,(100,10)),
   DCB=(RECFM=F,BLKSIZE=80)
//SYSIN DD DNS=DSN8CPN.SDSNSAMP(DSN8CPN)
   DISP=SHR
//**
//**  STEP 33: LINKEDIT CICS BMS PHYSICAL MAPS
//PH05PS33 EXEC PGM=IEWL,PARM='LIST,LET,XREF',COND=(4,LT)
//SYSUT1 DD UNIT=SYSDA,SPACE=(1024,(100,10))
//SYSLMOD DD DNS=DSN8CPN.SDSNSAMP(DSN8CPN)
//SYSPRINT DD SYSPRINT*
//SYSDUMP DD SYSDUMP*
//SYSLIN DD DNS=&&TEMP,DISP=(OLD,DELETE)
// DD *
// NAME DSN8CPN(R)
//**
//**  STEP 34: CREATE CICS BMS PHYSICAL MAPS
//PH05PS34 EXEC DPHASMSV,COND=(4,LT),OUTC='*
//SYSPUNCH DD DNS=&&TEMP,
   DISP=(NEW,PASS),
   UNIT=SYSDA,SPACE=(1024,(100,10)),
   DCB=(RECFM=F,BLKSIZE=80)
//SYSIN DD DNS=DSN8CPN.SDSNSAMP(DSN8CPN)
   DISP=SHR
//**
//**  STEP 35: LINKEDIT CICS BMS PHYSICAL MAPS
//PH05PS35 EXEC PGM=IEWL,PARM='LIST,LET,XREF',COND=(4,LT)
//SYSUT1 DD UNIT=SYSDA,SPACE=(1024,(100,10))
//SYSLMOD DD DNS=DSN8CPN.SDSNSAMP(DSN8CPN)
//SYSPRINT DD SYSPRINT*
//SYSDUMP DD SYSDUMP*
//SYSLIN DD DNS=&&TEMP,DISP=(OLD,DELETE)
// DD *
// NAME DSN8CPN(R)
//**
//**  STEP 36: CREATE CICS BMS PHYSICAL MAPS
//PH05PS36 EXEC DPHASMSV,COND=(4,LT),OUTC='*
//SYSPUNCH DD DNS=&&TEMP,
   DISP=(NEW,PASS),
   UNIT=SYSDA,SPACE=(1024,(100,10)),
   DCB=(RECFM=F,BLKSIZE=80)
//SYSIN DD DNS=DSN8CPN.SDSNSAMP(DSN8CPN)
   DISP=SHR
//**
//**  STEP 37: LINKEDIT CICS BMS PHYSICAL MAPS
//PH05PS37 EXEC PGM=IEWL,PARM='LIST,LET,XREF',COND=(4,LT)
Related reference

“Sample applications in CICS” on page 1392
A set of Db2 sample applications run in the CICS environment.
Information resources for Db2 for z/OS and related products

You can find the online product documentation for Db2 for z/OS in IBM Documentation. IBM Documentation is the home of all online product documentation for Db2 for z/OS and related products, including PDF format manuals.
This information was developed for products and services offered in the US. This material might be available from IBM in other languages. However, you may be required to own a copy of the product or product version in that language in order to access it.

IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing IBM Corporation
North Castle Drive, MD-NC119
Armonk, NY 10504-1785 US

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

19-21, Nihonbashi-Hakozakicho, Chuo-ku
Tokyo 103-8510, Japan

INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some jurisdictions do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM websites are provided for convenience only and do not in any manner serve as an endorsement of those websites. The materials at those websites are not part of the materials for this IBM product and use of those websites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

IBM Director of Licensing IBM Corporation
North Castle Drive, MD-NC119
Armonk, NY 10504-1785 US

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this document and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement or any equivalent agreement between us.
This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to actual people or business enterprises is entirely coincidental.

COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrate programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs. The sample programs are provided "AS IS", without warranty of any kind. IBM shall not be liable for any damages arising out of your use of the sample programs.

Each copy or any portion of these sample programs or any derivative work must include a copyright notice as shown below:

© (your company name) (year).

 Portions of this code are derived from IBM Corp. Sample Programs.
 © Copyright IBM Corp. (enter the year or years).

If you are viewing this information softcopy, the photographs and color illustrations may not appear.

Programming interface information

This information is intended to help you to write programs that contain SQL statements. This information primarily documents General-use Programming Interface and Associated Guidance Information provided by Db2 12 for z/OS. However, this information also documents Product-sensitive Programming Interface and Associated Guidance Information provided by Db2 12 for z/OS.

General-use Programming Interface and Associated Guidance Information

General-use Programming Interfaces allow the customer to write programs that obtain the services of Db2 12 for z/OS.

Product-sensitive Programming Interface and Associated Guidance Information

Product-sensitive Programming Interfaces allow the customer installation to perform tasks such as diagnosing, modifying, monitoring, repairing, tailoring, or tuning of this IBM software product. Use of such interfaces creates dependencies on the detailed design or implementation of the IBM software product. Product-sensitive Programming Interfaces should be used only for these specialized purposes. Because of their dependencies on detailed design and implementation, it is to be expected that programs written to such interfaces may need to be changed in order to run with new product releases or versions, or as a result of service.

Product-sensitive Programming Interface and Associated Guidance Information is identified where it occurs by the following markings:

Product-sensitive Programming Interface and Associated Guidance Information...

Trademarks

IBM, the IBM logo, and ibm.com® are trademarks or registered marks of International Business Machines Corp., registered in many jurisdictions worldwide. Other product and service names might be trademarks of IBM or other companies. A current list of IBM trademarks is available on the web at "Copyright and trademark information" at: http://www.ibm.com/legal/copytrade.shtml.

Linux is a registered trademark of Linus Torvalds in the United States, other countries, or both.
Terms and conditions for product documentation

Permissions for the use of these publications are granted subject to the following terms and conditions:

Applicability: These terms and conditions are in addition to any terms of use for the IBM website.

Personal use: You may reproduce these publications for your personal, noncommercial use provided that all proprietary notices are preserved. You may not distribute, display or make derivative work of these publications, or any portion thereof, without the express consent of IBM.

Commercial use: You may reproduce, distribute and display these publications solely within your enterprise provided that all proprietary notices are preserved. You may not make derivative works of these publications, or reproduce, distribute or display these publications or any portion thereof outside your enterprise, without the express consent of IBM.

Rights: Except as expressly granted in this permission, no other permissions, licenses or rights are granted, either express or implied, to the publications or any information, data, software or other intellectual property contained therein.

IBM reserves the right to withdraw the permissions granted herein whenever, in its discretion, the use of the publications is detrimental to its interest or, as determined by IBM, the above instructions are not being properly followed.

You may not download, export or re-export this information except in full compliance with all applicable laws and regulations, including all United States export laws and regulations.

IBM MAKES NO GUARANTEE ABOUT THE CONTENT OF THESE PUBLICATIONS. THE PUBLICATIONS ARE PROVIDED "AS-IS" AND WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING BUT NOT LIMITED TO IMPLIED WARRANTIES OF MERCHANTABILITY, NON-INFRINGEMENT, AND FITNESS FOR A PARTICULAR PURPOSE.

Privacy policy considerations

IBM Software products, including software as a service solutions, ("Software Offerings") may use cookies or other technologies to collect product usage information, to help improve the end user experience, to tailor interactions with the end user, or for other purposes. In many cases no personally identifiable information is collected by the Software Offerings. Some of our Software Offerings can help enable you to collect personally identifiable information. If this Software Offering uses cookies to collect personally identifiable information, specific information about this offering's use of cookies is set forth below.

This Software Offering does not use cookies or other technologies to collect personally identifiable information.

If the configurations deployed for this Software Offering provide you as customer the ability to collect personally identifiable information from end users via cookies and other technologies, you should seek your own legal advice about any laws applicable to such data collection, including any requirements for notice and consent.

Glossary

The glossary is available in IBM Documentation
For definitions of Db2 for z/OS terms, see Db2 glossary (Db2 Glossary).
Index

Special Characters

_(underscore)_

- assembler host variable 552

' (apostrophe)

- string delimiter precompiler option 852

Numerics

- 31-bit addressing 922

A

- abend
  - effect on cursor position 405
  - for synchronization calls 450
  - IMS U0102 941
  - system X"04E" 450
- abend recovery routine
  - in CAF 39
- access path
  - direct row access 432
- accessibility
  - keyboard xiii
  - shortcut keys xiii
- accessing data
  - from an application program 355
  - activity sample table 985
- adding
  - data 335
  - ALL quantified predicate 393
- ALTER PROCEDURE statement
  - external stored procedure 290
- AMODE link-edit option 863, 922
- ANY quantified predicate 393
- APOST precompiler option 852
- APPLCOMPAT 828
- APPLCOMPAT subsystem parameter
  - changing to new release 832
- application
  - rebinding 883
- application compatibility
  - subsystem parameter 832
  - verifying 831
- application plan
  - binding 869
  - creating 864
  - dynamic plan selection for CICS applications 881
  - listing packages 869
  - rebinding 885
- application program
  - bill of materials 145
  - checking success of SQL statements 475
  - coding SQL statements (continued)
  - coding SQL statements (continued)
  - data entry 335
  - dynamic SQL 496, 500
  - selecting rows using a cursor 401
- design considerations
  - checkpoint 450
  - IMS calls 450
  - programming for DL/I batch 450
  - SQL statements 450
  - structure 936
  - synchronization call abends 450
  - using ISPF (interactive system productivity facility) 835
  - XRST call 450
- duplicate CALL statements 764
- external stored procedures 228
- object extensions 168
- preparation
  - assembling 863
  - binding 864
  - compiling 863
  - DB2 precompiler option defaults 861
  - defining to CICS 863
  - DRDA access 874
  - example 901
  - link-editing 863
  - precompiler option defaults 839
  - preparing for running 835
  - program preparation panel 835
  - using DB2I (DB2 Interactive) 835
- preparation overview 893
- running
  - CICS 945
  - IMS 945
  - program synchronization in DL/I batch 450
  - TSO 933
  - TSO CLIST 945
- table and view declarations 464
- test environment 933
- testing 933
- application program design
  - planning for changes 14
- application programming
  - DCLGEN example 472
  - DCLGEN variable declarations 469
- application programs
  - compatibility 817, 828
  - compatible data types 484
  - host structures 479
  - host variables 477
  - host-variable arrays 478
  - performance 449
- application release incompatibilities 1
  - applications
    - designing 1
    - identifying incompatibilities 831
applications (continued)
migrating 1
planning 1
samples supplied with Db2 985
verifying changes for incompatibilities 831
writing for Db2 461
arithmetic expressions in UPDATE statement 350
array pointer host variable
declaring 609
referencing in SQL statements 608
arrays
example
using arrays in a native SQL procedure 174
native SQL procedure example 174
AS clause
naming columns for view 365
naming columns in union 365
naming derived columns 365
naming result columns 365
ORDER BY name 363
ASCII data, retrieving 504
assembler application program
assembling 863
data type compatibility 564
declaring tables 552
declaring views 552
defining the SQLDA 476, 556
host variable
naming convention 552
host variable, declaring 557
INCLUDE statement 552
including SQLCA 555
indicator variable declaration 563
reentrant 552
SQLCODE host variable 555
SQLSTATE host variable 555
variable declaration 558
assignment, compatibility rules 120
ATTACH precompiler option 852
attachment facility
options in z/OS environment 33
AUTH SIGNON (connection function of RRSAF)
language examples 87
syntax 87
authority
authorization ID 944
creating test tables 949
SYSIBM.SYSTABAUTH table 355
autobind 891
AUTOCOMMIT field of SPUFI panel 955
automatic query rewrite 130
automatic rebinding
conditions for 891
invalid package 891
SQLCA not available 891
automatic rebinds
old plans and packages 4
autonomous
native SQL procedures
autonomous 228
procedures
autonomous 228
batch processing
access to Db2 and DL/I together
binding a plan 877
checkpoint calls 450
commits 450
precompiling 840
batch Db2 application
running 944
starting with a CLIST 945
bill of materials applications 145
binary host variable
assembler 558
C/C++ 584
COBOL 650
PL/I 706
binary host-variable array
C/C++ 595
PL/I 712
BIND
command line processor command 867
BIND COPY
for native SQL procedures 249
BIND COPY REPLACE
for native SQL procedures 250
bind options
planning for 17
BIND PACKAGE subcommand of DSN
options
CURRENTDATA 875
ENCODING 875
location-name 875
OPTIONS 875
SQLERROR 875
options associated with DRDA access 874, 876
remote 876
BIND PLAN subcommand of DSN
options
CURRENTDATA 874
DISCONNECT 874
ENCODING 874
SQLRULES 874, 892
options associated with DRDA access 874
bind process
distributed data 873
binding
application plans 864
changes that require 14
checking BIND PACKAGE options 876
DBRMs precompiled elsewhere 840
options associated with DRDA access 874
packages
remote 876
plans 869
remote package requirements 876
specify SQL rules 892
block fetch
preventing 401
with cursor stability 401
BMP (batch message processing) program
checkpoints 25
bounded character pointer host variable
declaring 609
bounded character pointer host variable (continued)
  description 609
  referencing in SQL statements 608
BTS (batch terminal simulator) 979

C

C application program
  declaring tables 571
  sample application 1024
C/C++
  creating stored procedure 257
C/C++ application program
  data type compatibility 611
  DCLGEN support 469
  declaring views 571
  defining the SQLDA 476, 583
  host structure 603
  INCLUDE statement 571
  including SQLCA 582
  indicator variable array declaration 606
  indicator variable declaration 606
  naming convention 571
  precompiler option defaults 861
  SQLCODE host variable 582
  SQLSTATE host variable 582
  variable array declaration 595
  variable declaration 584
  with classes, preparing 840
C/C++ application programs
  pointer host variables 609
CAF (call attachment facility)
  description 37
CAF functions
  summary of behavior 45
calculated values
  groups with conditions 376
  summarizing group values 375
call attachment facility (CAF)
  application program
    examples 59
    preparation 41
  attention exit routines 39
  authorization IDs 37
  behavior summary 45
  connection functions 46
  connection name 37
  connection properties 37
  connection type 37
  Db2 abends 37
  description 37
  error messages 57
  implicit connections to 42
  invoking 34
  parameters for CALL DSNALI 42
  program requirements 41
  recovery routines 39
  register changes 41
  return codes
    example of checking 59
  return codes and reason codes 57
  sample scenarios 58
  scope 37
  terminated task 37
call attachment facility (CAF) (continued)
  trace 57
call attachment language interface
  loading 39
  making available 39
CALL DSNALI
  parameter list 42
  required parameters 42
CALL DSNRLI
  parameter list 74
  required parameters 74
CALL statement
  command line processor 943
  examples 753
  multiple 764
  syntax for invoking DSNTPSMP 298
catalog table
  SYSIBM.LOCATIONS 776
  SYSIBM.SYSCOLUMNS 356
  SYSIBM.SYSTABAUTH 355
CCSID (coded character set identifier)
  controlling in COBOL programs 675
  precompiler option 852
  setting for host variables 481
SQLDA 504
CEEDUMP
  using to debug stored procedures 967
character host variable
  assembler 558
  C/C++ 584
  COBOL 650
  Fortran 688
  PL/I 706
character host-variable array
  C/C++ 595
  COBOL 660
  PL/I 712
character input data
  REXX program 747
character string
  mixed data 120
  width of column in results 958, 963
check constraint
  considerations 127
  CURRENT RULES special register effect 128
  defining 127
  description 127
  determining violations 983
  enforcement 127
  programming considerations 983
checkpoint
  calls 24, 25
  specifying frequency 25
CHKP call, IMS 24
CICS
  DSNTIAC subroutine
    assembler 552
    C 571
    COBOL 619
    PL/I 695
  environment planning 945
  facilities
    command language translator 849
CICS (continued)
facilities (continued)
control areas 933
EDF (execution diagnostic facility) 979
language interface module (DSNCLI)
use in link-editing an application 863
programming
running a program 933
preparing with JCL procedures 898
storage handling
assembler 552
C 571
COBOL 619
PL/I 695
sync point 20
unit of work 20
CICS applications
thread reuse 117
CICS attachment facility
controlling from applications 116
detecting whether it is operational 117
starting 116
stopping 116
client 30
client program
preparing for calling a remote stored procedure 762
CLOSE statement
description 410
recommendation 415
WHENEVER NOT FOUND clause 502, 504
CLOSE (connection function of CAF)
description 46
language examples 52
program example 59
syntax 52
COALESCE function 384
COBOL
creating stored procedure 257
COBOL application program
compiling 863
controlling CCSID 675
data type compatibility 677
Db2 precompiler option defaults 861
DCLGEN support 469
declaring tables 619
declaring views 619
defining the SQLDA 476, 649
dynamic SQL 500
host structure 668
host variable
use of hyphens 619
host variable, declaring 650
host-variable array, declaring 650
INCLUDE statement 619
including SQLCA 648
indicator variable array declaration 674
indicator variable declaration 674
naming convention 619
object-oriented extensions 682
COBOL application program (continued)
options 619
preparation 863
resetting SQL-INIT-FLAG 619
sample program 623
SQLCODE host variable 648
SQLSTATE host variable 648
variable array declaration 660
variable declaration 650
WHENEVER statement 619
with classes, preparing 840
coding SQL statements
dynamic 496
collection, package
identifying 871
SET CURRENT PACKAGESET statement 871
colon
preceding a host variable 487
preceding a host-variable array 492
column
data types 120
default value
system-defined 119
user-defined 119
displaying, list of 356
heading created by SPUFI 964
labels, usage 504
name, with UPDATE statement 350
retrieving, with SELECT 356
specified in CREATE TABLE 119
width of results 958, 963
COMMA precompiler option 852
command line processor
binding 866
CALL statement 943
stored procedures 942
commands
MQListener 804
commit point
description 20
IMS unit of work 24
COMMIT statement
description 955
in a stored procedure 230
when to issue 20
with RRSAF 67
common table expressions
description 144
examples 145
in a CREATE VIEW statement 143
in a SELECT statement 143
in an INSERT statement 143
infinite loops 388
recursion 145
comparison
compatibility rules 120
HAVING clause
subquery 393
operator, subquery 393
WHERE clause
subquery 393
compatibility
data types 120
compatibility (continued)
  rules 120
composite key 132
compound statement
  example
dynamic SQL 227
  nested IF and WHILE statements 226
EXIT handler 237
  labels 224
compound statements
  nested 234
  within the declaration of a condition handler 238
condition handlers
  empty 245
  conditions
  ignoring 245
CONNECT
  statement
  SPUFI 955
CONNECT (connection function of CAF)
  description 46
  language examples 47
  program example 59
  syntax 47
CONNECT LOCATION field of SPUFI panel 955
CONNECT precompiler option 852
CONNECT processing option
  enforcing restricted system rules 31
CONNECT statement, with DRDA access 775
connecting
  Db2 33
connection
  Db2
  connecting from tasks 936
  function of CAF
    CLOSE 52
    CONNECT 47
    DISCONNECT 54
    OPEN 51
    TRANSLATE 56
  function of RRSAF
    AUTH SIGNON 87
    CONTEXT SIGNON 92
    CREATE THREAD 101
    FIND_DB2_SYSTEMS 107
    IDENTIFY 76
    SET_CLIENT_ID 97
    SET_ID 96
    SET_REPLICATION 100
    SIGNON 82
    SWITCH TO 80
    TERMINATE IDENTIFY 104
    TERMINATE THREAD 103
    TRANSLATE 106
connection properties
  call attachment facility (CAF) 37
  Resource Recovery Services attachment facility (RRSAF) 69
connection to Db2
  environment requirements 33
constants, syntax
  C/C++ 584
  Fortran 688
CONTEXT SIGNON (connection function of RRSAF)
  (continued)
  language examples 92
  syntax 92
CONTINUE clause of WHENEVER statement 533
CONTINUE handler (SQL procedure)
  description 237
  example 237
  coordinating updates
  distributed data 31
  correlated reference
    correlation name 396
    SQL rules 378
    usage 378
    using in subquery 396
  correlation name 396
create
  external SQL procedure by using DSNTPSMP 294
  external SQL procedure by using JCL 304
  external stored procedure 257
CREATE GLOBAL TEMPORARY TABLE statement 135
CREATE PROCEDURE statement
  external stored procedure 257
  for external SQL procedures 304
CREATE TABLE statement
  DEFAULT clause 119
  NOT NULL clause 119
  PRIMARY KEY clause 131
  relationship names 133
  UNIQUE clause 119, 131
  usage 119
CREATE THREAD (connection function of RRSAF)
  language examples 101
  program example 111
  syntax 101
CREATE TRIGGER
  activation order 161
  description 149
  example 149
  timestamp 161
  trigger naming 149
CREATE TYPE statement
  example 168
CREATE VIEW statement 141
created temporary table
  instances 135
  working with 137
creating objects
  in an application program 119
creating stored procedures
  external SQL procedures 291
CURRENT PACKAGESET special register
  dynamic plan switching 881
  identify package collection 871
CURRENT RULES special register
  effect on check constraints 128
  usage 892
current server 30
CURRENT SERVER special register
  description 871
  saving value in application program 778
CURRENT SQLID special register
  use in test 947
  value in INSERT statement 119
cursor
cursor (continued)
  attributes
    using GET DIAGNOSTICS 424
    using SQLCA 424
  closing
    CLOSE statement 415
  deleting a current row 412
  description 401
  dynamic scrollable 402
  effect of abend on position 405
  example
    retrieving backward with scrollable cursor 428
    updating specific row with rowset-positioned cursor 430
    updating with non-scrollable cursor 428
    updating with rowset-positioned cursor 429
  insensitive scrollable 402
  maintaining position 405
  non-scrollable 402
  open state 405
  OPEN statement 408
  result table 401
  row-positioned
    declaring 406
    deleting a current row 409
    description 401
    end-of-data condition 408
    retrieving a row of data 409
    steps in using 406
    updating a current row 409
  rowset-positioned
    declaring 411
    description 401
    end-of-data condition 411
    number of rows 412
    number of rows in rowset 415
    opening 411
    retrieving a rowset of data 412
    steps in using 410
    updating a current rowset 412
  scrollable
    description 402
    dynamic 402
    fetch orientation 416
    INSENSITIVE 402
    retrieving rows 416
    SENSITIVE DYNAMIC 402
    SENSITIVE STATIC 402
    sensitivity 402
    static 402
    updatable 402
  static scrollable 402
  types 402
  WITH HOLD
    description 405
  cursors
    declaring in SQL procedures 236

D

data (continued)
  associated with WHERE clause 358
  currency 401
  distributed 30
  modifying 335
  not in a table 449
  retrieval using SELECT * 390
  retrieving a rowset 412
  retrieving a set of rows 409
  retrieving large volumes 447
  scrolling backward through 425
  security and integrity 19
  updating during retrieval 389
  updating previously retrieved data 426
  data encryption 134
  data integrity
    tables 126
  data type
    built-in 120
    comparisons 487
  compatibility
    assembler application program 564
    C application program 611
    COBOL and SQL 677
    Fortran and SQL 693
    PL/I application program 720
    REXX and SQL 743
  data types
    compatibility 484
    used by DCLGEN 469
  DATE precompiler option 852
  datetime data type 120
  Db2
    connection from a program 33
  Db2 abend
    DL/I batch 450
  Db2 coprocessor
    processing SQL statements 839
  Db2 functions
    IBM MQ
      MQREADALL 782
      MQREADALLCLOB 782
      MQREADCLOB 782
      MQRECEIVE 782
      MQRECEIVEALL 782
      MQRECEIVEALLCLOB 782
      MQRECEIVECLOB 782
      MQSEND 782
  Db2 MQ tables
    descriptions 785
  Db2 private protocol access
    coding an application 773
  Db2 sample application
    DSN8BC3 1030
    DSN8BD3 1038
    DSN8BE3 1044
    DSN8BF3 1054
    DSN8BP3 1048
    DSN8CC0 1393
    DSN8CC1 1400
    DSN8CC2 1402
    DSN8CP0 1408
    DSN8CP1 1414
    DSN8CP2 1416
Db2 sample application (continued)

DSN8CP3 1434
DSN8CP6 1421
DSN8CP7 1427
DSN8CP8 1429
DSN8DLPL 1253
DSN8DLPV 1279
DSN8DLRV 1265
DSN8DUAD 1180
DSN8DUAT 1187
DSN8DUCD 1193
DSN8DUCY 1222
DSN8DUT1 1227
DSN8DUWC 1234
DSN8DUWF 1238

DB2 sample application (continued)

DSNTEJ61 1311, 1443
DSNTEJ62 1313
DSNTEJ63 1315
DSNTEJ64 1317
DSNTEJ65 1318
DSNTEJ66 1326
DSNTEJ67 328
DSNTEJ6D 1308
DSNTEJ6F 1305, 1341
DSNTEJ6G 1306
DSNTEJ6T 1309, 1345
DSNTEJ6W 1321
DSNTEJ6Z 1324
DSNTEJ71 1305, 1341
DSNTEJ73 1343
DSNTEJ75 1309, 1345
DSNTIJLC 821
DSNTIJLR 824

DB2_RETURN_STATUS

using to get procedure status 767

DB2I

default panels 836
invoking DCLGEN 465

DB2I (DB2 Interactive)

background processing
run time libraries 905
EDITJCL processing
run time libraries 905
interrupting 951
menu 951

BIND PACKAGE 910
BIND PLAN 912
Compile, Link, and Run 922
Current SPUFI Defaults 956
DB2I Primary Option Menu 951
Defaults for BIND PACKAGE 915
Defaults for BIND PLAN 917
Defaults for REBIND PACKAGE 915
Defaults for REBIND PLAN 917
Precompile 908
Program Preparation 901
System Connection Types 919

preparing programs 835
program preparation example 901

SPUFI 951
SPUFI 951

DB2I defaults
setting 836

DBCS (double-byte character set)
translation in CICS 849

DBINFO

passing to external stored procedure 257
user-defined function 194

DBRM (database request module)
description 845

DBRM in HFS files
binding 866

DCLGEN

COBOL example 472
data types 469
declaring indicator variable arrays 465
DCLGEN (continued)
generating table and view declarations 464
   generating table and view declarations from DB2I 465
   INCLUDE statement 471
   including declarations in a program 471
   invoking 464
   using from DB2I 465
   variable declarations 469
DCLGEN (declarations generator)
description 464
DDITV02 input data set 895
DDOTV02 output data set 895
Debug Tool
   user-defined function 964
debugging
   recording messages for stored procedures 972
   stored procedures 967
debugging application programs 973
DEC15
   precompiler option 852
   rules 390
DEC31
   avoiding overflow 390
   precompiler option 852
   rules 390
decimal
   15 digit precision 390
   31 digit precision 390
   arithmetic 390
DECIMAL data type
   C/C++ 584
declarations generator (DCLGEN)
description 464
DECLARE CURSOR statement
description, row-positioned 406
   description, rowset-positioned 411
   FOR UPDATE clause 406
   multilevel security 406
   prepared statement 502
   scrollable cursor 402
   WITH HOLD clause 405
   WITH RETURN option 279
   WITH ROWSET POSITIONING clause 411
DECLARE GLOBAL TEMPORARY TABLE statement 137
DECLARE TABLE statement
   assembler 552
   C 571
   COBOL 619
   Fortran 684
   in application programs 463
   PL/I 695
declared temporary table
   including column defaults 137
   including identity columns 137
   instances 137
   ON COMMIT clause 139
   qualifier for 137
   remote access using a three-part name 773
   requirements 137
   working with 137
declaring tables and views
   advantages 463
   DELETE statement
   correlated subquery 396
DELETE statement (continued)
description 341, 352
   positioned 412
   restrictions 409
   WHERE CURRENT clause 409, 412
deleting
   current rows 409
   data 352
   every row from a table
   with TRUNCATE 352
   rows from a table 352
   DENSE_RANK specification
   example 368
department sample table
   creating 134
DEPLOY bind option
   for native SQL procedures 253
DESCRIBE INPUT statement 525
DESCRIBE statement
   column labels 504
   INTO clauses 504
designing
   applications 1
   designing applications
   distributed data 29
   DFHEIENT macro 552
   DFS1000 (IMS language interface module) 863
diagnostics area
   RESIGNAL affect on 249
   SIGNAL affect on 249
direct row access 432
disability xiii
DISCONNECT (connection function of CAF)
description 46
   language examples 54
   program example 59
   syntax 54
displaying
   table columns 356
   table privileges 355
DISTINCT
   clause of SELECT statement 364
   unique values 364
distinct type
   assigning values 339
   comparing types 399
   description 169
   example
   argument of user-defined function (UDF) 170
   arguments of infix operator 458
   casting constants 459
   casting function arguments 458
   casting host variables 459
   LOB data type 170
   function arguments 458
   strong typing 169
   UNION with INTERSECT 398
   with EXCEPT 398
   with UNION 398
distinct types
   creating 168
   distributed data
   bind process 873
distributed data (continued)
  coordinating updates 31
  copying a remote table 773
  DBPROTOCOL bind option 773
  designing applications for 29
  encoding scheme of retrieved data 779
  example
    accessing remote temporary table 775
    binding at remote server 873
    connecting to remote server 775
    using alias for multiple sites 776
    using RELEASE statement 777
    using three-part table names 773
  executing long SQL statements 778
  identifying server at run time 778
  maintaining data currency 401
  program preparation 876
  programming
    coding with Db2 private protocol access 773
    coding with DRDA access 773
  retrieving from ASCII or Unicode tables 779
  three-part table names 773
  transmitting mixed data 777
  two-phase commit 31
  using alias for location 776
DL/I batch
  application programming 450
  checkpoint ID 942
  Db2 requirements 450
  DDITV02 input data set 895
  DSNMTV01 module 939
  features 450
  SSM= parameter 939
  submitting an application 939
DRDA access
  accessing remote temporary table 775
  bind options 874
  coding an application 773
  connecting to remote server 775
  precompiler options 862
  preparing programs 874
  programming hints 778
  releasing connections 777
  sample program 633
  SQL limitations at different servers 778
DRDA access with CONNECT statements
  sample program 633
DRDA with three-part names
  sample program 638
DROP TABLE statement 141
DSN applications, running with CAF 34
DSN command of TSO
  return code processing 933
  RUN subcommands 933
DSN_FUNCTION_TABLE table 457
DSN8BC3 1030
DSN8BC3 sample program 619
DSN8BD3 1038
DSN8BD3 sample program 571
DSN8BE3 1044
DSN8BE3 sample program 571
DSN8BF3 1054
DSN8BF3 sample program 684
DSN8BP3 1048
DSN8BP3 sample program 695
DSN8CC0 1393
DSN8CC1 1400
DSN8CC2 1402
DSN8CP0 1408
DSN8CP1 1414
DSN8CP2 1416
DSN8CP3 1434
DSN8CP6 1421
DSN8CP7 1427
DSN8CP8 1429
DSN8DPL PL 1253
DSN8DLPV 1279
DSN8DLRV 1265
DSN8DUAD 1180
DSN8DUAT 1187
DSN8DUCD 1193
DSN8DUCT 1209
DSN8DUCY 1222
DSN8DUWC 1234
DSN8DUWF 1238
DSN8EC1 1139
DSN8EC2 1146
DSN8ED1 1121
DSN8ED2 1132
DSN8ED3 1153
DSN8ED4 308
DSN8ED5 324
DSN8ED6 1159
DSN8ED7 1163
DSN8ED9 1167
DSN8EPU 1118
DSN8ESET 1151
DSN8ES2 1157
DSN8ES3 1173
DSN8EUAD 1246
DSN8EUMN 1250
DSN8HC3 1061
DSN8HEC1 1347
DSN8HEC2 1351
DSN8HEC3 1353
DSN8HEP 1358
DSN8HEP1 1361
DSN8HEP2 1363
DSN8HEP3 1379
DSN8HEP6 1368
DSN8HEP7 1372
DSN8HEP8 1374
DSN8HC3 1088
DSN8HSP3 1094
DSN8HSP4 1082
DSN8HSP5 1083
DSN8HSP6 1088
DSN8HSP7 1094
DSN8HSP8 1094
DSN8HSP9 1094
DSN8HSTP 1095
DSN8HSTU 1095
DSN8HSTW 1095
DSN8HSTX 1095
DSN8HSTY 1095
DSN8HSTZ 1095
DSN8HSTP 1095
DSN8HSTW 1095
DSN8HSTX 1095
DSN8HSTY 1095
DSN8HSTZ 1095
DSN8HSTP 1095
DSN8HSTW 1095
DSN8HSTX 1095
DSN8HSTY 1095
DSN8HSTZ 1095
DSNALI
  loading 39
  making available 39
  DSNCLI (CAF language interface module)
  example of deleting 59
  example of loading 59
  DSNCL1 (CICS language interface module) 863
DSNEBP10 915
DSNEBP11 915
DSNH command of TSO 974

Index 1469
DSNHASM procedure 897
DSNHC procedure 897
DSNHC0B procedure 897
DSNHC0B2 procedure 897
DSNHC0P procedure 897
DSNHC0P2 procedure 897
DSNHFOR procedure 897
DSNHICB2 procedure 897
DSNHIC0B procedure 897
DSNHIC0B2 procedure 897
DSNHCPP procedure 897
DSNHCPP2 procedure 897
DSNHPLI procedure 897
DSNHPI procedure 897
DSNHLI entry point to DSNALI
program example 59
DSNHLI2 entry point to DSNALI
program example 59
DSNMTV01 module 939
DSNRLI loading 71
making available 71
DSNETEDIT CLIST 886
DSNETEJ1L 1303
DSNETEJ1P 1301
DSNETEJ2A 1299
DSNETEJ2C 1285
DSNETEJ2D 1287
DSNETEJ2E 1288
DSNETEJ2F 1293
DSNETEJ2P 1290
DSNETEJ2U 1330
DSNETEJ3C 1295
DSNETEJ3P 1298
DSNETEJ4C 1386
DSNETEJ4P 1388
DSNETEJ5C 1441
DSNETEJ5P 1311, 1443
DSNETEJ6 1297
DSNETEJ61 1311, 1443
DSNETEJ62 1313
DSNETEJ63 1315
DSNETEJ64 1317
DSNETEJ65 1318
DSNETEJ66 1326
DSNETEJ67 328
DSNETEJ6D 1308
DSNETEJ6P 1305, 1341
DSNETEJ6S 1306
DSNETEJ6T 1309, 1345
DSNETEJ6W 1321
DSNETEJ6Z 1324
DSNETEJ71 1305, 1341
DSNETEJ73 1343
DSNETEJ75 1309, 1345
DSNTEP2 and DSNTEP4 sample program
specifying SQL terminator 1009, 1017
DSNTEP2 sample program
how to run 1008
parameters 1008
program preparation 1008
DSNTEP4 sample program
how to run 1008
parameters 1008
program preparation 1008
DSNTIAC subroutine
assembler 552
C 571
DSNTIAC subroutine (continued)
COBOL 619
PL/I 695
DSNTIAD sample program
how to run 1008
parameters 1008
program preparation 1008
specifying SQL terminator 1015
DSNTIAR subroutine
assembler 539
C 571
COBOL 619
description 529
Fortran 684
PL/I 695
return codes 530
using 529
DSNTIAUL sample program
how to run 1008
parameters 1008
program preparation 1008
DSNTIJLC 821
DSNTIJLR 824
DSNTIJSD sample program
using to set up the Unified Debugger 970
DSNTIR subroutine 684
DSNTPSMP
creating external SQL procedures 294
required authorizations 294
syntax for invoking 298
DSNTRACE data set 57
DSNU1 115
DSNXDBRM 845
DSNXNBRM 845
DYNAM option of COBOL 619
dynamic buffer allocation
FETCH WITH CONTINUE 422
dynamic plan selection
restrictions with CURRENT PACKAGESET special register 881
using packages with 881
dynamic SQL
advantages and disadvantages 497
assembler program 504
C program 504
COBOL application program 619
COBOL program 500
description 496
effect of bind option REOPT(ALWAYS) 504
effect of WITH HOLD cursor 520
EXECUTE IMMEDIATE statement 519
fixed-list SELECT statements 502
Fortran program 684
host languages 500
non-SELECT statements 501, 520
PL/I 504
PREPARE and EXECUTE 520
programming 496
requirements 497
restrictions 497
sample C program 574
varying-list SELECT statements 504
DYNAMICRULES bind option 879
ECB (event control block)
CONNECT function of CAF 47
IDENTIFY function of RRSAF 76
SET_REPLICATION function of RRSAF 100
EDIT panel, SPUFI
SQL statements 951
embedded semicolon 1015
embedded SQL applications
host variables, XML data 542
XML data 541
employee photo and resume sample table 992
employee sample table 988
employee-to-project activity sample table 995
ENCRYPT_TDES function 134
end-of-data condition 408, 411
error
arithmetic expression 540
division by zero 540
handling 533
messages generated by precompiler 974
overflow 540
return codes 528
run 973
errors when retrieving data into a host variable
determining cause 482
examples
MQListener 809
EXCEPT
keeping duplicate rows with ALL 371
EXCEPT clause
columns of result table 371
exception condition handling 533
EXECUTE IMMEDIATE statement 519
EXECUTE statement
dynamic execution 520
parameter types 504
USING DESCRIPTOR clause 504
EXISTS predicate, subquery 393
EXIT handler (SQL procedure) 237
exit routine
abend recovery with CAF 39
attention processing with CAF 39
EXPLAIN
automatic rebind 891
external SQL procedure
creating 291
external SQL procedures
creating by using DSNTPSMP 294
creating by using JCL 304
debugging with the Unified Debugger 970
migrating to native SQL procedures 292
external stored procedure
creating 257
modifying the definition 290
package 276
package authorizations 276
plan 276
preparing 257
reentrant 281
running as authorized program 257
FETCH CURRENT CONTINUE 421
FETCH statement
description, multiple rows 412
description, single row 409
fetch orientation 416
host variables 502
multiple-row
assembler 552
description 412
FOR n ROWS clause 415
number of rows in rowset 415
using with descriptor 412
using with host-variable arrays 412
row and rowset positioning 427
scrolling through data 425
USING DESCRIPTOR clause 504
using row-positioned cursor 409
FETCH WITH CONTINUE 421
file reference variable
Db2-generated construct 445
FIND_DB2_SYSTEMS (connection function of RRSAF)
language examples 107
syntax 107
fixed buffer allocation
FETCH WITH CONTINUE 423
FLAG precompiler option 852
FLOAT precompiler option 852
FOLD
value for C and CPP 852
value of precompiler option HOST 852
FOR UPDATE clause 406
FOREIGN KEY clause
description 133
usage 133
format
SELECT statement results 962
SQL in input data set 951
formatting
result tables 364
Fortran application program
@PROCESS statement 684
byte data type 684
constant syntax 688
data type compatibility 693
declaring tables 684
declaring views 684
defining the SQLDA 476, 687
host variable, declaring 688
INCLUDE statement 684
including SQLCA 686
indicator variable declaration 691
naming convention 684
parallel option 684
precompiler option defaults 861
SQLCODE host variable 686
SQLSTATE host variable 686
statement labels 684
variable declaration 688
WHENEVER statement 684
FROM clause
joining tables 378
FROM clause (continued)
SELECT statement 356
FRR (functional recovery routine) in CAF 39
FULL OUTER JOIN clause 384
function resolution 455
functional recovery routine (FRR) in CAF 39

G
GENERAL linkage convention 260, 262
GENERAL WITH NULLS linkage convention 260, 265
general-use programming information, described 1456
generating table and view declarations by using DCLGEN 464
with DCLGEN from DB2I 465
generating XML documents for MQ message queue 785
GET DIAGNOSTICS
using to get procedure status 767
GET DIAGNOSTICS statement
condition items 534
connection items 534
data types for items 534, 536
description 534
multiple-row INSERT 534
RETURN_STATUS item 246
ROW_COUNT item 412
statement items 534
using in handler 245
GO TO clause of WHENEVER statement 533
GRANT statement 949

graphic host variable
assembler 558
C/C++ 584
COBOL 650
PL/I 706
graphic host-variable array
C/C++ 595
COBOL 660
PL/I 712
GRAPHIC precompiler option 852
GROUP BY clause
use with aggregate functions 375

H
handler, using in SQL procedure 237
HAVING clause
selecting groups subject to conditions 376
HOST
FOLD value for C and CPP 852
precompiler option 852
host language
dynamic SQL 500
host language data types
compatibility with SQL data types 484
host structure
C/C++ 603
COBOL 668
description 479
indicator structure 479
PL/I 717
host variable (continued)
retrieving row of data 495
using SELECT INTO 495
host variables
compatible data types 484
XML in assembler 542, 543
XML in C language 543, 544
XML in COBOL 544, 545
XML in embedded SQL applications 542
XML in PL/I 545–547
host-variable array
C/C++ 595
COBOL 650, 660
description 478, 492
indicator variable array 479
inserting multiple rows 493
PL/I 705, 712
retrieving multiple rows 493
IBM Data Studio Developer
creating external SQL procedures 291
creating native SQL procedures 231
IBM MQ
APIs 780
Db2 functions
connecting applications 797
programming considerations 782
retrieving messages 796
sending messages 795
description 780
interaction with Db2 780
message handling 780
Message Queue Interface (MQI) 781
messages 780
scalar functions
MQREADCLOB 782
MQRECEIVE 782
MQRECEIVECLOB 782
IBM MQ (continued)
scalar functions (continued)
MQSEND 782
table functions
MOREADALL scalar function 782
MOREADALLCLOB 782
MORECEIVEALL 782
MORECEIVEALLCLOB 782
IDENTIFY (connection function of RRSAF)
language examples 76
program example 111
syntax 76
identity column
defining 124, 339
IDENTITY_VAL_LOCAL function 124
inserting in table 139
inserting values into 339
trigger 149
using as parent key 124
IFCID 0366 828
IFCID 0376 828
IKJEFT01 terminal monitor program in TSO 944
implicit CAF connection 42
implicit RRSAF connections 73
IMS
checkpoint calls 24
checkpoints 25
CHKP call 24
commit point 24
environment planning 945
language interface module (DFSLI000) 863
link-editing 863
recovery 21
ROLL call 21, 24
SYNC call 24
unit of work 24
IMS programs
recovery 27
IN predicate, subquery 393
incompatibilities of releases
applications and SQL 1
incompatible applications
identifying 831
index
types
foreign key 133
primary 140
unique 140
unique on primary key 132
indicator structure
description 479
indicator variable
description 479
inserting null values 494
indicator variable array
description 479
inserting null values 494
indicator variable arrays
declaring with DCLGEN 465
indicator variable arraysC/C++ syntax 606
indicator variable arraysCOBOL syntax 674
indicator variable arraysPL/I syntax 719
indicator variables
using to pass large output parameters 758
indicator variables
assembler syntax 563
indicator variablesC/C++ syntax 606
indicator variablesCOBOL syntax 674
indicator variablesFortran syntax 691
indicator variablesPL/I syntax 719
infinite loop 388
informational referential constraint
automatic query rewrite 130
description 130
INNER JOIN clause 382
input data set DDITV02 895
input parameters
stored procedures 219
INSERT statement
description 335
multiple rows 337
single row 335
VALUES clause 335
with identity column 339
with ROWID column 338
inserting
values from host-variable arrays 493
inserting data
by using host variables 492
Interactive System Productivity Facility (ISPF) 951
INTERSECT
keeping duplicate rows with ALL 371
INTERSECT clause
columns of result table 371
invalid SQL terminator characters 1015
invalidated packages
identifying 17
invoking
call attachment facility (CAF) 34
Resource Recovery Services attachment facility (RRSAF) 64
invoking stored procedures
syntax for command line processor 943
isolation level
REXX 748
ISPF (Interactive System Productivity Facility)
browse 955, 962
DB2 uses dialog management 951
Program Preparation panel 901
programming 936
scroll command 964
ISPLINK SELECT services 936
J
Java stored procedures
debugging with the Unified Debugger 970
JCL (job control language)
batch backout example 941
DDNAME list format 844
page number format 845
precompilation procedures 897
precompiler option list format 844
preparing a CICS program 898
preparing a object-oriented program 840
starting a TSO batch application 944
JOIN operation
- FULL OUTER JOIN 384
- INNER JOIN 382
- joining a table to itself 382
- joining tables 378
- LEFT OUTER JOIN 385
- more than one join 380
- operand
  - nested table expression 378
  - user-defined table function 378
- RIGHT OUTER JOIN 386
- SQL rules 386

KEEP DYNAMIC bind option
- ROLLBACK 5

KEEP DYNAMIC bind option
- ROLLBACK 5

K

KEEP DYNAMIC bind option
- ROLLBACK 5

K

KEEP DYNAMIC bind option
- ROLLBACK 5

L

label, column 504
language interface modules
- DSNCLI 863
large object (LOB)
- character conversion 442
- declaring host variables
  - for precompiler 435
- declaring LOB file reference variables 435
- declaring LOB locators 435
- defining and moving data into Db2 122
- expression 443
- file reference variable 444
- indicator variable 442
- locator 441
- materialization 440
- sample applications 433
- LEFT OUTER JOIN clause 385
- LEVEL precompiler option 852
- libraries
  - for table declarations and host-variable structures 472
- LINECOUNT precompiler option 852
- link-edit 115
- link-editing
  - AMODE option 922
  - RMODE option 922
- linkage conventions
  - GENERAL 260, 262
  - GENERAL WITH NULLS 260, 265
  - SQL 260, 268
  - stored procedures 260
- links

links (continued)
- non-IBM Web sites 1457
- LOAD z/OS macro used by CAF 41
- LOAD z/OS macro used by RRSAF 72
- LOB column, definition 122
- LOB file reference variable
  - assembler 558
  - C/C++ 584, 595
  - COBOL 650, 660
  - PL/I 706, 712
- LOB host-variable array
  - C/C++ 595
  - COBOL 660
  - PL/I 712
- LOB locator
  - assembler 558
  - C/C++ 584, 595
  - COBOL 660
  - Fortran 688
  - PL/I 706, 712
- LOB values
  - fetching 421
- LOB variable
  - assembler 558
  - C/C++ 584
  - COBOL 660
  - Fortran 688
  - PL/I 706
- location name 30
- lock
  - escalation
    - when retrieving large numbers of rows 447

M

mapping macro
- assembler applications 570
- DSNXDRBM 845
- DSNXGRM 845
MARGINS precompiler option 852
- materialization
- LOBs 440
- merging
- data 341
- message
  - analyzing 974
  - obtaining text
    - assembler 539
    - C 571
    - COBOL 619
    - Fortran 684
    - PL/I 695
  - message data
    - IBM MQ 780
  - message handling
    - IBM MQ 780
  - Message Queue Interface (MQI)
    - Db2 MQ tables 785
    - policies 782
    - services 781
    - WebSphere MQ 781
  - messages
    - IBM MQ 780
migrating
applications 1 777
mixed data
converting 120
transmitting to remote location 777
MLS (multilevel security)
referential constraints 130
triggers 163
modified source statements 845
modify
external stored procedure definition 290
modifying
data 335
MPP program
checkpoints 25
MQ message queue
sending table data 785
shredding XML documents 785
MQ XML composition stored procedures
alternative method 785
MQ XML decomposition stored procedures
alternative method 785
MQListener
commands 804
examples 809
MQREADALL table function 782
MQREADALLCLOB table function 782
MQREADCLOB scalar function 782
MQRECEIVE scalar function 782
MQRECEIVEALL table function 782
MQRECEIVEALLCLOB table function 782
MQRECEIVECLOB scalar function 782
MQSEND scalar function 782
MSGFILe run time option
using to debug stored procedures 972
multilevel security (MLS) check
referential constraints 130
triggers 163
multiple-row FETCH statement
checking DB2_LAST_ROW 536
SQLCODE +100 528
multiple-row INSERT statement
dynamic execution 523
NOT ATOMIC CONTINUE ON SQLEXCEPTION 534
using GET DIAGNOSTICS 534

N
naming convention
assembler 552
C 571
COBOL 619
Fortran 684
PL/I 695
REXX 725
tables you create 134
native SQL procedures
BIND COPY 249
BIND COPY REPLACE 250
creating 231
debugging with the Unified Debugger 970
deploying to another server 253
deploying to production 253
native SQL procedures (continued)
migrating from external SQL procedures 292
packages for 249
replacing packages for 250
nested compound statements
cursor declarations 236
definition 234
for controlling scope of conditions 238
scope of variables 233
statement labels 235
nested table expression
correlated reference 378
correlation name 378
join operation 378
NEWFUN precompiler option 852
NODYNAM option of COBOL 619
NOFOR precompiler option 852
NOGRAPHIC precompiler option 852
non-Db2 resources
accessing from stored procedure 277
nontabular data storage 350
NOOPTIONS precompiler option 852
NOPADNTSTR precompiler option 852
NOSOURCE precompiler option 852
NOT FOUND clause of WHENEVER statement 533
not logged
table spaces
recovering 28
NOXREF precompiler option 852
NUL character in C 571
null
determining value of output host variable 489
NULL
pointer in C 571
null value
column value of UPDATE statement 350
determining column value 361
inserting into columns 494
Null, in REXX 725
numeric
data
width of column in results 963
numeric data
description 120
width of column in results 958
numeric host variable
assembler 558
C/C++ 584
COBOL 650
Fortran 688
PL/I 706
numeric host-variable
array
C/C++ 595
COBOL 660
PL/I 712
NUMTCB parameter 766

O
object-oriented program, preparation 840
objects
creating in a application program 119
ON clause, joining tables 378
ONEPASS precompiler option 852
OPEN
statement
  opening a cursor 408
  opening a rowset cursor 411
  prepared SELECT 502
  USING DESCRIPTOR clause 504
  without parameter markers 504
OPEN (connection function of CAF)
  description 46
  language examples 51
  program example 59
  syntax 51
  syntax usage 51
OPTIONS precompiler option 852
ORDER BY clause
  SELECT statement 366
  with ORDER OF clause 348
ORDER OF clause 348
organization application
examples 1024
outer join
  FULL OUTER JOIN 384
  LEFT OUTER JOIN 385
  RIGHT OUTER JOIN 386
output host variable
  determining if null 489
  determining if truncated 489
output host variable processing
  errors 482
output parameters
  stored procedures 219, 758
P
package
  binding
    DBRM to a package 864
    remote 876
    to plans 869
    identifying at run time 871
    invalid 14, 17
    invalidated 891
    listing 869
    location 871
    rebinding examples 883
    rebinding with pattern-matching characters 883
    selecting 871
    trigger 160
    version, identifying 865
package authorization
  for external stored procedures 276
packages
  automatic rebinds 4
  collection ID for stored procedure packages 280
  for external procedures 276
  for native SQL procedures 249
  for nested routines 280
PADNTSTR precompiler option 852
panel
  Current SPUFI Defaults 956, 960
  DB2I Primary Option Menu 951
  DSNEPRI 951
  DSNESP01 951
panel (continued)
  DSNESP02 956
  DSNESP07 960
  EDIT (for SPUFI input data set) 951
  SPUFI 951
panels
  DB2I (DB2 Interactive) 472
  DB2I DEFAULTS 472
  DCLGEN 472
  DSNEDP01 472
  DSNEP01 472
  DSNEP02 472
  REBIND PACKAGE 925
  REBIND TRIGGER PACKAGE 927
parameter list
  stored procedures 219
parameter marker
  casting in function invocation 459
  dynamic SQL 520
  more than one 520
  values provided by OPEN 502
  with arbitrary statements 504
parameter marker information
  obtaining by using an SQLDA 525
PARAMETER STYLE SQL option
  using to debug stored procedures 967
parent key 132
PARMS option 935
performance
  affected by
    application structure 936
    application programs 449
    programming applications 17
PERIOD precompiler option 852
phone application, description 1024
PL/I
  creating stored procedure 257
PL/I application program
  coding considerations 695
  data type compatibility 720
  DBCS constants 695
  DCLGEN support 469
  declaring tables 695
  declaring views 695
  defining the SQLDA 476, 705
  host structure 717
  host variable, declaring 705
  host-variable array, declaring 705
  INCLUDE statement 695
  including SQLCA 704
  indicator variable array declaration 719
  indicator variable declaration 719
  naming convention 695
  SQLCODE host variable 704
  SQLSTATE host variable 704
  statement labels 695
  variable array declaration 712
  variable declaration 706
  WHENEVER statement 695
planning
  applications 1
  bind options 17
  planning application programs
  SQL processing options 13
Index
referential constraint (continued)
  informational 130
  name 133
  on tables with data encryption 134
  on tables with multilevel security 130
referential integrity
  effect on subqueries 397
  programming considerations 983
register conventions
  RRSAF 73
registers
  changed by CAF (call attachment facility) 41
release incompatibilities
  applications and SQL 1
RELEASE SAVEPOINT statement 27
RELEASE statement, with DRDA access 777
remote stored procedure
  preparing client program 762
REPLACE statement (COBOL) 619
requester 30
resetting control blocks
  CAF 54
RESIGNAL statement
  raising a condition 246
  setting SQLSTATE value 247
Resource Recovery Services attachment facility (RRSAF)
  application program
    preparation 72
    authorization IDs 69
    behavior summary 75
    connection functions 76
    connection name 69
    connection properties 69
    connection type 69
    Db2 abends 69
    description 67
    implicit connections 73
    invoking 64
    loading 71
    making available 71
    parameters for CALL DSNRLI 74
    program examples 111
    program requirements 72
    register conventions 73
    return codes and reason codes 108
    sample JCL 111
    sample scenarios 109
    scope 69
    terminated task 69
restart, DL/I batch programs using JCL 941
restricted system
  definition 31
  forcing rules 31
  update rules 31
restricted systems 29
result column
  join operation 384
  naming with AS clause 365
result set locator
  assembler 558
  C/C++ 584
  COBOL 650
  Fortran 688
  PL/I 706
result sets
  receiving from a stored procedure 768
result table
  description 364
  example 364
  numbering rows 367
  of SELECT statement 364
  read-only 406
result tables
  formatting 364
retrieving
  data in ASCII from Db2 for z/OS 504
  data in Unicode from Db2 for z/OS 504
  data using SELECT * 390
  data, changing the CCSID 447
  large volumes of data 447
  multiple rows into host-variable arrays 493
retrieving a single row
  into host variables 487
return code
  CAF 57
  DSN command 933
  RRSAF 108
RETURN statement
  returning SQL procedure status 767
REXX
  creating stored procedure 257
REXX application program
  including SQLCA 742
  SQLCODE host variable 742
  SQLSTATE host variable 742
REXX program
  application programming interface
    CONNECT 745
    DISCONNECT 745
    EXECSQL 745
  character input data 747
  data type conversion 743
  DSNREXX 745
  error handling 725
  input data type 743
  isolation level 748
  naming convention 725
  naming cursors 750
  naming prepared statements 750
  running 936
  SQLDA 476, 743
  statement label 725
RIB (release information block)
  CONNECT function of CAF 47
  IDENTIFY function of RRSAF 76
  SET_REPLICATION function of RRSAF 100
RID
  for direct row access 432
  RID function 432
RIGHT OUTER JOIN clause 386
RMODE link-edit option 922
ROLB call, IMS 21, 24
ROLL call, IMS 21, 24
rollback
  changes within a unit of work 27
ROLLBACK option
  CICS SYNCPOINT command 20
Sample application (continued)

- DRDA access 633
- DRDA access with CONNECT statements 633
- DRDA with three-part names 638
- Dynamic SQL 574
- Environments 1026
- Languages 1026
- LOB 1024
- Organization 1024
- Phone 1024
- Programs 1008, 1024
- Project 1024
- Static SQL 574
- Stored procedure 1024
- Use 1024

User-defined function 1024

Sample applications

- Databases 1005
- Storage 1003
- Storage groups 1005
- Structure 1003

Sample applications

- TSO 1027
- Sample applications supplied 985
- Sample data 985

Sample data

- Joins 387
- Sample program
  - DSN8BC3 619
  - DSN8BD3 571
  - DSN8BE3 571
  - DSN8BF3 684
  - DSN8BP3 695

Sample tables

- DSN8C10.ACT (activity) 985
- DSN8C10.DEMO_UNICODE (Unicode sample) 996
- DSN8C10.DEPT (department) 986
- DSN8C10.EMP (employee) 988
- DSN8C10.EMP_PHOTO_RESUME (employee photo and resume) 992
- DSN8C10.EMPPROJACT (employee-to-project activity) 995
- DSN8C10.PROJ (project) 993
- PROJACT (project activity) 994
- Relationships 997
- Storage 1003
- Views 999

SAVEPOINT statement 27
- Savepoints 27

Sample application supplied

- Declaring 609
- Referencing in SQL statements 608

Scrollable cursor

- Comparison of types 417
- DSN2 for z/OS down-level requester 779
- Dynamic
  - Dynamic model 402
  - Fetching current row 420
- Retrieving rows 416
- Sensitive dynamic 402
- Sensitive static 402
- Sensitivity 417

S

Sample application

- Rollback statement description 955
- Error in IMS 450
- In a stored procedure 230
- To SAVEPOINT clause 27
- When to issue 20
- With RRSASF 67

Routine

- Inheriting special registers 204

Row

- Selecting with WHERE clause 356
- Updating 350
- Updating current 409
- Updating large volumes 352
- Row change timestamp 377
- Row_number 367
- Row-level security 130

RowID

- Data type 120
- Inserting in table 139

RowID column

- Defining 338
- Defining LOBs 122
- Inserting values into 338
- Using for direct row access 432

RowID host-variable array

- C/C++ 595
- COBOL 660
- PL/I 712

RowID variable

- Assembler 558
- C/C++ 584
- COBOL 650
- Fortran 688
- PL/I 706

Rowset

- Deleting current 412
- Updating current 412

Rowset cursor

- Closing 415
- Db2 for z/OS down-level requester 780
- Declaring 411
- End-of-data condition 411
- Example 429
- Multiple-row FETCH 412
- Opening 411
- Using 410

RRSAF functions

- Summary of behavior 75

RUN subcommand of DSN

- Return code processing 933
- Running a program in TSO foreground 933

Run time libraries, DB2I

- Background processing 905
- EDITJCL processing 905

Running application program

- CICS 945
- Errors 973
- IMS 945
scrollable cursor (continued)

static
creating delete hole 420
creating update hole 420
holes in result table 420
number of rows 418
removing holes 419
static model 402

updatable 402

scrolling
backward through data 425
backward using identity columns 425
backward using ROWIDs 425
in any direction 417

ISPF (Interactive System Productivity Facility) 964

search condition
comparison operators 358
NOT keyword 358
SELECT statement 391
WHERE clause 358

SELECT FROM DELETE statement
description 354
retrieving
multiple rows 354
with INCLUDE clause 354

SELECT FROM INSERT statement
BEFORE trigger values 343
default values 343
description 343
inserting into view 343
multiple rows
cursor sensitivity 343
effect of changes 343
effect of SAVEPOINT and ROLLBACK 343
effect of WITH HOLD 343
processing errors 343
result table of cursor 343
using cursor 343
using FETCH FIRST 343
using INPUT SEQUENCE 343
result table 343
retrieving
BEFORE trigger values 343
default values 343
generated values 343
multiple rows 343
special registers 343
using SELECT INTO 343

SELECT FROM MERGE statement
description 342
with INCLUDE clause 342

SELECT FROM UPDATE statement
description 351
retrieving
multiple rows 351
with INCLUDE clause 343, 351

SELECT INTO
using with host variables 487

SELECT statement
AS clause
with ORDER BY clause 366
changing result format 963
clauses
DISTINCT 364

SELECT statement (continued)
classes (continued)

EXCEPT 371
FROM 356
GROUP BY 375
HAVING 376
INTERSECT 371
ORDER BY 366
UNION 371
WHERE 358
derived column with AS clause 363
filtering by time changed 377
fixed-list 502
named columns 356
ORDER BY clause
derived columns 366
with AS clause 366
parameter markers 504
search condition 391
selecting a set of rows 401
subqueries 391
unnamed columns 363
using with
* (to select all columns) 356
column-name list 356
DECLARE CURSOR statement 406, 411
varying-list 504

selecting
all columns 356
named columns 356
rows 356
some columns 356
unnamed columns 363

semicolon
default SPUFI statement terminator 956
embedded 1015

sequence numbers
COBOL application program 619
Fortran 684
PL/I 695

sequence object
creating 166
referencing 447
using across multiple tables 166

server 30

services
IBM MQ 780
Message Queue Interface (MQI) 781
SET clause of UPDATE statement 350
SET CURRENT PACKAGESET statement 871
SET ENCRYPTION PASSWORD statement 134
SET_CLIENT_ID (connection function of RRSAF)
language examples 97
syntax 97
SET_ID (connection function of RRSAF)
language examples 96
syntax 96
SET_REPLICATION (connection function of RRSAF)
language examples 100
syntax 100

setting SQL terminator
DSNTIAD 1015

shortcut keys
shortcut keys (continued)

shredding XML documents from MQ messages 785
SIGNAL statement
raising a condition 246
setting condition message text 247
SIGNON (connection function of RRSF)
language examples 82
program example 111
syntax 82
SOME quantified predicate 393
sort key
ORDER BY clause 366
ordering 366
SOURCE precompiler option 852
special register
behavior in stored procedures 231
behavior in user-defined functions and stored procedures 204
CURRENT PACKAGE PATH 870
CURRENT PACKAGESET 870
CURRENT RULES 892

SQL procedure
allowable statements 224
body 224
changing 256
conditions, handling 237
ignoring conditions 245
parameters 224
preparation using DSNTIPSMP procedure 296
SQL variable 224

SQL procedure processor (DSNTIPSMP)
result set 304

SQL procedure statement
CONTINUE handler 237
EXIT handler 237
handler 237
handling errors 237

SQL procedures
creating versions 251
declaring cursors 236
nested compound statements 234
variables 225

SQL processing options
planning for 13

SQL statement nesting
restrictions 400
stored procedures 400
user-defined functions 400

SQL statement terminator
modifying in DSNTIPS2 and DSNTIPS4 1009, 1017
modifying in DSNTIAD 1015
modifying in SPUFI 956
specifying in SPUFI 956

SQL statements
ALTER FUNCTION 176
checking for successful execution 475
CLOSE 410, 415, 502
COBOL program sections 619
comments
assembler 552
C 571
COBOL 619
Fortran 684
PL/I 695
REXX 725
CONNECT, with DRDA access 775
continuation
assembler 552
C 571
COBOL 619
Fortran 684
PL/I 695
REXX 725
CREATE FUNCTION 176
DECLARE CURSOR
description 406, 411
example 502, 504
DELETE
description 409
example 352
DESCRIBE 504
embedded 843
error return codes 529
EXECUTE 520
EXECUTE IMMEDIATE 519

Index 1481
SQL statements (continued)

FETCH
  description 409, 412
  example 502
Fortran program sections 684

INSERT 335
labels
  assembler 552
  C 571
  COBOL 619
  Fortran 684
  PL/I 695
  REXX 725

merges
  assembler 552
  C 571
  COBOL 619
  Fortran 684
  PL/I 695
  REXX 725

MERGE
  example 341
OPEN
  description 408, 411
  example 502
PL/I program sections 695
PREPARE 520
RELEASE, with DRDA access 777
REXX program sections 725

SELECT
  description 358
  joining a table to itself 382
  joining tables 378
SELEcT FROM DELETE 354
SELEcT FROM INSERT 343
SELEcT FROM MERGE 342
SELEcT FROM UPDATE 351
set symbols 552
UPDATE
  description 409, 412
  example 350
WHENEVER 533

SQL table functions 182
SQL terminator, specifying in DSNTEP2 and DSNTEP4 1009, 1017

SQL terminator, specifying in DSNTIAD 1015
SQL variable 224
SQL-INIT-FLAG, resetting 619

SQLCA (SQL communication area) (continued)
  DSNTIAR subroutine (continued)
    PL/I 695
      sample C program 574
SQLCA (SQL communications area)
  assembler 555
    C/C++ 582
    COBOL 648
    deciding whether to include 475
    Fortran 686
    PL/I 704
    REXX 742
SQLCODE
  -923 895
  -925 450
  -926 450
  +100 533
  +802 540
  values 532
SQLCODE host variable
  deciding whether to declare 475
SQLDA
  setting an XML host variable 504
  XML column 504
SQLDA (SQL descriptor area)
  allocating storage 412, 504
  assembler 476, 556
  assembler program 504
  C 504
  C/C++ 476, 583
  COBOL 476, 649
  declaring 412
  dynamic SELECT example 504
  for LOBs and distinct types 504
  Fortran 476, 687
  multiple-row FETCH statement 412
  no occurrences of SQLVAR 504
  OPEN statement 502
  parameter markers 504
  PL/I 476, 504, 705
  requires storage addresses 504
  REXX 476, 743
  setting output fields 412
  storing parameter marker information 525
  varying-list SELECT statement 504

SQLERROR clause of WHENEVER statement 533

SQLLEVEL precompiler option 852

SQLN field of SQLDA 504
SQLRULES, option of BIND PLAN subcommand 892

SQLSTATE
  "01519" 540
  "2D521" 450
  "57015" 895
  values 532
SQLSTATE host variable
  deciding whether to declare 475

SQLSTATEs
  web service consumer 813
SQLVAR field of SQLDA 504
SQLWARNING clause of WHENEVER statement 533

SSID (subsystem identifier), specifying 905
static SQL
  C/C++ application
  program
stored procedures (continued)
  recording debugging messages 972
  running concurrently 766
  syntax for invoking from command line processor 943
storm drain effect 117
string
  data type 120
  structure array host variable
    declaring 609
    referencing in SQL statements 608
subquery
  basic predicate 393
  conceptual overview 391
  correlated
    DELETE statement 396
    description 395
    example 395
    UPDATE statement 396
  subquery
    DELETE statement 396
    description 391
    EXISTS predicate 393
    IN predicate 393
    quantified predicate 393
    referential constraints 397
    restrictions with DELETE 397
    UPDATE statement 396
subsystem
  identifier (SSID), specifying 905
  subsystem parameters 766
  summarizing group values 375
  supplied application programs 985
  SWITCH TO (connection function of RRSAF)
    language examples 80
    syntax 80
SYNC call, IMS 24
synchronization call abends 450
SYNCPPOINT command of CICS 20
syntax diagram
  how to read xiv
SYSDUMMY1 1007
SYSDUMMYA 1007
SYSDUMMYE 1007
SYSDUMMYU 1007
SYSIBM.MQPOLICY_TABLE
  column descriptions 785
SYSIBM.MQSERVICE_TABLE
  column descriptions 785
SYSLIB data sets 897
SYSPRINT precompiler output
  options section 975
  source statements section, example 975
  summary section, example 975
  symbol cross-reference section 975
  used to analyze errors 975
SYSTERM output to analyze errors 974
T
  table
    altering 134
  changing definitions 134
  using CREATE and ALTER 540
  copying from remote locations 773
  declaring in a program 463

Index 1483
table (continued)
  deleting rows 352
  dependent, cycle restrictions 129
  displaying, list of 355
  DROP statement 141
  filling with test data 950
  incomplete definition of 140
  inserting multiple rows 337
  inserting single row 335
  merging rows 341
  populating 950
  referential structure 128
  retrieving 401
  selecting values as you delete rows 354
  selecting values as you insert rows 343
  selecting values as you merge rows 342
  selecting values as you update rows 351
  temporary 137
  updating rows 350
  using three-part table names 773

  table and view declarations
  including in an application program 471
  table and view declarationsgenerating with DCLGEN 464
  table declarations
  adding to libraries 472
  table locator
  assembler 558
  C/C++ 584
  COBOL 650
  PL/I 706
  table space
  not logged
  recovering 28
  table spaces
  for sample applications 1005
  tables
  creating for data integrity 126
  TCB (task control block)
  capabilities with CAF 37
  capabilities with RRSAF 67
  temporary table
  advantages of 137
  working with 137
  terminal monitor program (TMP) 933
  TERMINATE IDENTIFY (connection function of RRSAF)
  language examples 104
  program example 111
  syntax 104
  TERMINATE THREAD (connection function of RRSAF)
  language examples 103
  program example 111
  syntax 103
  TEST command of TSO 978
  test environment, designing 933
  test tables 947
  test views of existing tables 947
  TIME precompiler option 852
  time that row was changed
  determining 447
  TMP (terminal monitor program)
  DSN command processor 933
  running under TSO 944
  trace field QW0366FN 828
  trace field QW0376FN 828
  transition table, trigger 149
  transition variable, trigger 149
  TRANSLATE (connection function of CAF)
    description 46
    language example 56
    program example 59
    syntax 56
  TRANSLATE (connection function of RRSAF)
    language examples 106
    syntax 106
  translating requests into SQL 541
  trigger
    activation order 161
    activation time 149
    cascading 160
    coding 149
    data integrity 163
    delete 149
    description 149
    FOR EACH ROW 149
    FOR EACH STATEMENT 149
    granularity 149
    insert 149
    interaction with constraints 161
    interaction with security label columns 163
    invoking stored procedure 156
    invoking user-defined function 156
    naming 149
    parts example 149
    parts of 149
    passing transition tables 156
    subject table 149
    transition table 149
    transition variable 149
    triggering event 149
    update 149
    using identity columns 149
    with row-level security 163
  troubleshooting
    errors for output host variables 482
  TRUNCATE
    example 352
  truncated
     determining value of output host variable 489
  TSO
    CLISTS
      calling application programs 945
      running in foreground 945
    TEST command 978
    TWOPASS precompiler option 852
  U
  Unicode
    data, retrieving from Db2 for z/OS
    sample table 996
  Unified Debugger
    debugging stored procedures 970
    setting up 970
  UNION
    keeping duplicate rows with ALL 371
  UNION clause
    columns of result table 371
variable array (continued)  
   PL/I 712
variables  
   in SQL procedures 225
version  
   changing for SQL procedure 256
   version of a package 865
   VERSION precompiler option 852, 865
   versions  
   procedures 251
view  
   contents 140, 142
   declaring in a program 463  
   description 141
   dropping 143
   identity columns 142
   join of two or more tables 142  
   referencing special registers 142
   retrieving 401
   summary data 142
   union of two or more tables 142
   using  
   deleting rows 352
   inserting rows 335
   updating rows 350

W  
web service consumer  
   SQLSTATEs 813
WHENEVER statement  
   assembler 552
   C 571
   COBOL 619
   CONTINUE clause 533
   Fortran 684
   GO TO clause 533
   NOT FOUND clause 408, 533
   PL/I 695
   specifying 533
   SQL error codes 533
   SQLERROR clause 533
   SQLWARNING clause 533
WHERE clause  
   SELECT statement  
   description 358
   joining a table to itself 382
   joining tables 378
WITH clause  
   common table expressions 144
WITH HOLD clause  
   and CICS 405
   and IMS 405
   DECLARE CURSOR statement 405
   restrictions 405
   WITH HOLD cursor  
   effect on dynamic SQL 520
   write-down privilege 163

X  
XML data  
   embedded SQL applications 541
   retrieving from tables, embedded SQL applications 549
   updating, embedded SQL applications 547
XML file reference variable  
   assembler 558
   C/C++ 584, 595
   COBOL 650, 660
   PL/I 706, 712
XML host variable  
   SQLDA 504
XML host-variable array  
   C/C++ 595
   COBOL 660
   PL/I 712
XML values  
   fetching 421
XML variable  
   assembler 558
   C/C++ 584
   COBOL 650
   PL/I 706
XMLEXISTS  
   description 448
   example 448
XMLQUERY  
   description 363
   example 363
XPath  
   XPath contexts 363
   XPath contexts
   XMLEXISTS 448
   XPath expressions 363
   XREF precompiler option 852