DB2®

DB2 Version 9
for Linux, UNIX, and Windows

# Developing Embedded SQL Applications

SC10-4232-00

DB2®

IBM

**DB2 Version 9**
for Linux, UNIX, and Windows

# Developing Embedded SQL Applications

SC10-4232-00

Before using this information and the product it supports, be sure to read the general information under *Notices*.

**Edition Notice**

You can order IBM publications online or through your local IBM representative.

- To order publications online, go to the IBM Publications Center at www.ibm.com/shop/publications/order
- To find your local IBM representative, go to the IBM Directory of Worldwide Contacts at www.ibm.com/planetwide

To order DB2 publications from DB2 Marketing and Sales in the United States or Canada, call 1-800-IBM-4YOU (426-4968).

# Contents

# Part 1. Developing embedded SQL client applications

# Chapter 1. Introduction to embedded SQL

## Introduction to embedded SQL

Embedded SQL database applications connect to databases and execute embedded SQL statements. Embedded SQL statements are embedded within a host language application. Embedded SQL database applications support the embedding of SQL statements to be executed statically or dynamically.

You can develop embedded SQL applications for DB2® in the following host programming languages: C, C++, COBOL, FORTRAN, and REXX.

**Note:** Support for embedded SQL in FORTRAN and REXX has been deprecated and will remain at the DB2 Universal Database™ , Version 5.2 level.

Building embedded SQL applications involves two prerequisite steps prior to application compilation and linking.

- Preparing the source files containing embedded SQL statements using the DB2 precompiler.

  The `PREP` (`PRECOMPILE`) command is used to invoke the DB2 precompiler, which reads your source code, parses and converts the embedded SQL statements to DB2 run-time services API calls, and finally writes the output to a new modified source file. The precompiler produces access plans for the SQL statements, which are stored together as a package within the database.
- Binding the statements in the application to the target database.

  Binding is done by default during precompilation (the `PREP` command). If binding is to be deferred (for example, running the `BIND` command later), then the `BINDFILE` option needs to be specified at `PREP` time in order for a bind file to be generated.

Once you have precompiled and bound your embedded SQL application, it is ready to be compiled and linked using the host language-specific development tools.

To aid in the development of embedded SQL applications, you can refer to the embedded SQL template in C. Examples of working embedded SQL sample applications can also be found in the `%DB2PATH%\SQLLIB\samples` directory.

**Note:** %DB2PATH% refers to the DB2 installation directory

**Static and dynamic SQL:**

SQL statements can be executed in one of two ways: statically or dynamically.

**Statically executed SQL statements**

For statically executed SQL statements, the syntax is fully known at precompile time. The structure of an SQL statement must be completely specified for a statement to be considered static. For example, the names for the columns and tables referenced in a statement must be fully known at precompile time. The only information that can be specified at run time are values for any host variables referenced by the statement. However, host variable information, such as data types, must still be precompiled. You precompile, bind, and compile statically executed SQL statements before you run your application. Static SQL is best used on databases whose statistics do not change a great deal.

**Dynamically executed SQL statements**

Dynamically executed SQL statements are built and executed by an application at run-time. An interactive application that prompts the end user for key parts of an SQL statement, such as the names of the tables and columns to be searched, is a good example of a situation suited for dynamic SQL.

**Related concepts:**

- "Embedded SQL statements in REXX applications" on page 9
- "Embedded SQL statements in C and C++ applications" on page 5
- "Embedded SQL statements in COBOL applications" on page 6
- "Embedded SQL statements in FORTRAN applications" on page 8

**Related reference:**

- "PRECOMPILE command" in *Command Reference*

## Embedding SQL statements in a host language

### Embedding SQL statements in a host language

Structured Query Language (SQL) is a standardized language which can be used to manipulate database objects and the data they contain. Despite differences between host languages, embedded SQL applications are all made up of three main elements which are required to setup and execute an SQL statement:

1. A DECLARE SECTION for declaring host variables. The declaration of the SQLCA structure does not need to be in the DECLARE section.
2. The main body of the application, the setup and execution of SQL statements.
3. Placements of logic that either commit or rollback the changes made by the SQL statements.

For each host language, there are differences between the general guidelines which apply to all languages, and rules that are specific to individual languages.

**Related concepts:**

- "Embedded SQL statements in C and C++ applications" on page 5
- "Embedded SQL statements in COBOL applications" on page 6
- "Embedded SQL statements in FORTRAN applications" on page 8
- "Embedded SQL statements in REXX applications" on page 9

## Embedded SQL statements in C and C++ applications

Embedded SQL C and C++ applications consist of three main elements to setup and execute an SQL statement.

- A DECLARE SECTION for declaring host variables. The declaration of the SQLCA structure does not need to be in the DECLARE section.
- The main body of the application, the setup and execution of SQL statements
- Placements of logic that either commit or rollback the changes made by the SQL statements

**Correct C and C++ element syntax**

| | |
|---|---|
| **Statement initializer** | EXEC SQL |
| **Statement string** | Any valid SQL statement |
| **Statement terminator** | Semicolon (;) |

For example, to execute an SQL statement statically within a C application, you might include the following within your application code:

```
   EXEC SQL SELECT col INTO :hostvar FROM table;
```

The following example demonstrates how to execute an SQL statement dynamically using the host variable `stmt1`:

```
 strcpy(stmt1, "CREATE TABLE table1(col1 INTEGER)");
 EXEC SQL EXECUTE IMMEDIATE :stmt1;
```

The following guidelines and rules apply to the execution of embedded SQL statements in C and C++ applications:

- You can begin the SQL statement string on the same line as the EXEC SQL statement initializer.
- Do not split the EXEC SQL between lines.
- You must use the SQL statement terminator. If you do not use it, the precompiler will continue to the next terminator in the application. This can cause indeterminate errors.
- C and C++ comments can be placed before the statement initializer or after the statement terminator.
- Multiple SQL statements and C or C++ statements may be placed on the same line. For example:

  ```
  EXEC SQL OPEN c1; if (SQLCODE >= 0) EXEC SQL FETCH c1 INTO :hv;
  ```
- Carriage returns, line feeds, and TABs can be included within quoted strings. The SQL precompiler will leave these as is.
- Do not use the #include statement to include files containing SQL statements. SQL statements are precompiled before the module is compiled. The precompiler will ignore the #include statement. Instead, use the SQL INCLUDE statement to import the include files.
- SQL comments are allowed on any line that is part of an embedded SQL statement, with the exception of dynamically executed statements.
  - The format for an SQL comment is a double dash (`--`), followed by a string of zero or more characters, and terminated by a line end.
  - Do not place SQL comments after the SQL statement terminator. These SQL comments cause compilation errors because compilers interpret them as C or C++ syntax.

  - You can use SQL comments in a static statement string wherever blanks are allowed.
  - The use of C and C++ comment delimiters `/* */` are allowed in both static and dynamic embedded SQL statements.
  - The use of `//`-style C++ comments are not permitted within static SQL statements
- SQL string literals and delimited identifiers can be continued over line breaks in C and C++ applications. To do this, use a back slash (\) at the end of the line where the break is desired. For example, to select data from the NAME column in the staff table where the NAME column equals 'Sanders' you could do something similar to the following:

  ```
  EXEC SQL SELECT "NA\
  ME" INTO :n FROM staff WHERE name='Sa\
  nders';
  ```

  Any new line characters (such as carriage return and line feed) are not included in the string that is passed to the database manager as an SQL statement.
- Substitution of white space characters, such as end-of-line and TAB characters, occurs as follows:
  - When they occur outside quotation marks (but inside SQL statements), end-of-lines and TABs are substituted by a single space.
  - When they occur inside quotation marks, the end-of-line characters disappear, provided the string is continued properly for a C program. TABs are not modified.

  Note that the actual characters used for end-of-line and TAB vary from platform to platform. For example, UNIX® and Linux® based systems use a line feed.

**Related concepts:**
- "Comments in embedded SQL applications" on page 136
- "Embedded SQL dynamic statements" on page 18

**Related reference:**
- "Supported SQL Statements" in *Developing SQL and External Routines*

## Embedded SQL statements in COBOL applications

Embedded SQL statements in COBOL applications consist of the following three elements:

| **Element** | **Correct COBOL Syntax** |
|---|---|
| **Statement initializer** | EXEC SQL |
| **Statement string** | Any valid SQL statement |
| **Statement terminator** | END-EXEC. |

For example:

```
EXEC SQL SELECT col INTO :hostvar FROM table END-EXEC.
```

The following rules apply to embedded SQL statements in COBOL applications:
- Executable SQL statements must be placed in the `PROCEDURE DIVISION` section. The SQL statements can be preceded by a paragraph name, just as a COBOL statement.

- SQL statements can begin in either `Area A` (columns 8 through 11) or `Area B` (columns 12 through 72).
- Start each SQL statement with the statement initializer EXEC SQL and end it with the statement terminator END-EXEC. The SQL precompiler includes each SQL statement as a comment in the modified source file.
- You must use the SQL statement terminator. If you do not use it, the precompiler will continue to the next terminator in the application. This may cause indeterminate errors.
- SQL comments are allowed on any line that is part of an embedded SQL statement. These comments are not allowed in dynamically executed statements. The format for an SQL comment is a double dash (`--`), followed by a string of zero or more characters and terminated by a line end. Do not place SQL comments after the SQL statement terminator as they will cause compilation errors because they would appear to be part of the COBOL language.
- COBOL comments are allowed in most places. The exceptions are:
  - Comments are not allowed between EXEC and SQL.
  - Comments are not allowed in dynamically executed statements.
- SQL statements follow the same line continuation rules as the COBOL language. However, do not split the EXEC SQL statement initializer between lines.
- Do not use the COBOL COPY statement to include files containing SQL statements. SQL statements are precompiled before the module is compiled. The precompiler will ignore the COBOL COPY statement. Instead, use the SQL INCLUDE statement to import the include files.
- To continue a string constant to the next line, column 7 of the continuing line must contain a '-' and column 12 or beyond must contain a string delimiter.
- SQL arithmetic operators must be delimited by blanks.
- Substitution of white space characters, such as end-of-line and TAB characters, occurs as follows:
  - When they occur outside quotation marks (but inside SQL statements), end-of-lines and TABs are substituted by a single space.
  - When they occur inside quotation marks, the end-of-line characters disappear, provided the string is continued properly for a COBOL program. TABs are not modified.

  Note that the actual characters used for end-of-line and TAB vary from platform to platform. For example, Windows®-based platforms use Carriage Return/Line Feed for end-of-line, whereas UNIX and Linux based systems use just a Line Feed.

**Related concepts:**
- "Comments in embedded SQL applications" on page 136
- "Restrictions on using COBOL to program embedded SQL applications" on page 25

**Related reference:**
- "Supported SQL Statements" in *Developing SQL and External Routines*
- "INCLUDE statement" in *SQL Reference, Volume 2*
- "Include files for COBOL embedded SQL applications" on page 45

## Embedded SQL statements in FORTRAN applications

Embedded SQL statements in FORTRAN applications consist of the following three elements:

| **Element** | **Correct FORTRAN Syntax** |
|---|---|
| **Statement initializer** | EXEC SQL |
| **Statement string** | Any valid SQL statement with blanks as delimiters |
| **Statement terminator** | End of source line. |

The end of the source line serves as the statement terminator. If the line is continued, the statement terminator is the end of the last continued line.

For example:

```
EXEC SQL SELECT COL INTO :hostvar FROM TABLE
```

The following rules apply to embedded SQL statements in FORTRAN applications:

- Code SQL statements between columns 7 and 72 only.
- Use full-line FORTRAN comments, or SQL comments, but do not use the FORTRAN end-of-line comment '!' character in SQL statements. This comment character may be used elsewhere, including host variable declarations.
- Use blanks as delimiters when coding embedded SQL statements, even though FORTRAN statements do not require blanks as delimiters.
- Use only one SQL statement for each FORTRAN source line. Normal FORTRAN continuation rules apply for statements that require more than one source line. Do not split the EXEC SQL statement initializer between lines.
- SQL comments are allowed on any line that is part of an embedded SQL statement. These comments are not allowed in dynamically executed statements. The format for an SQL comment is a double dash (`--`), followed by a string of zero or more characters and terminated by a line end.
- FORTRAN comments are allowed *almost* anywhere within an embedded SQL statement. The exceptions are:
  - Comments are not allowed between EXEC and SQL.
  - Comments are not allowed in dynamically executed statements.
  - The extension of using `!` to code a FORTRAN comment at the end of a line is not supported within an embedded SQL statement.
- Use exponential notation when specifying a real constant in SQL statements. The database manager interprets a string of digits with a decimal point in an SQL statement as a decimal constant, not a real constant.
- Statement numbers are invalid on SQL statements that precede the first executable FORTRAN statement. If an SQL statement has a statement number associated with it, the precompiler generates a labeled CONTINUE statement that directly precedes the SQL statement.
- Use host variables exactly as declared when referencing host variables within an SQL statement.
- Substitution of white space characters, such as end-of-line and TAB characters, occurs as follows:
  - When they occur outside quotation marks (but inside SQL statements), end-of-lines and TABs are substituted by a single space.

– When they occur inside quotation marks, the end-of-line characters disappear, provided the string is continued properly for a FORTRAN program. TABs are not modified.

Note that the actual characters used for end-of-line and TAB vary from platform to platform. For example, Windows-based platforms use the Carriage Return/Line Feed for end-of-line, whereas UNIX and Linux based platforms use just a Line Feed.

**Related concepts:**

- "Comments in embedded SQL applications" on page 136
- "Concurrent transactions and multi-threaded database access in embedded SQL applications" on page 27
- "Host variable names in FORTRAN embedded SQL applications" on page 120
- "Restrictions on using FORTRAN to program embedded SQL applications" on page 25

**Related reference:**

- "Supported SQL Statements" in *Developing SQL and External Routines*

## Embedded SQL statements in REXX applications

REXX applications use APIs that enable them to use most of the features provided by database manager APIs and SQL. Unlike applications written in a compiled language, REXX applications are not precompiled. Instead, a dynamic SQL handler processes all SQL statements. By combining REXX with these callable APIs, you have access to most of the database manager capabilities. Although REXX does not directly support some APIs using embedded SQL, they can be accessed using the DB2 command line processor from within the REXX application.

As REXX is an interpreted language, you will find it is easier to develop and debug your application prototypes in REXX, as compared to compiled host languages. Although database applications coded in REXX do not provide the performance of database applications that use compiled languages, they do provide the ability to create database applications without precompiling, compiling, linking, or using additional software.

Use the SQLEXEC routine to process all SQL statements. The character string arguments for the SQLEXEC routine are made up of the following elements:

- SQL keywords
- Pre-declared identifiers
- Statement host variables

Make each request by passing a valid SQL statement to the SQLEXEC routine. Use the following syntax:

```
CALL SQLEXEC 'statement'
```

SQL statements can be continued onto more than one line. Each part of the statement should be enclosed in single quotation marks, and a comma must delimit additional statement text as follows:

```
CALL SQLEXEC 'SQL text',
             'additional text',
                  .
                  .
                  .
             'final text'
```

The following is an example of embedding an SQL statement in REXX:

```
statement = "UPDATE STAFF SET JOB = 'Clerk' WHERE JOB = 'Mgr'"
CALL SQLEXEC 'EXECUTE IMMEDIATE :statement'
IF ( SQLCA.SQLCODE < 0) THEN
   SAY 'Update Error:  SQLCODE = ' SQLCA.SQLCODE
```

In this example, the `SQLCODE` field of the `SQLCA` structure is checked to determine whether the update was successful.

The following rules apply to embedded SQL statements: in REXX applications

- The following SQL statements can be passed directly to the SQLEXEC routine:
  - CALL
  - CLOSE
  - COMMIT
  - CONNECT
  - CONNECT TO
  - CONNECT RESET
  - DECLARE
  - DESCRIBE
  - DISCONNECT
  - EXECUTE
  - EXECUTE IMMEDIATE
  - FETCH
  - FREE LOCATOR
  - OPEN
  - PREPARE
  - RELEASE
  - ROLLBACK
  - SET CONNECTION

  Other SQL statements must be processed dynamically using the EXECUTE IMMEDIATE, or PREPARE and EXECUTE statements in conjunction with the SQLEXEC routine.
- You cannot use host variables in the CONNECT and SET CONNECTION statements in REXX.
- Cursor names and statement names are predefined as follows:

  **c1 to c100**

  Cursor names, which range from *c1* to *c50* for cursors declared without the WITH HOLD option, and *c51* to *c100* for cursors declared using the WITH HOLD option.

  The cursor name identifier is used for DECLARE, OPEN, FETCH, and CLOSE statements. It identifies the cursor used in the SQL request.

  **s1 to s100**

  Statement names, which range from *s1* to *s100*.

  The statement name identifier is used with the DECLARE, DESCRIBE, PREPARE, and EXECUTE statements.

  The pre-declared identifiers must be used for cursor and statement names. Other names are not allowed.
- When declaring cursors, the cursor name and the statement name should correspond in the DECLARE statement. For example, if *c1* is used as a cursor name, *s1* must be used for the statement name.
- Do not use comments within an SQL statement.

**Note:** REXX does not support multi-threaded database access.

**Related concepts:**
- "API Syntax for REXX" on page 157
- "Concurrent transactions and multi-threaded database access in embedded SQL applications" on page 27
- "Restrictions on using REXX to program embedded SQL applications" on page 26

**Related reference:**
- "REXX samples" on page 377

## Supported development software for embedded SQL applications

DB2 database systems support compilers, interpreters, and related development software for embedded SQL applications in the following operating systems:
- AIX®
- HP-UX
- Linux
- Solaris
- Windows

32-bit and 64-bit embedded SQL applications can be built from embedded SQL source code.

The following host languages require specific compilers to develop embedded SQL applications:
- C
- C++
- COBOL
- Fortran
- REXX

**Related concepts:**
- "32-bit and 64-bit support for embedded SQL applications" on page 23
- "Restrictions on using COBOL to program embedded SQL applications" on page 25
- "Restrictions on using FORTRAN to program embedded SQL applications" on page 25

## Setting up the embedded SQL development environment

Before you can start building embedded SQL applications, you need to install the supported compiler for the host language you will be using to develop your applications and set up the embedded SQL environment.

**Prerequisites:**
- DB2 database server installed on a supported platform
- DB2 Client installed
- Supported embedded SQL application development software installed

**Procedure:**

Assign the user the authority to issue the PREP command and BIND command.

To verify that the embedded SQL application development environment is set up properly, try building and running the embedded SQL application template found in the topic: Embedded SQL application template in C.

**Related concepts:**

- "Embedded SQL application template in C" on page 39
- "Building embedded SQL applications using the sample build script" on page 200
- "Building embedded SQL applications" on page 180
- "Building embedded SQL applications from the command line" on page 241
- "Precompilation of embedded SQL applications with the PRECOMPILE command" on page 182
- "Bind considerations" on page 195

**Related reference:**

- "Support for elements of the database application development environment" in *Getting Started with Database Application Development*
- "GRANT (Package Privileges) statement" in *SQL Reference, Volume 2*

# Chapter 2. Designing embedded SQL applications

## Designing embedded SQL applications

When designing embedded SQL applications you must make use of either statically or dynamically executed SQL statements. Static SQL statements come in two flavors: statements that contain no host variables (used mainly for initialization and simple SQL examples), and statements that make use of host variables. Dynamic SQL statements also come in two flavors: they can either contain no parameter markers (typical of interfaces such as CLP) or contain parameter markers, which allows for greater flexibility within applications.

The choice of whether to use statically or dynamically executed statements depend on a number of factors, including: portability, performance and restrictions of embedded SQL applications.

**Related concepts:**

- "32-bit and 64-bit support for embedded SQL applications" on page 23
- "Concurrent transactions and multi-threaded database access in embedded SQL applications" on page 27
- "Performance of embedded SQL applications" on page 22
- "Static and dynamic SQL statement execution in embedded SQL applications" on page 17

## Static SQL usage

Static SQL is generally used in embedded SQL database applications. Static SQL statements are hard-coded in an embedded SQL application program when the source code is written. The source code must be processed by the DB2 database manager using a SQL precompiler before it can be compiled and executed. During this process, the SQL precompiler evaluates references to tables, columns, and declared data types of all host variables and determines which data conversion methods need to be used when data is moved to and from the database. Additionally, the SQL precompiler performs error checking on each coded SQL statement and ensures that appropriate host variable data types are used for their respective table column values. Static SQL is called static because the SQL statements in the program do not change each time the program is run. Static SQL works well in many situations. In fact, it can be used in any application for which the data access can be determined at program design time. For example, an order entry program always uses the same statement to insert a new order, and an airline reservation system always uses the same statement to change the status of a seat from 'available' to 'reserved'. Each of these statements would be generalized through the use of host variables. Different values can be inserted in a sales order and different seats can be reserved. Because such statements can be hard-coded in the program, such programs have the advantage that statements need to be parsed, validated, and optimized only once, at compile time. This results in relatively fast code at runtime.

Although static SQL statements are easy to use, they are limited because their format must be known in advance by the precompiler and because they can only work with host variables. There are many situations in which the content of an SQL statement is not known by the programmer in advance. For example, suppose a spreadsheet allows a user to enter a query, which the spreadsheet then sends to the database management system to retrieve data. The contents of this query obviously cannot be known to the programmer when the spreadsheet program is written.

Static SQL feature details are provided below:

**Interfaces that support static SQL execution::**

Static SQL can be executed from the following interfaces:
- Embedded SQL applications
- Embedded SQL routines
- SQLJ applications
- SQLJ routines
- SQL routines

**Ease of programming implementation:**

In general programming using static SQL is easier than using embedded dynamic SQL. Static SQL statements are simply embedded into the host language source file, and the precompiler handles the necessary conversion to database manager run-time service API calls that the host language compiler can process. Within embedded SQL programs, there are two main implementations of static SQL:
- Static SQL containing no host variables
  – This type of implementation is generally rare, and used mainly for initialization code and in simple SQL examples. Static SQL without host variables performs very well, because with no host variables to resolve, there is no run-time performance overhead, and the DB2 optimizer's capabilities can be fully realized.
- Static SQL containing host variables
  – This is the traditional legacy style of static SQL. Although static SQL with host variables performs better at run time than dynamic SQL which requires statement preparation and the establishment of catalog locks during statement compilation, the full power of the optimizer cannot be utilized because the optimizer does not know the entire SQL statement. This can be problematic when the data a SQL statement operates on has a highly non-uniform data

distribution. For this reason static SQL without host variables can perform better. Of course host variables provide support for binding variable values to a statement.

**Performance:**

In general executing a query as static SQL will ensure good query performance. Both forms of static SQL statements described above can be used to achieve better query performance than is possible when dynamic SQL is used. For simple, short-running SQL programs, a static SQL statement executes faster than the same statement processed dynamically because the overhead of preparing an executable form of the statement is done at precompile time instead of at run time.

The performance of static SQL depends on the statistics of the database the last time the application was bound. However, if these statistics change, the performance of equivalent dynamic SQL can be very different. If, for example, an index is added to a database at a later time, an application using static SQL cannot take advantage of the index unless it is rebound to the database. In addition, if you are using host variables in a static SQL statement, the optimizer will not be able to take advantage of any information about data distribution.

**Authorization:**

The authorization of the person that binds an application package is used at runtime to validate the privileges required to execute the static SQL statements used in the application. This means that the end user running the application does not directly require the privileges to execute the individual statements in the package. For example, consider an application that contains SQL statements that update a table storing inventory data. Once the package is bound to the database by someone with the necessary privileges to update the table, other users will be able to run the application and thereby make updates to the inventory table.

The combination of static SQL features make it an appropriate choice for applications where SQL statement syntax is known at application programming time and in cases where SQL statement performance is critical to application performance. It can also make application programming code easier to maintain and simply the management of authorizations required to execute SQL within applications.

## Dynamic SQL usage

Dynamic SQL is generally used for sending SQL statements to the database manager from interactive query building graphical user interfaces and SQL command line processors as well as from applications where the complete structure of queries is not known at application compilation time and the programming API supports dynamic SQL.

There are many situations in which the content of an SQL statement is not known by a user or programmer in advance. For example, suppose a spreadsheet allows a user to enter a query, which the spreadsheet then sends to the database management system to retrieve data. The contents of this query obviously cannot be known to the programmer when the spreadsheet program is written. To solve this problem, the spreadsheet uses a form of embedded SQL called dynamic SQL. Unlike static SQL statements, which are hard-coded in the program, Dynamic SQL statements can be built at run time and placed in a string host variable. They are then sent to the database management system for processing. Because the database management system must generate an access plan at run time for dynamic SQL statements, dynamic SQL is generally slower than static SQL. When a program containing dynamic SQL statements is compiled, the dynamic SQL statements are not stripped from the program, as in static SQL. Instead, they are replaced by a function call that passes the statement to the DBMS.

Since the actual creation of dynamic SQL statements is based upon the flow of programming logic at application run time, they are more powerful than Static SQL statements. However, because the DBMS must go through each of the multiple steps of processing a SQL statement for each dynamic request, Dynamic SQL tends to be slower than Static SQL. DB2 database manager provides support internally for a dynamic statement cache which automatically assists in improving the performance of dynamic SQL queries.

Dynamic SQL can be used from the following interfaces:

- Applications and external routines that employ APIs that support dynamic SQL including: Embedded SQL, JDBC,CLI, ADO.NET
- Interactive DB2 GUI interfaces including: DB2 Command Editor in the DB2 Control Center
- DB2 Command Line Processor
- DB2 Command Windows

## Authorization Considerations for Embedded SQL

An *authorization* allows a user or group to perform a general task such as connecting to a database, creating tables, or administering a system. A *privilege* gives a user or group the right to access one specific database object in a specified way. DB2® uses a set of privileges to provide protection for the information that you store in it.

Most SQL statements require some type of privilege on the database objects which the statement utilizes. Most API calls usually do not require any privilege on the database objects which the call utilizes, however, many APIs require that you possess the necessary authority in order to invoke them. The DB2 APIs enable you to perform the DB2 administrative functions from within your application program. For example, to recreate a package stored in the database without the need for a bind file, you can use the `sqlarbnd` (or REBIND) API.

Groups provide a convenient means of performing authorization for a collection of users without having to grant or revoke privileges for each user individually. Group membership is considered for the execution of dynamic SQL statements, but not for static SQL statements. `PUBLIC` privileges are, however, considered for the execution of static SQL statements. For example, suppose you have an embedded SQL stored procedure with statically bound SQL queries against a table called `STAFF`. If you try to build this procedure with the `CREATE PROCEDURE` statement, and your account belongs to a group that has the select privilege for the `STAFF` table, the `CREATE` statement will fail with a SQL0551N error. For the `CREATE` statement to work, your account directly needs the select privilege on the `STAFF` table.

When you design your application, consider the privileges your users will need to run the application. The privileges required by your users depend on:

- Whether your application uses dynamic SQL, including JDBC and DB2 CLI, or static SQL. For information about the privileges required to issue a statement, see the description of that statement.
- Which APIs the application uses. For information about the privileges and authorities required for an API call, see the description of that API.

Groups provide a convenient means of performing authorization for a collection of users without having to grant or revoke privileges for each user individually. In general, group membership is considered for dynamic SQL statements, but is not considered for static SQL statements. The exception to this general case occurs when privileges are granted to PUBLIC: these are considered when static SQL statements are processed.

Consider two users, PAYROLL and BUDGET, who need to perform queries against the STAFF table. PAYROLL is responsible for paying the employees of the company, so it needs to issue a variety of SELECT statements when issuing paychecks. PAYROLL needs to be able to access each employee's salary. BUDGET is responsible for determining how much money is needed to pay the salaries. BUDGET should not, however, be able to see any particular employee's salary.

Because PAYROLL issues many different SELECT statements, the application you design for PAYROLL could probably make good use of dynamic SQL. The dynamic SQL would require that PAYROLL have SELECT privilege on the STAFF table. This requirement is not a problem because PAYROLL requires full access to the table.

BUDGET, on the other hand, should not have access to each employee's salary. This means that you should not grant SELECT privilege on the STAFF table to BUDGET. Because BUDGET does need access to the total of all the salaries in the STAFF table, you could build a static SQL application to execute a SELECT SUM(SALARY) FROM STAFF, bind the application and grant the EXECUTE privilege on your application's package to BUDGET. This enables BUDGET to obtain the required information, without exposing the information that BUDGET should not see.

**Related concepts:**
- "Authorization" in *Administration Guide: Planning*
- "Authorization considerations for dynamic SQL" in *Developing SQL and External Routines*
- "Authorization considerations for static SQL" in *Developing SQL and External Routines*

## Static and dynamic SQL statement execution in embedded SQL applications

### Static and dynamic SQL statement execution in embedded SQL applications

Both static and dynamic SQL statement execution is supported in embedded SQL applications. The decision to execute SQL statements statically or dynamically requires an understanding of packages, how SQL statements are executed at run-time, host variables, parameter markers, and how these things are related to application performance.

**Static SQL in embedded SQL programs:**

An example of a statically executed statement in C is:

```
/* select values from table into host variables using STATIC SQL and print them*/
EXEC SQL SELECT id, name, dept, salary INTO :id, :name, :dept, :salary
           FROM staff WHERE id = 310;
printf("    %3d %-8.8s %4d %7.2f\n\n", id, name, dept, salary);
```

**Dynamic SQL in embedded SQL programs:**

An example of a dynamically executed statement in C is:

```
/* Update column in table using DYNAMIC SQL*/
strcpy(hostVarStmtDyn, "UPDATE staff SET salary = salary + 1000  WHERE dept = ?");
EXEC SQL PREPARE StmtDyn FROM :hostVarStmtDyn;
EXEC SQL EXECUTE StmtDyn USING :dept;
```

**Related concepts:**

- "Dynamic SQL usage" on page 15
- "Static SQL usage" on page 13
- "Embedding SQL statements in a host language" on page 4
- "Embedded SQL application template in C" on page 39
- "Precompilation of embedded SQL applications with the PRECOMPILE command" on page 182

## Embedded SQL dynamic statements

Dynamic SQL statements accept a character-string host variable and a statement name as arguments. The host variable contains the SQL statement to be processed dynamically in text form. The statement text is not processed when an application is precompiled. In fact, the statement text does not have to exist at the time the application is precompiled. Instead, the SQL statement is treated as a host variable for precompilation purposes and the variable is referenced during application execution.

Dynamic SQL support statements are required to transform the host variable containing SQL text into an executable form. Also, dynamic SQL support statements operate on the host variable by referencing the statement name. These support statements are:

**EXECUTE IMMEDIATE**
Prepares and executes a statement that does not use any host variables. Use this statement as an alternative to the PREPARE and EXECUTE statements.

For example consider the following statement in C:

```
strcpy (qstring,"INSERT INTO WORK_TABLE SELECT *
   FROM EMP_ACT WHERE ACTNO >= 100");
EXEC SQL  EXECUTE IMMEDIATE :qstring;
```

**PREPARE**
Turns the character string form of the SQL statement into an executable form of the statement, assigns a statement name, and optionally places information about the statement in an SQLDA structure.

**EXECUTE**
Executes a previously prepared SQL statement. The statement can be executed repeatedly within a connection.

**DESCRIBE**
Places information about a prepared statement into an SQLDA.

For example consider the following statement in C;

```
strcpy(hostVarStmt, "DELETE FROM org WHERE deptnumb = 15");
EXEC SQL PREPARE Stmt FROM :hostVarStmt;
EXEC SQL DESCRIBE Stmt INTO :sqlda;
EXEC SQL EXECUTE Stmt;
```

**Note:** The content of dynamic SQL statements follows the same syntax as static SQL statements, with the following exceptions:

- The statement cannot begin with EXEC SQL.
- The statement cannot end with the statement terminator. An exception to this is the CREATE TRIGGER statement which can contain a semicolon (;).

**Related concepts:**

- "Dynamic SQL usage" on page 15
- "Static SQL usage" on page 13
- "Precompilation of embedded SQL applications with the PRECOMPILE command" on page 182
- "Host Variables in embedded SQL applications" on page 66
- "Embedded SQL application template in C" on page 39
- "Retrieving host variable information from the SQLDA structure in embedded SQL applications" on page 138
- "Data types that map to SQL data types in embedded SQL applications" on page 53

**Related tasks:**

- "Declaring host variables in embedded SQL applications" on page 68
- "Referencing host variables in embedded SQL applications" on page 76

**Related reference:**

- "Supported SQL Statements" in *Developing SQL and External Routines*

## Determining when to execute SQL statements statically or dynamically in embedded SQL applications

There are several considerations that must be considered before determining whether to execute a SQL statement statically or dynamically in an embedded SQL application. The following tables lists the major considerations to consider and recommendations on when to use static or dynamic SQL or where either choice is equally good.

**Note:** These are general recommendations only. Your application's requirement, its intended usage, and working environment dictate the actual choice. When in doubt, prototyping your statements as static SQL, then as dynamic SQL, and comparing the differences is the best approach.

*Table 1. Comparing Static and Dynamic SQL*

| Consideration | Likely Best Choice |
|---|---|
| Desired time within which query should be executed:<br>• Less than 2 seconds<br>• 2 to 10 seconds<br>• More than 10 seconds | <br>• Static<br>• Either<br>• Dynamic |
| Uniformity of data being queried or operated upon by the SQL statement<br>• Uniform data distribution<br>• Slight non-uniformity<br>• Highly non-uniform distribution | <br>• Static<br>• Either<br>• Dynamic |
| Quantity of range predicates within the query<br>• Few<br>• Some<br>• Many | <br>• Static<br>• Either<br>• Dynamic |
| Likelihood of repeated SQL statement execution<br>• Runs many times (10 or more times)<br>• Runs a few times (less than 10 times)<br>• Runs once | <br>• Either<br>• Either<br>• Static |
| Nature of Query<br>• Random<br>• Permanent | <br>• Dynamic<br>• Either |
| Types of SQL statements (DML/DDL/DCL)<br>• Transaction Processing (DML Only)<br>• Mixed (DML and DDL - DDL affects packages)<br>• Mixed (DML and DDL - DDL does not affect packages) | <br>• Either<br>• Dynamic<br>• Either |
| Frequency with which the `RUNSTATS` command is issued<br>• Very infrequently<br>• Regularly<br>• Frequently | <br>• Static<br>• Either<br>• Dynamic |

SQL statements are always compiled before they are run. The difference is that dynamic SQL statements are compiled at runtime, so the application might be a bit slower due to the overhead of compiling each of the dynamic statements at application runtime versus during a single initial compilation stage as is the case with static SQL.

In a mixed environment, the choice between static and dynamic SQL must also factor in the frequency in which packages are invalidated. If the DDL does invalidate packages, dynamic SQL is more efficient as only those queries executed are recompiled when they are next used. Others are not recompiled. For static SQL, the entire package is rebound once it has been invalidated.

There will be times when it won't matter very much whether or not you use static SQL or dynamic SQL. For example it might be the case within an application that contains mostly references to SQL statements to be executed dynamically that there might be one statement that might more suitably be executed as static SQL. In such a case, to be consistent in your coding, it might make sense to simply execute that one statement dynamically too. Note that the considerations in the above table are listed roughly in order of importance.

Do not assume that a static version of an SQL statement automatically executes faster than the same statement processed dynamically. In some cases, static SQL is faster because of the overhead required to prepare the dynamic statement. In other cases, the same statement prepared dynamically executes faster, because the optimizer can make use of current database statistics, rather than the database statistics available at an earlier bind time. Note that if your transaction takes less than a couple of seconds to complete, static SQL will generally be faster. To choose which method to use, you should prototype both forms of binding.

**Note:** Static and dynamic SQL each come in two types, statements which make use of host variables and ones which don't. These types are:

1. Static SQL statements containing no host variables

   This is an unlikely situation which you may see only for:

   - Initialization code
   - Very simple SQL statements

   Simple SQL statements without host variables perform well from a performance perspective in that there is no run-time performance overhead, and the DB2 optimizer's capabilities can be fully realized.
2. Static SQL containing host variables

   Static SQL statements which make use of host variables are considered as the traditional style of DB2 applications. The static SQL statements avoids the run-time overhead of a PREPARE and catalog locks acquired during statement compilation. Unfortunately, the full power of the optimizer cannot be utilized because the optimizer does not know the entire SQL statement. A particular problem exists with highly non-uniform data distributions.
3. Dynamic SQL containing no parameter markers

   This is typical of interfaces such as the CLP which is often used for executing ad hoc queries. From the CLP, SQL statements can only be executed dynamically.
4. Dynamic SQL containing parameter markers

   The key benefit of dynamic SQL statements is that the presence of parameter markers allows the cost of the statement preparation to be amortized over the repeated executions of the statement, typically a select or insert. This amortization is true for all repetitive dynamic SQL applications. Unfortunately, just like static SQL with host variables, parts of the DB2 optimizer will not work because complete information is unavailable.

   The recommendation is to use static SQL with host variables or dynamic SQL without parameter markers as the most efficient options.

**Related concepts:**

- "Example of parameter markers in a dynamically executed SQL program" on page 154
- "Binding embedded SQL packages to a database" on page 190

**Related tasks:**

- "Providing variable input to dynamically executed SQL statement using parameter markers" on page 153
- "Setting up the embedded SQL development environment" on page 11
- "Declaring host variables in embedded SQL applications" on page 68
- "Referencing host variables in embedded SQL applications" on page 76

## Performance of embedded SQL applications

Performance is an important factor to consider when developing database applications. Embedded SQL applications can perform well, because they support static SQL statement execution and a mix of static and dynamic SQL statement execution. Due to how static SQL statements are compiled, there are steps that a developer or database administrator must take to ensure that embedded SQL applications continue to perform well over time.

The following factors can impact embedded SQL application performance:

- Changes in database schemas over time
- Changes in the cardinalities of tables (the number of rows in tables) over time
- Changes in the host variable values bound to SQL statements

Embedded SQL application performance is impacted by these factors because the package is created once when a database might have a certain set of characteristics. These characteristics are factored into the creation of the package run time access plans which define how the database manager will most efficiently execute SQL statements. Over time a database schema and data might change rendering the run time access plans sub-optimal. This can lead to degradation in application performance.

For this reason it is important to periodically refresh the information that is used to ensure that the package run-time access plans are well-maintained.

The `RUNSTATS` command is used to collect current statistics on tables and indexes, especially if significant update activity has occurred or new indexes have been created since the last time the `RUNSTATS` command was executed. This provides the optimizer with the most accurate information with which to determine the best access plan.

Performance of Embedded SQL applications can be improved in several ways:

- Run the `RUNSTATS` command to update database statistics.
- Rebind application packages to the database to regenerate the run time access plans (based on the updated statistics) that the database will use to physically retrieve the data on disk.
- Using the REOPT bind option in your static and dynamic programs.

**Related concepts:**

- "Improving performance by binding with REOPT" in *Performance Guide*
- "Query optimization using the REOPT bind option" in *Performance Guide*
- "Rebinding existing packages with the REBIND command" on page 194
- "Host Variables in embedded SQL applications" on page 66
- "Performance improvements when using REOPT option of the BIND command" on page 197
- "Using special registers to control the statement compilation environment" on page 193

**Related tasks:**

- "Declaring host variables in embedded SQL applications" on page 68
- "Referencing host variables in embedded SQL applications" on page 76

**Related reference:**

- "db2Runstats API - Update statistics for tables and indexes" in *Administrative API Reference*
- "RUNSTATS command" in *Command Reference*

## 32-bit and 64-bit support for embedded SQL applications

Embedded SQL applications can be built on both 32-bit and 64-bit platforms. However, there are separate building and running considerations. Build scripts contain a check to determine the bitwidth. If the bitwidth detected is 64-bit an extra set of switches is set to accommodate the necessary changes.

DB2 database systems are supported on 32-bit and 64-bit versions of operating systems listed below. There are differences for building embedded SQL applications in 32-bit and 64-bit environments in most cases on these operating systems.

- AIX
- HP-UX
- Linux
- Solaris
- Windows

The only 32-bit instances that will be supported in DB2 Version 9 are:

- Linux on x86
- Windows on x86
- Windows on X64 (when using the DB2 for Windows on x86 install image)

The only 64-bit instances that will be supported in DB2 Version 9 are:

- AIX
- Sun
- HP PA-RISC
- HP IPF
- Linux on x86_64
- Linux on POWER
- Linux on zSeries
- Windows on X64 (when using the Windows for X64 install image)
- Windows on IPF
- Linux on IPF

DB2 database systems support running 32-bit applications and routines on all supported 64-bit operating system environments except Linux IA64 and Linux zSeries®.

For each of the host languages, the host variables used can be better in either 32-bit or 64-bit platform or both. Check the various data types for each of the programming languages.

**Related tasks:**

- "Migrating embedded SQL applications" on page 243

**Related reference:**

- "Data types for procedures, functions, and methods in C and C++ embedded SQL applications" on page 58
- "Declaration of file reference type host variables in C and C++ embedded SQL applications" on page 96
- "Declaration of file reference type host variables in COBOL embedded SQL applications" on page 115
- "Declaration of file reference type host variables in FORTRAN embedded SQL applications" on page 126
- "Declaration of file reference type host variables in REXX embedded SQL applications" on page 134

## Restrictions on embedded SQL applications

### Restrictions on character sets using C and C++ to program embedded SQL applications

Some characters from the C or C++ character set are not available on all keyboards. These characters can be entered into a C or C++ source program using a sequence of three characters called a *trigraph*. Trigraphs are not recognized in SQL statements. The precompiler recognizes the following trigraphs within host variable declarations:

| Trigraph | Definition |
|---|---|
| **??(** | Left bracket '[' |
| **??)** | Right bracket ']' |
| **??<** | Left brace '{' |
| **??>** | Right brace '}' |

The remaining trigraphs listed below may occur elsewhere in a C or C++ source program:

| Trigraph | Definition |
|---|---|
| **??=** | Hash mark '#' |
| **??/** | Back slash '\' |
| **??'** | Caret '^' |
| **??!** | Vertical Bar '\|' |
| **??–** | Tilde '~' |

**Related concepts:**

- "Embedded SQL statements in C and C++ applications" on page 5
- "Host variables in C and C++ embedded SQL applications" on page 78
- "Host variable names in C and C++ embedded SQL applications" on page 79
- "Declare section for host variables in C and C++ embedded SQL applications" on page 80
- "Initialization of host variables in C and C++ embedded SQL applications" on page 101

- “Example: SQL declare section template for C and C++ embedded SQL applications” on page 81

**Related reference:**

- “C samples” on page 369
- “Supported SQL data types in C and C++ embedded SQL applications” on page 53
- “Data types for procedures, functions, and methods in C and C++ embedded SQL applications” on page 58

## Restrictions on using COBOL to program embedded SQL applications

The following are the restrictions for API calls in COBOL applications:

- All integer variables used as value parameters in API calls must be declared with a `USAGE COMP-5` clause.

In an object-oriented COBOL program:

- SQL statements can only appear in the first program or class in a compile unit. This restriction exists because the precompiler inserts temporary working data into the first `Working-Storage Section` it sees.
- Every class containing SQL statements must have a class-level `Working-Storage Section`, even if it is empty. This section is used to store data definitions generated by the precompiler.

**Related concepts:**

- “Embedded SQL statements in COBOL applications” on page 6
- “Host Variables in embedded SQL applications” on page 66
- “Host variables in COBOL” on page 107
- “Null-indicator variables and null or truncation indicator variable tables in COBOL embedded SQL applications” on page 119
- “Host variables in FORTRAN” on page 120

**Related reference:**

- “Supported SQL data types in COBOL embedded SQL applications” on page 60
- “COBOL samples” on page 373

## Restrictions on using FORTRAN to program embedded SQL applications

Embedded SQL support for FORTRAN was stabilized in DB2 UDB Version 5, and no enhancements are planned for the future. For example, the FORTRAN precompiler cannot handle SQL object identifiers, such as table names, that are longer than 18 bytes. To use features introduced to DB2 database systems after UDB Version 5, such as table names from 19 to 128 bytes long, you must write your applications in a language other than FORTRAN.

FORTRAN database application development is not supported with DB2 instances in Windows or Linux environments.

FORTRAN does not support multi-threaded database access.

Some FORTRAN compilers treat lines with a 'D' or 'd' in column 1 as conditional lines. These lines can either be compiled for debugging or treated as comments. The precompiler will always treat lines with a 'D' or 'd' in column 1 as comments.

Some API parameters require addresses rather than values in the call variables. The database manager provides the GET ADDRESS, DEREFERENCE ADDRESS, and COPY MEMORY APIs, which simplify your ability to provide these parameters.

The following items affect the precompiling process:
- The precompiler allows only digits, blanks, and tab characters within columns 1-5 on continuation lines.
- Hollerith constants are not supported in `.sqf` source files.

**Related concepts:**
- "Embedded SQL statements in FORTRAN applications" on page 8
- "Null or truncation indicator variables in FORTRAN embedded SQL applications" on page 128
- "Example: SQL declare section template for FORTRAN embedded SQL applications" on page 121
- "Comments in embedded SQL applications" on page 136

**Related reference:**
- "Supported SQL data types in FORTRAN embedded SQL applications" on page 63

## Restrictions on using REXX to program embedded SQL applications

Following are the restrictions for embedded SQL in REXX applications:
- Embedded SQL support for REXX stabilized in DB2 UDB Version 5, and no enhancements are planned for the future. For example, REXX cannot handle SQL object identifiers, such as table names, that are longer than 18 bytes. To use features introduced to DB2 database systems after Version 5, such as table names from 19 to 128 bytes long, you must write your applications in a language other than REXX.
- Compound SQL is not supported in REXX/SQL.
- REXX does not support static SQL.
- REXX applications are not supported under Japanese or Traditional Chinese EUC environments.

**Related concepts:**
- "API Syntax for REXX" on page 157
- "Embedded SQL statements in REXX applications" on page 9

**Related tasks:**
- "Considerations while programming REXX embedded SQL applications" on page 131

**Related reference:**
- "Supported SQL data types in REXX embedded SQL applications" on page 64

- “REXX samples” on page 377

### Recommendations for developing embedded SQL applications with XML and XQuery

The following recommendations and restrictions apply to using XML and XQuery within embedded SQL applications.

- Applications must access all XML data in the serialized string format.
  - You must represent all data, including numeric and date time data, in its serialized string format.
- Externalized XML data is limited to 2 GB.
- All cursors containing XML data are non-blocking (each fetch operation produces a database server request).
- Whenever character host variables contain serialized XML data, the application code page is assumed to be used as the encoding of the data and must match any internal encoding that exists in the data.
- You must specify a LOB data type as the base type for an XML host variable.
- The following apply to static SQL:
  - Character and binary host variables cannot be used to retrieve XML values from a `SELECT INTO` operation.
  - Where an XML data type is expected for input, the use of CHAR, VARCHAR, CLOB, and BLOB host variables will be subject to an `XMLPARSE` operation with default whitespace handling characteristics (`'STRIP WHITESPACE'`). Any other non-XML host variable type will be rejected.
  - There is no support for static XQuery expressions; attempts to precompile an XQuery expression will fail with an error. You can only execute XQuery expressions through the XMLQUERY function.
- An XQuery expression can be dynamically executed by pre-pending the expression with the string ″XQUERY″.

**Related concepts:**

- “Errors and warnings from precompilation of embedded SQL applications” on page 189
- “Example: Referencing XML host variables in embedded SQL applications” on page 76

**Related tasks:**

- “Declaring XML host variables in embedded SQL applications” on page 71
- “Executing XQuery expressions in embedded SQL applications” on page 151

## Concurrent transactions and multi-threaded database access in embedded SQL applications

### Concurrent transactions and multi-threaded database access in embedded SQL applications

One feature of some operating systems is the ability to run several threads of execution within a single process. The multiple threads allow an application to handle asynchronous events, and makes it easier to create event-driven applications, without resorting to polling schemes. The information that follows describes how the DB2 database manager works with multiple threads, and lists some design guidelines that you should keep in mind.

If you are not familiar with terms relating to the development of multi-threaded applications (such as critical section and semaphore), consult the programming documentation for your operating system.

A DB2 embedded SQL application can execute SQL statements from multiple threads using *contexts*. A context is the environment from which an application runs all SQL statements and API calls. All connections, units of work, and other database resources are associated with a specific context. Each context is associated with one or more threads within an application. Developing multi-threaded embedded SQL applications with thread-safe code is only supported in C and C++. It is possible to write your own precompiler, that along with features supplied by the language allows concurrent multithread database access.

For each executable SQL statement in a context, the first run-time services call always tries to obtain a latch. If it is successful, it continues processing. If not (because an SQL statement in another thread of the same context already has the latch), the call is blocked on a signaling semaphore until that semaphore is posted, at which point the call gets the latch and continues processing. The latch is held until the SQL statement has completed processing, at which time it is released by the last run-time services call that was generated for that particular SQL statement.

The net result is that each SQL statement within a context is executed as an atomic unit, even though other threads may also be trying to execute SQL statements at the same time. This action ensures that internal data structures are not altered by different threads at the same time. APIs also use the latch used by run-time services; therefore, APIs have the same restrictions as run-time services routines within each context.

Contexts may be exchanged between threads in a process, but not exchanged between processes. One use of multiple contexts is to provide support for concurrent transactions.

In the default implementation of threaded applications against a DB2 database, serialization of access to the database is enforced by the database APIs. If one thread performs a database call, calls made by other threads will be blocked until the first call completes, even if the subsequent calls access database objects that are unrelated to the first call. In addition, all threads within a process share a commit scope. True concurrent access to a database can only be achieved through separate processes, or by using the APIs that are described in this topic.

DB2 database systems provide APIs that can be used to allocate and manipulate separate environments (contexts) for the use of database APIs and embedded SQL. Each context is a separate entity, and any connection or attachment using one context is independent of all other contexts (and thus all other connections or attachments within a process). In order for work to be done on a context, it must first be associated with a thread. A thread must always have a context when making database API calls or when using embedded SQL.

All DB2 database system applications are multithreaded by default, and are capable of using multiple contexts. You can use the following DB2 APIs to use multiple contexts. Specifically, your application can create a context for a thread, attach to or detach from a separate context for each thread, and pass contexts between threads. If your application does not call *any* of these APIs, DB2 will automatically manage the multiple contexts for your application:

- `sqleAttachToCtx` - Attach to context
- `sqleBeginCtx` - Create and attach to an application context
- `sqleDetachFromCtx` - Detach from context
- `sqleEndCtx` - Detach and destory application context
- `sqleGetCurrentCtx` - Get current context
- `sqleInterruptCtx` - Interrupt context

These APIs have no effect (that is, they are no-ops) on platforms that do not support application threading.

Contexts need not be associated with a given thread for the duration of a connection or attachment. One thread can attach to a context, connect to a database, detach from the context, and then a second thread can attach to the context and continue doing work using the already existing database connection. Contexts can be passed around among threads in a process, but not among processes.

Even if the new APIs are used, the following APIs continue to be serialized:

- `sqlabndx` - Bind
- `sqlaprep` - Precompile Program
- `sqluexpr` - Export
- `db2Import` and `sqluimpr` - Import

**Notes:**

1. The DB2 CLI automatically uses multiple contexts to achieve thread-safe, concurrent database access on platforms that support multi-threading. While not recommended by DB2, users can explicitly disable this feature if required.
2. By default, AIX does not permit 32-bit applications to attach to more than 11 shared memory segments per process, of which a maximum of 10 can be used for DB2 connections.

   When this limit is reached, DB2 returns SQLCODE -1224 on an SQL CONNECT. DB2 Connect™ also has the 10-connection limitation if local users are running two-phase commit over SNA, or two-phase commit with a TP Monitor (SNA or TCP/IP).

   The AIX environment variable `EXTSHM` can be used to increase the maximum number of shared memory segments to which a process can attach.

   To use `EXTSHM` with DB2, do the following:

   In client sessions:

   ```
   export EXTSHM=ON
   ```

   When starting the DB2 server:

   ```
   export EXTSHM=ON
   db2set DB2ENVLIST=EXTSHM
   db2start
   ```

   On DPF, also add the following lines to your userprofile or usercshrc files:

   ```
   EXTSHM=ON
   export EXTSHM
   ```

   An alternative is to move the local database or DB2 Connect into another machine and to access it remotely, or to access the local database or the DB2 Connect database with TCP/IP loop-back by cataloging it as a remote node that has the TCP/IP address of the local machine.

**Related concepts:**

- “Concurrent transactions” in *Developing SQL and External Routines*
- “Recommendations for using multiple threads” on page 30
- “Executing SQL statements in embedded SQL applications” on page 136

**Related reference:**

- “Precompiler customization APIs” in *Administrative API Reference*
- “sqleAttachToCtx API - Attach to context” in *Administrative API Reference*
- “sqleBeginCtx API - Create and attach to an application context” in *Administrative API Reference*
- “sqleDetachFromCtx API - Detach from context” in *Administrative API Reference*
- “sqleEndCtx API - Detach from and free the memory associated with an application context” in *Administrative API Reference*
- “sqleGetCurrentCtx API - Get current context” in *Administrative API Reference*
- “sqleInterruptCtx API - Interrupt context” in *Administrative API Reference*
- “sqleSetTypeCtx API - Set application context type” in *Administrative API Reference*
- “Administrative APIs and application migration” in *Administrative API Reference*
- “Changed APIs and data structures” in *Administrative API Reference*

**Related samples:**

- “dbthrds.sqc -- How to use multiple context APIs on UNIX (C)”
- “dbthrds.sqC -- How to use multiple context APIs on UNIX (C++)”

## Recommendations for using multiple threads

Follow these guidelines when accessing a database from multiple thread applications:

**Serialize alteration of data structures.**
Applications must ensure that user-defined data structures used by SQL statements and database manager routines are not altered by one thread while an SQL statement or database manager routine is being processed in another thread. For example, do not allow a thread to reallocate an SQLDA while it is being used by an SQL statement in another thread.

**Consider using separate data structures.**
It may be easier to give each thread its own user-defined data structures to avoid having to serialize their usage. This guideline is especially true for the SQLCA, which is used not only by every executable SQL statement, but also by all of the database manager routines. There are three alternatives for avoiding this problem with the SQLCA:

- Use `EXEC SQL INCLUDE SQLCA`, but add `struct sqlca sqlca` at the beginning of any routine that is used by any thread other than the first thread.
- Place `EXEC SQL INCLUDE SQLCA` inside each routine that contains SQL, instead of in the global scope.
- Replace `EXEC SQL INCLUDE SQLCA` with `#include "sqlca.h"`, then add `"struct sqlca sqlca"` at the beginning of any routine that uses SQL.

**Related concepts:**

- “Concurrent transactions and multi-threaded database access in embedded SQL applications” on page 27
- “Troubleshooting multi-threaded embedded SQL applications” on page 31

## Code page and country or region code considerations for multi-threaded UNIX applications

This section is specific to C and C++ embedded SQL applications.

On AIX, the Solaris Operating Environment and HP-UX, the functions that are used for run-time querying of the code page and country or region code to be used for a database connection are now thread safe. But these functions can create some lock contention (and resulting performance degradation) in a multi-threaded application that uses a large number of concurrent database connections.

You can use the *DB2_FORCE_NLS_CACHE* environment variable to eliminate the chance of lock contention in multi-threaded applications. When *DB2_FORCE_NLS_CACHE* is set to `TRUE`, the code page and country or region code information is saved the first time a thread accesses it. From that point on, the cached information will be used for any other thread that requests this information. By saving this information, lock contention is eliminated, and in certain situations a performance benefit will be realized.

You should not set DB2_FORCE_NLS_CACHE to TRUE if the application changes locale settings between connections. If this situation occurs, the original locale information will be returned even after the locale settings have been changed. In general, multi-threaded applications will not change locale settings, which, ensures that the application remains thread safe.

**Related concepts:**
- “DB2 registry and environment variables” in *Performance Guide*
- “Connecting to DB2 databases in embedded SQL applications” on page 52
- “Disconnecting from embedded SQL applications” on page 178
- “Performance of embedded SQL applications” on page 22
- “Embedded SQL statements in C and C++ applications” on page 5

## Troubleshooting multi-threaded embedded SQL applications

The sections that follow describe problems that can occur with multi-threaded embedded SQL applications and how to avoid them.

An application that uses multiple threads is more complex than a single-threaded application. This extra complexity can potentially lead to some unexpected problems. When writing a multi-threaded application, exercise caution with the following:

**Database dependencies between two or more contexts.**
Each context in an application has its own set of database resources, including locks on database objects. This characteristic makes it possible for two contexts, if they are accessing the same database object, to deadlock. The database manager may detect the deadlock and one of the contexts will receive SQLCODE -911 and its unit of work will be rolled back.

**Application dependencies between two or more contexts.**

Be careful with any programming techniques that establish inter-context dependencies. Latches, semaphores, and critical sections are examples of programming techniques that can establish such dependencies. If an application has two contexts that have both application and database dependencies between the contexts, it is possible for the application to become deadlocked. If some of the dependencies are outside of the database manager, the deadlock is not detected, thus the application gets suspended or hung.

**Deadlock prevention for multiple contexts.**

Because the database manager cannot detect deadlocks between threads, design and code your application in a way that will prevent (or at least avoid) deadlocks.

As an example of a deadlock that the database manager would not detect, consider an application that has two contexts, both of which access a common data structure. To avoid problems where both contexts change the data structure simultaneously, the data structure is protected by a semaphore. The contexts look like this:

```
context 1
SELECT * FROM TAB1 FOR UPDATE....
UPDATE TAB1 SET....
get semaphore
access data structure
release semaphore
COMMIT

context 2
get semaphore
access data structure
SELECT * FROM TAB1...
release semaphore
COMMIT
```

Suppose the first context successfully executes the SELECT and the UPDATE statements, while the second context gets the semaphore and accesses the data structure. The first context now tries to get the semaphore, but it cannot because the second context is holding the semaphore. The second context now attempts to read a row from table TAB1, but it stops on a database lock held by the first context. The application is now in a state where context 1 cannot finish before context 2 is done and context 2 is waiting for context 1 to finish. The application is deadlocked, but because the database manager does not know about the semaphore dependency neither context will be rolled back. The unresolved dependency leaves the application suspended.

You can avoid the deadlock that would occur for the previous example in several ways.

- Release all locks held before obtaining the semaphore.

  Change the code for context 1 to perform a commit before it gets the semaphore.
- Do not code SQL statements inside a section protected by semaphores.

  Change the code for context 2 to release the semaphore before doing the SELECT.
- Code all SQL statements within semaphores.

Change the code for context 1 to obtain the semaphore before running the SELECT statement. While this technique will work, it is not highly recommended because the semaphores will serialize access to the database manager, which potentially negates the benefits of using multiple threads.

- Set the *locktimeout* database configuration parameter to a value other than -1.

  While a value other than -1 will not prevent the deadlock, it will allow execution to resume. Context 2 is eventually rolled back because it is unable to obtain the requested lock. When handling the roll back error, context 2 should release the semaphore. Once the semaphore has been released, context 1 can continue and context 2 is free to retry its work.

The techniques for avoiding deadlocks are described in terms of the example, but you can apply them to all multi-threaded applications. In general, treat the database manager as you would treat any protected resource and you should not run into problems with multi-threaded applications.

**Related concepts:**

- "Error information in the SQLCODE, SQLSTATE, and SQLWARN fields" on page 173
- "Concurrent transactions and multi-threaded database access in embedded SQL applications" on page 27

**Related tasks:**

- "Including SQLSTATE and SQLCODE host variables in embedded SQL applications" on page 75

# Chapter 3. Programming embedded SQL applications

## Programming embedded SQL applications

Programming embedded SQL applications involves of all of the steps required to assemble an application in a chosen embedded SQL programming language. Once you determine that embedded SQL is the appropriate API to meet your programming needs, and after you design your embedded SQL application, you will be ready to program an embedded SQL application.

Prerequisites:

- Choose whether to use static or dynamic SQL statements
- Design of an embedded SQL application

Programming embedded SQL applications consists of the following sub-tasks:

- Including the required header files
- Choosing a supported embedded SQL programming language
- Declaring host variables for representing values to be included in SQL statements
- Connecting to a data source
- Executing SQL statements
- Handling SQL errors and warnings related to SQL statement execution
- Disconnecting from the data source

Once you have a complete embedded SQL application you'll be ready to compile and run your application: Building embedded SQL applications.

**Related concepts:**

- "Include files and definitions required for embedded SQL applications" on page 42
- "Host Variables in embedded SQL applications" on page 66
- "Executing SQL statements in embedded SQL applications" on page 136
- "Errors and warnings from precompilation of embedded SQL applications" on page 189

**Related tasks:**

- "Declaring host variables in embedded SQL applications" on page 68
- "Referencing host variables in embedded SQL applications" on page 76

## Embedded SQL source files

When you develop source code that includes embedded SQL, you need to follow specific file naming conventions for each of the supported host languages.

**Input and output files for C and C++:**

By default, the source application can have the following extensions:

| | |
|---|---|
| **.sqc** | For C files on all supported operating systems |
| **.sqC** | For C++ files on UNIX and Linux operating systems |
| **.sqx** | For C++ files on Windows operating systems |

By default, the corresponding precompiler output files have the following extensions:

| | |
|---|---|
| **.c** | For C files on all supported operating systems |
| **.C** | For C++ files on UNIX and Linux operating systems |
| **.cxx** | For C++ files on Windows operating systems |

You can use the `OUTPUT` precompile option to override the name and path of the output modified source file. If you use the `TARGET C` or `TARGET CPLUSPLUS` precompile option, the input file does not need a particular extension.

**Input and output files for COBOL:**

By default, the source application has an extension of:

| | |
|---|---|
| **.sqb** | For COBOL files on all operating systems |

However, if you use the TARGET precompile option (TARGET ANSI_COBOL, TARGET IBMCOB or TARGET MFCOB), the input file can have any extension you prefer.

By default, the corresponding precompiler output files have the following extensions:

| | |
|---|---|
| **.cbl** | For COBOL files on all operating systems |

However, you can use the OUTPUT precompile option to specify a new name and path for the output modified source file.

**Input and output files for FORTRAN:**

By default, the source application has an extension of:

| | |
|---|---|
| **.sqf** | For FORTRAN files on all operating systems |

However, if you use the TARGET precompile option with the FORTRAN option the input file can have any extension you prefer.

By default, the corresponding precompiler output files have the following extensions:

| | |
|---|---|
| **.f** | For FORTRAN files on UNIX and Linux operating systems |
| **.for** | For FORTRAN files on Windows operating systems |

However, you can use the OUTPUT precompile option to specify a new name and path for the output modified source file.

**Related concepts:**
- "Building embedded SQL applications using the sample build script" on page 200
- "Building embedded SQL applications" on page 180
- "Building embedded SQL applications from the command line" on page 241

## Embedded SQL application template in C

This is a simple embedded SQL application that is provided for you to use to test your embedded SQL development environment and to help you learn about the basic structure of embedded SQL applications.

Embedded SQL applications require the following structure:
- including the required header files
- host variable declarations for values to be included in SQL statements
- a database connection
- the execution of SQL statements
- the handling of SQL errors and warnings related to SQL statement execution
- dropping the database connection

The following source code demostrates the basic structure required for embedded SQL applications written in C.

```
 #include <stdio.h>                                                    1
 #include <stdlib.h>
 #include <string.h>
 #include <sqlenv.h>
 #include <sqlutil.h>

EXEC SQL BEGIN DECLARE SECTION;                                        2
  short id;
  char name[10];
  short dept;
  double salary;
  char hostVarStmtDyn[50];
EXEC SQL END DECLARE SECTION;

int main()
 {
   int rc = 0;                                                         3
   EXEC SQL INCLUDE SQLCA;                                             4

   /* connect to the database */
   printf("\n Connecting to database...");
   EXEC SQL CONNECT TO "sample";                                       5
   if (SQLCODE < 0)                                                    6
   {
      printf("\nConnect Error:  SQLCODE = %ld \n", SQLCODE);
      goto connect_reset;
   }
   else
   {
      printf("\n Connected to database.\n");
   }

   /* execute an SQL statement (a query) using static SQL; copy the single row
      of result values into host variables*/
   EXEC SQL SELECT id, name, dept, salary                              7
           INTO :id, :name, :dept, :salary
           FROM staff WHERE id = 310;
   if (SQLCODE < 0)                                                    6
   {
      printf("Select Error:  SQLCODE = %ld \n", SQLCODE);
   }
   else
   {
      /* print the host variable values to standard output */
      printf("\n Executing a static SQL query statement, searching for
         \n the id value equal to 310\n");
      printf("\n ID   Name        DEPT       Salary\n");
      printf(" %3d %7s %7d %16.2f\n\n", id, name, dept, salary);
   }

   strcpy(hostVarStmtDyn, "UPDATE staff
                          SET salary = salary + 1000
                          WHERE dept = ?");
   /* execute an SQL statement (an operation) using a host variable
      and DYNAMIC SQL*/
   EXEC SQL PREPARE StmtDyn FROM :hostVarStmtDyn;
   if (SQLCODE < 0)                                                    6
   {
      printf("Prepare Error:  SQLCODE = %ld \n", SQLCODE);
   }
   else
   {
      EXEC SQL EXECUTE StmtDyn USING :dept;                            8
   }
   if (SQLCODE < 0)                                                    6
   {
      printf("Execute Error:  SQLCODE = %ld \n", SQLCODE);
   }
```

*Figure 1. Sample program: template.sqc (Part 1 of 2)*

```
/* Read the updated row using STATIC SQL and CURSOR */
EXEC SQL DECLARE posCur1 CURSOR FOR
   SELECT id, name, dept, salary
   FROM staff WHERE id = 310;
if (SQLCODE < 0)                                                 6
{
   printf("Declare Error:  SQLCODE = %ld \n", SQLCODE);
}
EXEC SQL OPEN posCur1;
EXEC SQL FETCH posCur1 INTO :id, :name, :dept, :salary ;         9
if (SQLCODE < 0)                                                 6
{
   printf("Fetch Error:  SQLCODE = %ld \n", SQLCODE);
}
else
{
   printf(" Executing an dynamic SQL statement, updating the
            \n salary value for the id equal to 310\n");
   printf("\n ID   Name        DEPT       Salary\n");
   printf(" %3d %7s %7d %16.2f\n\n", id, name, dept, salary);
}

EXEC SQL CLOSE posCur1;

/* Commit the transaction */
printf("\n  Commit the transaction.\n");
EXEC SQL COMMIT;                                                 10
if (SQLCODE < 0)                                                 6
{
   printf("Error:  SQLCODE = %ld \n", SQLCODE);
}

/* Disconnect from the database */
connect_reset :
   EXEC SQL CONNECT RESET;                                       11
   if (SQLCODE < 0)                                              6
   {
      printf("Connection Error:  SQLCODE = %ld \n", SQLCODE);
   }
return 0;
} /* end main */
```

*Figure 1. Sample program: template.sqc (Part 2 of 2)*

Notes to Figure 1 on page 40:

| Note | Description |
|---|---|
| 1 | Include files: This directive includes a file into your source application. |
| 2 | Declaration section: Declaration of host variables that will be used to hold values referenced in the SQL statements of the C application. |
| 3 | Local variable declaration: This block declares the local variables to be used in the application. These are not host variables. |
| 4 | Including the SQLCA structure: The SQLCA structure is updated after the execution of each SQL statement. This template application uses certain SQLCA fields for error handling. |
| 5 | Connection to a database: The initial step in working with the database is to establish a connection to the database. Here, a connection is made by executing the `CONNECT` SQL statement. |
| 6 | Error handling: Checks to see if an error occurred. |
| 7 | Executing a query: The execution of this SQL statement assigns data returned from a table to host variables. The C code below the SQL statement execution prints the values in the host variables to standard output. |
| 8 | Executing an operation: The execution of this SQL statement updates a set of rows in a table identified by their department number. Preparation (performed three lines above) is a step in which host variable values, such as the one referenced in this statement, are bound to the SQL statement to be executed. |

| Note | Description |
|---|---|
| 9 | Executing an operation: In this line and the previous line, this application uses cursors in static SQL to select information in a table and print the data. After the cursor is declared and opened, the data is fetched, and finally the cursor is closed. |
| 10 | Commit the transaction: The COMMIT statement finalizes the database changes that were made within a unit of work. |
| 11 | And finally, the database connection must be dropped. |

**Related concepts:**
- "Error message retrieval in embedded SQL applications" on page 174
- "Executing SQL statements in embedded SQL applications" on page 136
- "Include files and definitions required for embedded SQL applications" on page 42

**Related tasks:**
- "Declaring host variables in embedded SQL applications" on page 68
- "Setting up the embedded SQL development environment" on page 11

**Related reference:**
- "SQLCA (SQL communications area)" in *SQL Reference, Volume 1*
- "ROLLBACK statement" in *SQL Reference, Volume 2*
- "COMMIT statement" in *SQL Reference, Volume 2*

## Include files and definitions required for embedded SQL applications

### Include files and definitions required for embedded SQL applications

Include files are needed to provide functions and types used within the library. They must be included before the program can make use of the library functions. By default, these files will be installed in the $HOME/sqllib/include folder. Each host language has its own methods for including files, as well as using different file extensions. Depending on the language specified certain precautions such as specifying file paths must be taken.

**Related reference:**
- "Include files for C and C++ embedded SQL applications" on page 42
- "Include files for COBOL embedded SQL applications" on page 45
- "Include files for FORTRAN embedded SQL applications" on page 47

### Include files for C and C++ embedded SQL applications

The host-language-specific include files (header files) for C and C++ have the file extension `.h`. There are two methods for including files: the `EXEC SQL INCLUDE` statement and the #include macro. The precompiler will ignore the #include, and only process files included with the `EXEC SQL INCLUDE` statement. To locate files included using `EXEC SQL INCLUDE`, the DB2 C precompiler searches the current directory first, then the directories specified by the `DB2INCLUDE` environment variable. Consider the following examples:

- `EXEC SQL INCLUDE payroll;`

  If the file specified in the `INCLUDE` statement is not enclosed in quotation marks, as above, the C precompiler searches for `payroll.sqc`, then `payroll.h`, in each directory in which it looks. On UNIX and Linux operating systems, the C++ precompiler searches for `payroll.sqC`, then `payroll.sqx`, then `payroll.hpp`, then `payroll.h` in each directory it looks. On Windows-32 bit operating systems, the C++ precompiler searches for `payroll.sqx`, then `payroll.hpp`, then `payroll.h` in each directory it looks.
- `EXEC SQL INCLUDE 'pay/payroll.h';`

  If the file name is enclosed in quotation marks, as above, no extension is added to the name.

  If the file name in quotation marks does not contain an absolute path, then the contents of `DB2INCLUDE` are used to search for the file, prepended to whatever path is specified in the `INCLUDE` file name. For example, on UNIX and Linux operating systems, if `DB2INCLUDE` is set to '`/disk2:myfiles/c`', the C or C++ precompiler searches for '`./pay/payroll.h`', then '`/disk2/pay/payroll.h`', and finally '`./myfiles/c/pay/payroll.h`'. The path where the file is actually found is displayed in the precompiler messages. On Windows operating systems, substitute back slashes (\) for the forward slashes in the above example.

**Note:** The setting of `DB2INCLUDE` is cached by the command line processor. To change the setting of `DB2INCLUDE` after any CLP commands have been issued, enter the `TERMINATE` command, then reconnect to the database and precompile as usual.

To help relate compiler errors back to the original source, the precompiler generates #line macros in the output file. This allows the compiler to report errors using the file name and line number of the source or included source file, rather than the line number in the precompiled output source file.

However, if you specify the `PREPROCESSOR` option, all the #line macros generated by the precompiler reference the preprocessed file from the external C preprocessor. Some debuggers and other tools that relate source code to object code do not always work well with the #line macro. If the tool you want to use behaves unexpectedly, use the `NOLINEMACRO` option (used with DB2 PREP) when precompiling. This option prevents the #line macros from being generated.

The include files that are intended to be used in your applications are described below.

**SQLADEF (sqladef.h)**
: This file contains function prototypes used by precompiled C and C++ applications.

**SQLCA (sqlca.h)**
: This file defines the SQL Communication Area (SQLCA) structure. The SQLCA contains variables that are used by the database manager to provide an application with error information about the execution of SQL statements and API calls.

**SQLCODES (sqlcodes.h)**
: This file defines constants for the SQLCODE field of the SQLCA structure.

**SQLDA (sqlda.h)**
: This file defines the SQL Descriptor Area (SQLDA) structure. The SQLDA is used to pass data between an application and the database manager.

**SQLEXT (sqlext.h)**
: This file contains the function prototypes and constants of those ODBC Level 1 and Level 2 APIs that are not part of the X/Open Call Level Interface specification and is therefore used with the permission of Microsoft® Corporation.

**SQLE819A (sqle819a.h)**
: If the code page of the database is 819 (ISO Latin-1), this sequence sorts character strings that are not FOR BIT DATA according to the host CCSID 500 (EBCDIC International) binary collation. This file is used by the CREATE DATABASE API.

**SQLE819B (sqle819b.h)**
: If the code page of the database is 819 (ISO Latin-1), this sequence sorts character strings that are not `FOR BIT DATA` according to the host CCSID 037 (EBCDIC US English) binary collation. This file is used by the CREATE DATABASE API.

**SQLE850A (sqle850a.h)**
: If the code page of the database is 850 (ASCII Latin-1), this sequence sorts character strings that are not `FOR BIT DATA` according to the host CCSID 500 (EBCDIC International) binary collation. This file is used by the CREATE DATABASE API.

**SQLE850B (sqle850b.h)**
: If the code page of the database is 850 (ASCII Latin-1), this sequence sorts character strings that are not `FOR BIT DATA` according to the host CCSID 037 (EBCDIC US English) binary collation. This file is used by the CREATE DATABASE API.

**SQLE932A (sqle932a.h)**
: If the code page of the database is 932 (ASCII Japanese), this sequence sorts character strings that are not `FOR BIT DATA` according to the host CCSID 5035 (EBCDIC Japanese) binary collation. This file is used by the CREATE DATABASE API.

**SQLE932B (sqle932b.h)**
: If the code page of the database is 932 (ASCII Japanese), this sequence sorts character strings that are not `FOR BIT DATA` according to the host CCSID 5026 (EBCDIC Japanese) binary collation. This file is used by the CREATE DATABASE API.

**SQLJACB (sqljacb.h)**
: This file defines constants, structures, and control blocks for the DB2 Connect interface.

**SQLSTATE (sqlstate.h)**
: This file defines constants for the SQLSTATE field of the SQLCA structure.

**SQLSYSTM (sqlsystm.h)**
: This file contains the platform-specific definitions used by the database manager APIs and data structures.

**SQLUDF (sqludf.h)**
: This file defines constants and interface structures for writing user-defined functions (UDFs).

**SQLUV (sqluv.h)**
: This file defines structures, constants, and prototypes for the asynchronous Read Log API, and APIs used by the table load and unload vendors.

**Related reference:**

- "Include files for DB2 API applications" in *Administrative API Reference*
- "INCLUDE statement" in *SQL Reference, Volume 2*
- "C samples" on page 369

## Include files for COBOL embedded SQL applications

The host-language-specific include files for COBOL have the file extension `.cbl`. If you use the ″System/390® host data type support″ feature of the IBM® COBOL compiler, the DB2 include files for your applications are in the following directory:

```
$HOME/sqllib/include/cobol_i
```

If you build the DB2 sample programs with the supplied script files, you must change the include file path specified in the script files to the `cobol_i` directory and not the `cobol_a` directory.

If you do **not** use the ″System/390 host data type support″ feature of the IBM COBOL compiler, or you use an earlier version of this compiler, the DB2 include files for your applications are in the following directory:

```
$HOME/sqllib/include/cobol_a
```

To locate INCLUDE files, the DB2 COBOL precompiler searches the current directory first, then the directories specified by the DB2INCLUDE environment variable. Consider the following examples:

- `EXEC SQL INCLUDE payroll END-EXEC.`

  If the file specified in the INCLUDE statement is not enclosed in quotation marks, as above, the precompiler searches for `payroll.sqb`, then `payroll.cpy`, then `payroll.cbl`, in each directory in which it looks.
- `EXEC SQL INCLUDE 'pay/payroll.cbl' END-EXEC.`

  If the file name is enclosed in quotation marks, as above, no extension is added to the name.

  If the file name in quotation marks does not contain an absolute path, the contents of DB2INCLUDE are used to search for the file, prepended to whatever path is specified in the INCLUDE file name. For example, with DB2 database systems for AIX, if DB2INCLUDE is set to '`/disk2:myfiles/cobol`', the precompiler searches for '`./pay/payroll.cbl`', then '`/disk2/pay/payroll.cbl`', and finally '`./myfiles/cobol/pay/payroll.cbl`'. The path where the file is actually found is displayed in the precompiler messages. On Windows platforms, substitute back slashes (\) for the forward slashes in the above example.

**Note:** The setting of DB2INCLUDE is cached by the DB2 command line processor. To change the setting of DB2INCLUDE after any CLP commands have been issued, enter the TERMINATE command, then reconnect to the database and precompile as usual.

The include files that are intended to be used in your applications are described below.

**SQLCA (sqlca.cbl)**
This file defines the SQL Communication Area (SQLCA) structure. The SQLCA contains variables that are used by the database manager to provide an application with error information about the execution of SQL statements and API calls.

**SQLCA_92 (sqlca_92.cbl)**
This file contains a FIPS SQL92 Entry Level compliant version of the SQL Communications Area (SQLCA) structure. This file should be included in place of the `sqlca.cbl` file when writing DB2 applications that conform to the FIPS SQL92 Entry Level standard. The `sqlca_92.cbl` file is automatically included by the DB2 precompiler when the LANGLEVEL precompiler option is set to SQL92E.

**SQLCODES (sqlcodes.cbl)**
This file defines constants for the SQLCODE field of the SQLCA structure.

**SQLDA (sqlda.cbl)**
This file defines the SQL Descriptor Area (SQLDA) structure. The SQLDA is used to pass data between an application and the database manager.

**SQLEAU (sqleau.cbl)**
This file contains constant and structure definitions required for the DB2 security audit APIs. If you use these APIs, you need to include this file in your program. This file also contains constant and keyword value definitions for fields in the audit trail record. These definitions can be used by external or vendor audit trail extract programs.

**SQLETSD (sqletsd.cbl)**
This file defines the Table Space Descriptor structure, SQLETSDESC, which is passed to the Create Database API, sqlgcrea.

**SQLE819A (sqle819a.cbl)**
If the code page of the database is 819 (ISO Latin-1), this sequence sorts character strings that are not FOR BIT DATA according to the host CCSID 500 (EBCDIC International) binary collation. This file is used by the CREATE DATABASE API.

**SQLE819B (sqle819b.cbl)**
If the code page of the database is 819 (ISO Latin-1), this sequence sorts character strings that are not FOR BIT DATA according to the host CCSID 037 (EBCDIC US English) binary collation. This file is used by the CREATE DATABASE API.

**SQLE850A (sqle850a.cbl)**
If the code page of the database is 850 (ASCII Latin-1), this sequence sorts character strings that are not FOR BIT DATA according to the host CCSID 500 (EBCDIC International) binary collation. This file is used by the CREATE DATABASE API.

**SQLE850B (sqle850b.cbl)**
If the code page of the database is 850 (ASCII Latin-1), this sequence sorts character strings that are not FOR BIT DATA according to the host CCSID 037 (EBCDIC US English) binary collation. This file is used by the CREATE DATABASE API.

**SQLE932A (sqle932a.cbl)**
If the code page of the database is 932 (ASCII Japanese), this sequence sorts character strings that are not FOR BIT DATA according to the host CCSID 5035 (EBCDIC Japanese) binary collation. This file is used by the CREATE DATABASE API.

**SQLE932B (sqle932b.cbl)**
: If the code page of the database is 932 (ASCII Japanese), this sequence sorts character strings that are not FOR BIT DATA according to the host CCSID 5026 (EBCDIC Japanese) binary collation. This file is used by the CREATE DATABASE API.

**SQL1252A (sql1252a.cbl)**
: If the code page of the database is 1252 (Windows Latin-1), this sequence sorts character strings that are not FOR BIT DATA according to the host CCSID 500 (EBCDIC International) binary collation. This file is used by the CREATE DATABASE API.

**SQL1252B (sql1252b.cbl)**
: If the code page of the database is 1252 (Windows Latin-1), this sequence sorts character strings that are not FOR BIT DATA according to the host CCSID 037 (EBCDIC US English) binary collation. This file is used by the CREATE DATABASE API.

**SQLSTATE (sqlstate.cbl)**
: This file defines constants for the SQLSTATE field of the SQLCA structure.

**SQLUTBCQ (sqlutbcq.cbl)**
: This file defines the Table Space Container Query data structure, SQLB-TBSCONTQRY-DATA, which is used with the table space container query APIs, sqlgstsc, sqlgftcq, and sqlgtcq.

**SQLUTBSQ (sqlutbsq.cbl)**
: This file defines the Table Space Query data structure, SQLB-TBSQRY-DATA, which is used with the table space query APIs, sqlgstsq, sqlgftsq, and sqlgtsq.

**Related reference:**
- "Include files for DB2 API applications" in *Administrative API Reference*
- "COBOL samples" on page 373
- "INCLUDE statement" in *SQL Reference, Volume 2*

## Include files for FORTRAN embedded SQL applications

The host-language-specific include files for FORTRAN have the file extension `.f` on UNIX and Linux operating systems, and `.for` on Windows operating systems. There are two methods for including files: the EXEC SQL INCLUDE statement and the FORTRAN INCLUDE statement. The precompiler will ignore FORTRAN INCLUDE statements, and only process files included with the EXEC SQL statement. To locate the INCLUDE file, the DB2 FORTRAN precompiler searches the current directory first, then the directories specified by the DB2INCLUDE environment variable.

Consider the following examples:
- `EXEC SQL INCLUDE payroll`

  If the file specified in the INCLUDE statement is not enclosed in quotation marks, as above, the precompiler searches for `payroll.sqf`, then `payroll.f` (`payroll.for` on Windows operating systems) in each directory in which it looks.
- `EXEC SQL INCLUDE 'pay/payroll.f'`

If the file name is enclosed in quotation marks, as above, no extension is added to the name. (For Windows operating systems, the file would be specified as `'pay\payroll.for'`.)

If the file name in quotation marks does not contain an absolute path, then the contents of DB2INCLUDE are used to search for the file, prepended to whatever path is specified in the INCLUDE file name. For example, with DB2 for UNIX and Linux operating systems, if DB2INCLUDE is set to '`/disk2:myfiles/fortran`', the precompiler searches for '`./pay/payroll.f`', then '`/disk2/pay/payroll.f`', and finally '`./myfiles/cobol/pay/payroll.f`'. The path where the file is actually found is displayed in the precompiler messages. On Windows operating systems, substitute back slashes (\) for the forward slashes, and substitute `'for'` for the `'f'` extension in the above example.

**Note:** The setting of DB2INCLUDE is cached by the DB2 command line processor. To change the setting of DB2INCLUDE after any CLP commands have been issued, enter the TERMINATE command, then reconnect to the database and precompile as usual.

32-bit FORTRAN header files required for DB2 database applicaiton development, previously found in `$INSTHOME/sqllib/include` are now found in `$INSTHOME/sqllib/include32`.

In Version 8.1, these files were found in the `$INSTDIR/sqllib/include` directory which was a symbolic link to one of the following directories: `$DB2DIR/include` or `$DB2DIR/include64` depending on whether or not it was a 32-bit instance or a 64-bit instance.

In Version 9.1, `$DB2DIR/include` will contain all the include files (32-bit and 64-bit), and `$DB2DIR/include32` will contain 32-bit FORTRAN files only and a README file to indicate that 32-bit include files are the same as the 64-bit ones with the exception of FORTRAN.

The `$DB2DIR/include32` directory will only exist on AIX, Solaris, HP-PA, and HP-IPF.

You can use the following FORTRAN include files in your applications.

**SQLCA (sqlca_cn.f, sqlca_cs.f)**

This file defines the SQL Communication Area (SQLCA) structure. The SQLCA contains variables that are used by the database manager to provide an application with error information about the execution of SQL statements and API calls.

Two SQLCA files are provided for FORTRAN applications. The default, `sqlca_cs.f`, defines the SQLCA structure in an IBM SQL compatible format. The `sqlca_cn.f` file, precompiled with the `SQLCA NONE` option, defines the SQLCA structure for better performance.

**SQLCA_92 (sqlca_92.f)**

This file contains a FIPS SQL92 Entry Level compliant version of the SQL Communications Area (SQLCA) structure. This file should be included in place of either the `sqlca_cn.f` or the `sqlca_cs.f` files when writing DB2 applications that conform to the FIPS SQL92 Entry Level standard. The `sqlca_92.f` file is automatically included by the DB2 precompiler when the LANGLEVEL precompiler option is set to SQL92E.

**SQLCODES (sqlcodes.f)**
: This file defines constants for the SQLCODE field of the SQLCA structure.

**SQLDA (sqldact.f)**
: This file defines the SQL Descriptor Area (SQLDA) structure. The SQLDA is used to pass data between an application and the database manager.

**SQLEAU (sqleau.f)**
: This file contains constant and structure definitions required for the DB2 security audit APIs. If you use these APIs, you need to include this file in your program. This file also contains constant and keyword value definitions for fields in the audit trail record. These definitions can be used by external or vendor audit trail extract programs.

**SQLE819A (sqle819a.f)**
: If the code page of the database is 819 (ISO Latin-1), this sequence sorts character strings that are not FOR BIT DATA according to the host CCSID 500 (EBCDIC International) binary collation. This file is used by the CREATE DATABASE API.

**SQLE819B (sqle819b.f)**
: If the code page of the database is 819 (ISO Latin-1), this sequence sorts character strings that are not FOR BIT DATA according to the host CCSID 037 (EBCDIC US English) binary collation. This file is used by the CREATE DATABASE API.

**SQLE850A (sqle850a.f)**
: If the code page of the database is 850 (ASCII Latin-1), this sequence sorts character strings that are not FOR BIT DATA according to the host CCSID 500 (EBCDIC International) binary collation. This file is used by the CREATE DATABASE API.

**SQLE850B (sqle850b.f)**
: If the code page of the database is 850 (ASCII Latin-1), this sequence sorts character strings that are not FOR BIT DATA according to the host CCSID 037 (EBCDIC US English) binary collation. This file is used by the CREATE DATABASE API.

**SQLE932A (sqle932a.f)**
: If the code page of the database is 932 (ASCII Japanese), this sequence sorts character strings that are not FOR BIT DATA according to the host CCSID 5035 (EBCDIC Japanese) binary collation. This file is used by the CREATE DATABASE API.

**SQLE932B (sqle932b.f)**
: If the code page of the database is 932 (ASCII Japanese), this sequence sorts character strings that are not FOR BIT DATA according to the host CCSID 5026 (EBCDIC Japanese) binary collation. This file is used by the CREATE DATABASE API.

**SQL1252A (sql1252a.f)**
: If the code page of the database is 1252 (Windows Latin-1), this sequence sorts character strings that are not FOR BIT DATA according to the host CCSID 500 (EBCDIC International) binary collation. This file is used by the CREATE DATABASE API.

**SQL1252B (sql1252b.f)**
: If the code page of the database is 1252 (Windows Latin-1), this

sequence sorts character strings that are not FOR BIT DATA according to the host CCSID 037 (EBCDIC US English) binary collation. This file is used by the CREATE DATABASE API.

**SQLSTATE (sqlstate.f)**
This file defines constants for the SQLSTATE field of the SQLCA structure.

**Related concepts:**

- "Example: SQL declare section template for FORTRAN embedded SQL applications" on page 121

**Related reference:**

- "Include files for DB2 API applications" in *Administrative API Reference*
- "INCLUDE statement" in *SQL Reference, Volume 2*

## Declaring the SQLCA for Error Handling

You can declare the SQLCA in your application program so that the database manager can return information to your application. When you preprocess your program, the database manager inserts host language variable declarations in place of the INCLUDE SQLCA statement. The system communicates with your program using the variables for warning flags, error codes, and diagnostic information.

After executing each SQL statement, the system returns a return code in both SQLCODE and SQLSTATE. SQLCODE is an integer value that summarizes the execution of the statement, and SQLSTATE is a character field that provides common error codes across IBM's relational database products. SQLSTATE also conforms to the ISO/ANS SQL92 and FIPS 127-2 standard.

**Note:** FIPS 127-2 refers to *Federal Information Processing Standards Publication 127-2 for Database Language SQL*. ISO/ANS SQL92 refers to *American National Standard Database Language SQL X3.135-1992* and *International Standard ISO/IEC 9075:1992, Database Language SQL*.

Note that if SQLCODE is less than 0, it means an error has occurred and the statement has not been processed. If the SQLCODE is greater than 0, it means a warning has been issued, but the statement is still processed.

For a DB2 application written in C or C++, if the application is made up of multiple source files, only one of the files should include the EXEC SQL INCLUDE SQLCA statement to avoid multiple definitions of the SQLCA. The remaining source files should use the following lines:

```
#include "sqlca.h"
extern struct sqlca sqlca;
```

**Procedure:**

To declare the SQLCA, code the `INCLUDE SQLCA` statement in your program as follows:

- For C or C++ applications use:
  ```
  EXEC SQL INCLUDE SQLCA;
  ```
- For Java™ applications, you do not explicitly use the SQLCA. Instead, use the `SQLException` instance methods to get the SQLSTATE and SQLCODE values.

- For COBOL applications use:
  ```
  EXEC SQL INCLUDE SQLCA END-EXEC.
  ```
- For FORTRAN applications use:
  ```
  EXEC SQL INCLUDE SQLCA
  ```

If your application must be compliant with the ISO/ANS SQL92 or FIPS 127-2 standard, do not use the above statements or the `INCLUDE SQLCA` statement.

**Related concepts:**
- "Error Handling Using the WHENEVER Statement" on page 51
- "SQLSTATE and SQLCODE variables in C and C++ embedded SQL application" on page 82
- "SQLSTATE and SQLCODE Variables in COBOL embedded SQL application" on page 109
- "SQLSTATE and SQLCODE variables in FORTRAN embedded SQL application" on page 122
- "SQLSTATE and SQLCODE Variables in Perl" in *Developing Perl and PHP Applications*

## Error Handling Using the WHENEVER Statement

The WHENEVER statement causes the precompiler to generate source code that directs the application to go to a specified label if either an error, a warning, or no rows are found during execution. The WHENEVER statement affects all subsequent executable SQL statements until another WHENEVER statement alters the situation.

The WHENEVER statement has three basic forms:

```
EXEC SQL WHENEVER SQLERROR   action
EXEC SQL WHENEVER SQLWARNING action
EXEC SQL WHENEVER NOT FOUND  action
```

In the above statements:

**SQLERROR**
: Identifies any condition where SQLCODE < `0`.

**SQLWARNING**
: Identifies any condition where SQLWARN(0) = `W` or SQLCODE > `0` but is not equal to `100`.

**NOT FOUND**
: Identifies any condition where SQLCODE = `100`.

In each case, the *action* can be either of the following:

**CONTINUE**
: Indicates to continue with the next instruction in the application.

**GO TO** *label*
: Indicates to go to the statement immediately following the label specified after GO TO. (GO TO can be two words, or one word, GOTO.)

If the WHENEVER statement is not used, the default action is to continue processing if an error, warning, or exception condition occurs during execution.

The WHENEVER statement must appear before the SQL statements you want to affect. Otherwise, the precompiler does not know that additional error-handling code should be generated for the executable SQL statements. You can have any combination of the three basic forms active at any time. The order in which you declare the three forms is not significant.

To avoid an infinite looping situation, ensure that you undo the WHENEVER handling before any SQL statements are executed inside the handler. You can do this using the WHENEVER SQLERROR CONTINUE statement.

**Related reference:**
- "WHENEVER statement" in *SQL Reference, Volume 2*

## Connecting to DB2 databases in embedded SQL applications

Before working with a database, you need to establish a connection to that database. Embedded SQL provides multiple ways in which to include code for establishing database connections. Depending on the embedded SQL host programming language there might be one or more way of doing this.

Database connections can be established implicitly or explicitly. An implicit connection is a connection where the user ID is presumed to be the current user ID. This type of connection is not recommended for database applications. Explicit database connections, which require that a user ID and password be specified, are strongly recommended.

**Connecting to DB2 databases in C and C++ Embedded SQL applications:**

When working with C and C++ applications, a database connection can be established by executing the following statement.

```
EXEC SQL CONNECT TO sample;
```

If you want to use a specific user id (`herrick`) and password (`mypassword`), use the following statement:

```
EXEC SQL CONNECT TO sample USER herrick USING mypassword;
```

**Connecting to DB2 databases in COBOL Embedded SQL applications:**

When working with COBOL applications, a database connection is established by executing the following statement. This statement creates a connection to the sample database using the default user name.

```
EXEC SQL CONNECT TO sample END-EXEC.
```

If you want to use a specific user id (`herrick`) and password (`mypassword`), use the following statement:

```
EXEC SQL CONNECT TO sample USER herrick USING mypassword END-EXEC.
```

**Connecting to DB2 databases in FORTRAN Embedded SQL applications:**

When working with FORTRAN applications, a database connection is established by executing the following statement. This statement creates a connection to the sample database using the default user name.

```
EXEC SQL CONNECT TO sample
```

If you want to use a specific user id (herrick) and password (mypassword), use the following statement:

```
EXEC SQL CONNECT TO sample USER herrick USING mypassword
```

**Connecting to DB2 databases in REXX Embedded SQL applications:**

When working with REXX applications, a database connection is established by executing the following statement. This statement creates a connection to the sample database using the default user name.

```
CALL SQLEXEC 'CONNECT TO sample'
```

If you want to use a specific user id (herrick) and password (mypassword), use the following statement:

```
CALL SQLEXEC 'CONNECT TO sample USER herrick USING mypassword'
```

**Related concepts:**

- "Executing SQL statements in embedded SQL applications" on page 136

**Related reference:**

- "CONNECT (Type 1) statement" in *SQL Reference, Volume 2*
- "CONNECT (Type 2) statement" in *SQL Reference, Volume 2*
- "EXECUTE statement" in *SQL Reference, Volume 2*

## Data types that map to SQL data types in embedded SQL applications

### Data types that map to SQL data types in embedded SQL applications

To exchange data between an application and database, use the correct data type mappings for the variables used. When the precompiler finds a host variable declaration, it determines the appropriate SQL type value. With each host language there are special mapping rules which must be adhered to, unique only to that specific language.

**Related reference:**

- "Supported SQL data types in REXX embedded SQL applications" on page 64
- "Supported SQL data types in C and C++ embedded SQL applications" on page 53
- "Supported SQL data types in COBOL embedded SQL applications" on page 60
- "Supported SQL data types in FORTRAN embedded SQL applications" on page 63

### Supported SQL data types in C and C++ embedded SQL applications

#### Supported SQL data types in C and C++ embedded SQL applications

Certain predefined C and C++ data types correspond to DB2 database column types. Only these C and C++ data types can be declared as host variables.

The following tables show the C and C++ equivalent of each column type. When the precompiler finds a host variable declaration, it determines the appropriate SQL type value. The database manager uses this value to convert the data exchanged between the application and itself.

*Table 2. SQL Data Types Mapped to C and C++ Declarations*

| **SQL Column Type**[1] | **C and C++ Data Type** | **SQL Column Type Description** |
|---|---|---|
| SMALLINT<br>(500 or 501) | short<br>short int<br>sqlint16 | 16-bit signed integer |
| INTEGER<br>(496 or 497) | long<br>long int<br>sqlint32[2] | 32-bit signed integer |
| BIGINT<br>(492 or 493) | long long<br>long<br>__int64<br>sqlint64[3] | 64-bit signed integer |
| REAL[4]<br>(480 or 481) | float | Single-precision floating point |
| DOUBLE[5]<br>(480 or 481) | double | Double-precision floating point |
| DECIMAL(*p*,*s*)<br>(484 or 485) | No exact equivalent; use double | Packed decimal<br><br>(Consider using the CHAR and DECIMAL functions to manipulate packed decimal fields as character data.) |
| CHAR(1)<br>(452 or 453) | char | Single character |
| CHAR(*n*)<br>(452 or 453) | No exact equivalent; use char[*n+1*] where n is large enough to hold the data<br><br>1<=*n*<=254 | Fixed-length character string |
| VARCHAR(*n*)<br>(448 or 449) | struct tag {<br>    short int;<br>    char[*n*]<br>    }<br><br>1<=*n*<=32 672 | Non null-terminated varying character string with 2-byte string length indicator |
| | Alternatively, use char[*n+1*] where n is large enough to hold the data<br><br>1<=*n*<=32 672 | Null-terminated variable-length character string<br>**Note:** Assigned an SQL type of 460/461. |
| LONG VARCHAR<br>(456 or 457) | struct tag {<br>    short int;<br>    char[*n*]<br>    }<br><br>32 673<=*n*<=32 700 | Non null-terminated varying character string with 2-byte string length indicator |
| CLOB(*n*)<br>(408 or 409) | sql type is<br>    clob(*n*)<br><br>1<=*n*<=2 147 483 647 | Non null-terminated varying character string with 4-byte string length indicator |

Table 2. SQL Data Types Mapped to C and C++ Declarations (continued)

| SQL Column Type[1] | C and C++ Data Type | SQL Column Type Description |
|---|---|---|
| CLOB locator variable[6] (964 or 965) | sql type is clob_locator | Identifies CLOB entities residing on the server |
| CLOB file reference variable[6] (920 or 921) | sql type is clob_file | Descriptor for file containing CLOB data |
| BLOB(*n*) (404 or 405) | sql type is blob(*n*)<br><br>1<=*n*<=2 147 483 647 | Non null-terminated varying binary string with 4-byte string length indicator |
| BLOB locator variable[6] (960 or 961) | sql type is blob_locator | Identifies BLOB entities on the server |
| BLOB file reference variable[6] (916 or 917) | sql type is blob_file | Descriptor for the file containing BLOB data |
| DATE (384 or 385) | Null-terminated character form | Allow at least 11 characters to accommodate the null-terminator |
| | VARCHAR structured form | Allow at least 10 characters |
| TIME (388 or 389) | Null-terminated character form | Allow at least 9 characters to accommodate the null-terminator |
| | VARCHAR structured form | Allow at least 8 characters |
| TIMESTAMP (392 or 393) | Null-terminated character form | Allow at least 27 characters to accommodate the null-terminator |
| | VARCHAR structured form | Allow at least 26 characters |
| XML[7] (988 or 989) | struct {<br>  sqluint32 length;<br>  char data[*n*];<br>}<br><br>1<=*n*<=2 147 483 647<br><br>SQLUDF_CLOB | XML value |

The following data types are only available in the DBCS or EUC environment when precompiled with the WCHARTYPE NOCONVERT option.

Table 3. SQL Data Types Mapped to C and C++ Declarations

| SQL Column Type[1] | C and C++ Data Type | SQL Column Type Description |
|---|---|---|
| GRAPHIC(1) (468 or 469) | sqldbchar | Single double-byte character |
| GRAPHIC(*n*) (468 or 469) | No exact equivalent; use sqldbchar[*n+1*] where n is large enough to hold the data<br><br>1<=*n*<=127 | Fixed-length double-byte character string |

Table 3. SQL Data Types Mapped to C and C++ Declarations (continued)

| SQL Column Type[1] | C and C++ Data Type | SQL Column Type Description |
|---|---|---|
| VARGRAPHIC(*n*) (464 or 465) | struct tag { short int; sqldbchar[*n*] } <br><br> 1<=*n*<=16 336 | Non null-terminated varying double-byte character string with 2-byte string length indicator |
| | Alternatively use sqldbchar[*n+1*] where n is large enough to hold the data <br><br> 1<=*n*<=16 336 | Null-terminated variable-length double-byte character string **Note:** Assigned an SQL type of 400/401. |
| LONG VARGRAPHIC (472 or 473) | struct tag { short int; sqldbchar[*n*] } <br><br> 16 337<=*n*<=16 350 | Non null-terminated varying double-byte character string with 2-byte string length indicator |

The following data types are only available in the DBCS or EUC environment when precompiled with the WCHARTYPE CONVERT option.

Table 4. SQL Data Types Mapped to C and C++ Declarations

| SQL Column Type[1] | C and C++ Data Type | SQL Column Type Description |
|---|---|---|
| GRAPHIC(1) (468 or 469) | wchar_t | • Single wide character (for C-type) <br> • Single double-byte character (for column type) |
| GRAPHIC(*n*) (468 or 469) | No exact equivalent; use wchar_t [*n+1*] where n is large enough to hold the data <br><br> 1<=*n*<=127 | Fixed-length double-byte character string |
| VARGRAPHIC(*n*) (464 or 465) | struct tag { short int; wchar_t [*n*] } <br><br> 1<=*n*<=16 336 | Non null-terminated varying double-byte character string with 2-byte string length indicator |
| | Alternately use char[*n+1*] where n is large enough to hold the data <br><br> 1<=*n*<=16 336 | Null-terminated variable-length double-byte character string **Note:** Assigned an SQL type of 400/401. |
| LONG VARGRAPHIC (472 or 473) | struct tag { short int; wchar_t [*n*] } <br><br> 16 337<=*n*<=16 350 | Non null-terminated varying double-byte character string with 2-byte string length indicator |

The following data types are only available in the DBCS or EUC environment.

*Table 5. SQL Data Types Mapped to C and C++ Declarations*

| **SQL Column Type**[1] | **C and C++ Data Type** | **SQL Column Type Description** |
|---|---|---|
| DBCLOB(*n*) (412 or 413) | sql type is dbclob(*n*)<br><br>1<=*n*<=1 073 741 823 | Non null-terminated varying double-byte character string with 4-byte string length indicator |
| DBCLOB locator variable[6] (968 or 969) | sql type is dbclob_locator | Identifies DBCLOB entities residing on the server |
| DBCLOB file reference variable[6] (924 or 925) | sql type is dbclob_file | Descriptor for file containing DBCLOB data |

**Notes:**

1. The first number under **SQL Column Type** indicates that an indicator variable is not provided, and the second number indicates that an indicator variable is provided. An indicator variable is needed to indicate NULL values, or to hold the length of a truncated string. These are the values that would appear in the SQLTYPE field of the SQLDA for these data types.
2. For platform compatibility, use sqlint32. On 64-bit UNIX and Linux operating systems, ″long″ is a 64 bit integer. On 64-bit Windows operating systems and 32-bit UNIX and Linux operating systems ″long″ is a 32 bit integer.
3. For platform compatibility, use sqlint64. The DB2 database system sqlsystm.h header file will type define sqlint64 as ″__int64″ on the supported Windows operating systems when using the Microsoft compiler, ″long long″ on 32-bit UNIX and Linux operating systems, and ″long″ on 64 bit UNIX and Linux operating systems.
4. FLOAT(*n*) where $0 < n < 25$ is a synonym for REAL. The difference between REAL and DOUBLE in the SQLDA is the length value (4 or 8).
5. The following SQL types are synonyms for DOUBLE:
   - FLOAT
   - FLOAT(*n*) where $24 < n < 54$ is a synonym for DOUBLE
   - DOUBLE PRECISION
6. This is not a column type but a host variable type.
7. The SQL_TYP_XML/SQL_TYP_NXML value is returned by DESCRIBE requests only. It cannot be used directly by the application to bind application resources to XML values.

The following are additional rules for supported C and C++ data types:

- The data type `char` can be declared as `char` or `unsigned char`.
- The database manager processes null-terminated variable-length character string data type char[*n*] (data type 460), as VARCHAR(*m*).
  - If LANGLEVEL is SAA1, the host variable length *m* equals the character string length *n* in char[*n*] or the number of bytes preceding the first null-terminator (`\0`), whichever is smaller.
  - If LANGLEVEL is MIA, the host variable length *m* equals the number of bytes preceding the first null-terminator (`\0`).
- The database manager processes null-terminated, variable-length graphic string data type, `wchar_t[`*n*`]` or `sqldbchar[`*n*`]` (data type 400), as VARGRAPHIC(*m*).
  - If LANGLEVEL is SAA1, the host variable length *m* equals the character string length *n* in `wchar_t[`*n*`]` or `sqldbchar[`*n*`]`, or the number of characters preceding the first graphic null-terminator, whichever is smaller.
  - If LANGLEVEL is MIA, the host variable length *m* equals the number of characters preceding the first graphic null-terminator.
- Unsigned numeric data types are not supported.
- The C and C++ data type `int` is not allowed because its internal representation is machine dependent.

**Related concepts:**

- "Example: SQL declare section template for C and C++ embedded SQL applications" on page 81
- "Host variables in C and C++ embedded SQL applications" on page 78
- "Host variable names in C and C++ embedded SQL applications" on page 79
- "Declare section for host variables in C and C++ embedded SQL applications" on page 80

**Related reference:**

- "C++ samples" in *Samples Topics*
- "C samples" on page 369

## Data types for procedures, functions, and methods in C and C++ embedded SQL applications

The following table lists the supported mappings between SQL data types and C and C++ data types for procedures, UDFs, and methods.

*Table 6. SQL Data Types Mapped to C and C++ Declarations*

| SQL Column Type[1] | C and C++ Data Type | SQL Column Type Description |
|---|---|---|
| SMALLINT (500 or 501) | short | 16-bit signed integer |
| INTEGER (496 or 497) | sqlint32 | 32-bit signed integer |
| BIGINT (492 or 493) | sqlint64 | 64-bit signed integer |
| REAL (480 or 481) | float | Single-precision floating point |
| DOUBLE (480 or 481) | double | Double-precision floating point |
| DECIMAL(*p*,*s*) (484 or 485) | Not supported | To pass a decimal value, define the parameter to be of a data type castable from DECIMAL (for example CHAR or DOUBLE) and explicitly cast the argument to this type. |
| CHAR(*n*) (452 or 453) | char[*n+1*] where n is large enough to hold the data<br><br>1<=*n*<=254 | Fixed-length, null-terminated character string |
| CHAR(*n*) FOR BIT DATA (452 or 453) | char[*n+1*] where n is large enough to hold the data<br><br>1<=*n*<=254 | Fixed-length character string |
| VARCHAR(*n*) (448 or 449) (460 or 461) | char[*n+1*] where n is large enough to hold the data<br><br>1<=*n*<=32 672 | Null-terminated varying length string |
| VARCHAR(*n*) FOR BIT DATA (448 or 449) | struct {<br>sqluint16 length;<br>char[*n*]<br>}<br><br>1<=*n*<=32 672 | Not null-terminated varying length character string |

*Table 6. SQL Data Types Mapped to C and C++ Declarations (continued)*

| SQL Column Type[1] | C and C++ Data Type | SQL Column Type Description |
|---|---|---|
| LONG VARCHAR (456 or 457) | struct { sqluint16 length; char[*n*] } <br> 32 673<=*n*<=32 700 | Not null-terminated varying length character string |
| CLOB(*n*) (408 or 409) | struct { sqluint32 length; char data[n]; } <br> 1<=*n*<=2 147 483 647 | Not null-terminated varying length character string with 4-byte string length indicator |
| BLOB(*n*) (404 or 405) | struct { sqluint32 length; char data[n]; } <br> 1<=*n*<=2 147 483 647 | Not null-terminated varying binary string with 4-byte string length indicator |
| DATE (384 or 385) | char[11] | Null-terminated character form |
| TIME (388 or 389) | char[9] | Null-terminated character form |
| TIMESTAMP (392 or 393) | char[27] | Null-terminated character form |
| XML (988/989) | Not supported | This descriptor type value (988/989) will be defined to be used in the SQLDA for describe, and to indicate XML Data (in its serialized form). Existing character and binary types (including LOBs and LOB file reference types) can also be used to fetch and insert the data (dynamic SQL only) |

**Note:** The following data types are only available in the DBCS or EUC environment when precompiled with the WCHARTYPE NOCONVERT option.

*Table 7. SQL Data Types Mapped to C and C++ Declarations*

| SQL Column Type[1] | C and C++ Data Type | SQL Column Type Description |
|---|---|---|
| GRAPHIC(*n*) (468 or 469) | sqldbchar[*n+1*] where n is large enough to hold the data <br> 1<=*n*<=127 | Fixed-length, null-terminated double-byte character string |
| VARGRAPHIC(*n*) (400 or 401) | sqldbchar[*n+1*] where n is large enough to hold the data <br> 1<=*n*<=16 336 | Not null-terminated, variable-length double-byte character string |
| LONG VARGRAPHIC (472 or 473) | struct { sqluint16 length; sqldbchar[*n*] } <br> 16 337<=*n*<=16 350 | Not null-terminated, variable-length double-byte character string |
| DBCLOB(*n*) (412 or 413) | struct { sqluint32 length; sqldbchar data[n]; } <br> 1<=*n*<=1 073 741 823 | Not null-terminated varying length character string with 4-byte string length indicator |

*Table 7. SQL Data Types Mapped to C and C++ Declarations (continued)*

| SQL Column Type[1] | C and C++ Data Type | SQL Column Type Description |
|---|---|---|

**Notes:**

1. The first number under **SQL Column Type** indicates that an indicator variable is not provided, and the second number indicates that an indicator variable is provided. An indicator variable is needed to indicate NULL values, or to hold the length of a truncated string. These are the values that would appear in the SQLTYPE field of the SQLDA for these data types.

**Related concepts:**

- "Null or truncation indicator variables and indicator tables in C and C++ embedded SQL applications" on page 104
- "Calling stored procedures in C and C++ embedded SQL applications" on page 156

## Supported SQL data types in COBOL embedded SQL applications

Certain predefined COBOL data types correspond to DB2 database column types. Only these COBOL data types can be declared as host variables.

The following table shows the COBOL equivalent of each column type. When the precompiler finds a host variable declaration, it determines the appropriate SQL type value. The database manager uses this value to convert the data exchanged between the application and itself.

Not every possible data description for host variables is recognized. COBOL data items must be consistent with the ones described in the following table. If you use other data items, an error can result.

*Table 8. SQL Data Types Mapped to COBOL Declarations*

| SQL Column Type[1] | COBOL Data Type | SQL Column Type Description |
|---|---|---|
| SMALLINT (500 or 501) | 01 name PIC S9(4) COMP-5. | 16-bit signed integer |
| INTEGER (496 or 497) | 01 name PIC S9(9) COMP-5. | 32-bit signed integer |
| BIGINT (492 or 493) | 01 name PIC S9(18) COMP-5. | 64-bit signed integer |
| DECIMAL(*p*,*s*) (484 or 485) | 01 name PIC S9(*m*)V9(*n*) COMP-3. | Packed decimal |
| REAL[2] (480 or 481) | 01 name USAGE IS COMP-1. | Single-precision floating point |
| DOUBLE[3] (480 or 481) | 01 name USAGE IS COMP-2. | Double-precision floating point |
| CHAR(*n*) (452 or 453) | 01 name PIC X(*n*). | Fixed-length character string |
| VARCHAR(*n*) (448 or 449) | 01 name.<br>49 length PIC S9(4) COMP-5.<br>49 name PIC X(*n*).<br><br>1<=*n*<=32 672 | Variable-length character string |
| LONG VARCHAR (456 or 457) | 01 name.<br>49 length PIC S9(4) COMP-5.<br>49 data PIC X(*n*).<br><br>32 673<=*n*<=32 700 | Long variable-length character string |

Table 8. SQL Data Types Mapped to COBOL Declarations (continued)

| SQL Column Type[1] | COBOL Data Type | SQL Column Type Description |
|---|---|---|
| CLOB(*n*) (408 or 409) | 01 MY-CLOB USAGE IS SQL TYPE IS CLOB(n).<br><br>1<=*n*<=2 147 483 647 | Large object variable-length character string |
| CLOB locator variable[4] (964 or 965) | 01 MY-CLOB-LOCATOR USAGE IS SQL TYPE IS CLOB-LOCATOR. | Identifies CLOB entities residing on the server |
| CLOB file reference variable[4] (920 or 921) | 01 MY-CLOB-FILE USAGE IS SQL TYPE IS CLOB-FILE. | Descriptor for file containing CLOB data |
| BLOB(*n*) (404 or 405) | 01 MY-BLOB USAGE IS SQL TYPE IS BLOB(n).<br><br>1<=*n*<=2 147 483 647 | Large object variable-length binary string |
| BLOB locator variable[4] (960 or 961) | 01 MY-BLOB-LOCATOR USAGE IS SQL TYPE IS BLOB-LOCATOR. | Identifies BLOB entities residing on the server |
| BLOB file reference variable[4] (916 or 917) | 01 MY-BLOB-FILE USAGE IS SQL TYPE IS BLOB-FILE. | Descriptor for file containing BLOB data |
| DATE (384 or 385) | 01 identifier PIC X(10). | 10-byte character string |
| TIME (388 or 389) | 01 identifier PIC X(8). | 8-byte character string |
| TIMESTAMP (392 or 393) | 01 identifier PIC X(26). | 26-byte character string |
| XML[5] (988 or 989) | 01 name USAGE IS SQL TYPE IS XML AS CLOB (size). | XML value |

The following data types are only available in the DBCS environment.

Table 9. SQL Data Types Mapped to COBOL Declarations

| SQL Column Type[1] | COBOL Data Type | SQL Column Type Description |
|---|---|---|
| GRAPHIC(*n*) (468 or 469) | 01 name PIC G(*n*) DISPLAY-1. | Fixed-length double-byte character string |
| VARGRAPHIC(*n*) (464 or 465) | 01 name.<br> 49 length PIC S9(4) COMP-5.<br> 49 name PIC G(*n*) DISPLAY-1.<br><br>1<=*n*<=16 336 | Variable length double-byte character string with 2-byte string length indicator |
| LONG VARGRAPHIC (472 or 473) | 01 name.<br> 49 length PIC S9(4) COMP-5.<br> 49 name PIC G(*n*) DISPLAY-1.<br><br>16 337<=*n*<=16 350 | Variable length double-byte character string with 2-byte string length indicator |
| DBCLOB(*n*) (412 or 413) | 01 MY-DBCLOB USAGE IS SQL TYPE IS DBCLOB(n).<br><br>1<=*n*<=1 073 741 823 | Large object variable-length double-byte character string with 4-byte string length indicator |
| DBCLOB locator variable[4] (968 or 969) | 01 MY-DBCLOB-LOCATOR USAGE IS SQL TYPE IS DBCLOB-LOCATOR. | Identifies DBCLOB entities residing on the server |
| DBCLOB file reference variable[4] (924 or 925) | 01 MY-DBCLOB-FILE USAGE IS SQL TYPE IS DBCLOB-FILE. | Descriptor for file containing DBCLOB data |

*Table 9. SQL Data Types Mapped to COBOL Declarations (continued)*

| SQL Column Type[1] | COBOL Data Type | SQL Column Type Description |
|---|---|---|

**Notes:**

1. The first number under **SQL Column Type** indicates that an indicator variable is not provided, and the second number indicates that an indicator variable is provided. An indicator variable is needed to indicate NULL values, or to hold the length of a truncated string. These are the values that would appear in the SQLTYPE field of the SQLDA for these data types.
2. FLOAT(*n*) where $0 < n < 25$ is a synonym for REAL. The difference between REAL and DOUBLE in the SQLDA is the length value (4 or 8).
3. The following SQL types are synonyms for DOUBLE:
   - FLOAT
   - FLOAT(*n*) where $24 < n < 54$ is a synonym for DOUBLE.
   - DOUBLE PRECISION
4. This is not a column type but a host variable type.
5. The SQL_TYP_XML/SQL_TYP_NXML value is returned by DESCRIBE requests only. It cannot be used directly by the application to bind application resources to XML values.

The following are additional rules for supported COBOL data types:

- PIC S9 and COMP-3/COMP-5 are required where shown.
- You can use level number 77 instead of 01 for all column types except VARCHAR, LONG VARCHAR, VARGRAPHIC, LONG VARGRAPHIC and all LOB variable types.
- Use the following rules when declaring host variables for DECIMAL(p,s) column types. See the following sample:

  ```
  01 identifier PIC S9(m)V9(n) COMP-3
  ```
  - Use V to denote the decimal point.
  - Values for *n* and *m* must be greater than or equal to 1.
  - The value for *n* + *m* cannot exceed 31.
  - The value for *s* equals the value for *n*.
  - The value for *p* equals the value for *n* + *m*.
  - The repetition factors *(n)* and *(m)* are optional. The following examples are all valid:

    ```
    01 identifier PIC S9(3)V COMP-3
    01 identifier PIC SV9(3) COMP-3
    01 identifier PIC S9V COMP-3
    01 identifier PIC SV9 COMP-3
    ```
  - PACKED-DECIMAL can be used instead of COMP-3.
- Arrays are *not* supported by the COBOL precompiler.

**Related concepts:**

- "Declare section for host variables in COBOL embedded SQL applications" on page 108
- "Example: Referencing XML host variables in embedded SQL applications" on page 76
- "Example: SQL declare section template for COBOL embedded SQL applications" on page 108
- "Host variable names in COBOL" on page 107
- "Null-indicator variables and null or truncation indicator variable tables in COBOL embedded SQL applications" on page 119

**Related tasks:**

- "Declaring XML host variables in embedded SQL applications" on page 71
- "Executing XQuery expressions in embedded SQL applications" on page 151

**Related reference:**
- "COBOL samples" on page 373

## Supported SQL data types in FORTRAN embedded SQL applications

Certain predefined FORTRAN data types correspond to DB2 database column types. Only these FORTRAN data types can be declared as host variables.

The following table shows the FORTRAN equivalent of each column type. When the precompiler finds a host variable declaration, it determines the appropriate SQL type value. The database manager uses this value to convert the data exchanged between the application and itself.

*Table 10. SQL Data Types Mapped to FORTRAN Declarations*

| SQL Column Type[1] | FORTRAN Data Type | SQL Column Type Description |
|---|---|---|
| SMALLINT (500 or 501) | INTEGER*2 | 16-bit, signed integer |
| INTEGER (496 or 497) | INTEGER*4 | 32-bit, signed integer |
| REAL[2] (480 or 481) | REAL*4 | Single precision floating point |
| DOUBLE[3] (480 or 481) | REAL*8 | Double precision floating point |
| DECIMAL(*p*,*s*) (484 or 485) | No exact equivalent; use REAL*8 | Packed decimal |
| CHAR(*n*) (452 or 453) | CHARACTER**n* | Fixed-length character string of length *n* where *n* is from 1 to 254 |
| VARCHAR(*n*) (448 or 449) | SQL TYPE IS VARCHAR(*n*) where *n* is from 1 to 32 672 | Variable-length character string |
| LONG VARCHAR (456 or 457) | SQL TYPE IS VARCHAR(*n*) where *n* is from 32 673 to 32 700 | Long variable-length character string |
| CLOB(*n*) (408 or 409) | SQL TYPE IS CLOB (n) where *n* is from 1 to 2 147 483 647 | Large object variable-length character string |
| CLOB locator variable[4] (964 or 965) | SQL TYPE IS CLOB_LOCATOR | Identifies CLOB entities residing on the server |
| CLOB file reference variable[4] (920 or 921) | SQL TYPE IS CLOB_FILE | Descriptor for file containing CLOB data |
| BLOB(*n*) (404 or 405) | SQL TYPE IS BLOB(n) where *n* is from 1 to 2 147 483 647 | Large object variable-length binary string |
| BLOB locator variable[4] (960 or 961) | SQL TYPE IS BLOB_LOCATOR | Identifies BLOB entities on the server |
| BLOB file reference variable[4] (916 or 917) | SQL TYPE IS BLOB_FILE | Descriptor for the file containing BLOB data |
| DATE (384 or 385) | CHARACTER*10 | 10-byte character string |
| TIME (388 or 389) | CHARACTER*8 | 8-byte character string |
| TIMESTAMP (392 or 393) | CHARACTER*26 | 26-byte character string |
| XML (988 or 989) | SQL_TYP_XML | There is no XML support for FORTRAN; applications are able to get the describe type back but will not be able to make use of it. |

*Table 10. SQL Data Types Mapped to FORTRAN Declarations (continued)*

| SQL Column Type[1] | FORTRAN Data Type | SQL Column Type Description |
|---|---|---|

**Notes:**

1. The first number under **SQL Column Type** indicates that an indicator variable is not provided, and the second number indicates that an indicator variable is provided. An indicator variable is needed to indicate NULL values, or to hold the length of a truncated string. These are the values that would appear in the SQLTYPE field of the SQLDA for these data types.
2. FLOAT(*n*) where $0 < n < 25$ is a synonym for REAL. The difference between REAL and DOUBLE in the SQLDA is the length value (4 or 8).
3. The following SQL types are synonyms for DOUBLE:
   - FLOAT
   - FLOAT(*n*) where $24 < n < 54$ is a synonym for DOUBLE.
   - DOUBLE PRECISION
4. This is not a column type but a host variable type.

The following is an additional rule for supported FORTRAN data types:

- You can define dynamic SQL statements longer than 254 characters by using VARCHAR, LONG VARCHAR, OR CLOB host variables.

**Related concepts:**

- "Example: SQL declare section template for FORTRAN embedded SQL applications" on page 121
- "Null or truncation indicator variables in FORTRAN embedded SQL applications" on page 128
- "Host variable names in FORTRAN embedded SQL applications" on page 120
- "Declare section for host variables in FORTRAN embedded SQL applications" on page 121

**Related reference:**

- "Declaration of file reference type host variables in FORTRAN embedded SQL applications" on page 126

## Supported SQL data types in REXX embedded SQL applications

Certain predefined REXX data types correspond to DB2 database column types. Only these REXX data types can be declared as host variables. The following table shows how SQLEXEC and SQLDBS interpret REXX variables in order to convert their contents to DB2 data types.

*Table 11. SQL Column Types Mapped to REXX Declarations*

| SQL Column Type[1] | REXX Data Type | SQL Column Type Description |
|---|---|---|
| SMALLINT (500 or 501) | A number without a decimal point ranging from -32 768 to 32 767 | 16-bit signed integer |
| INTEGER (496 or 497) | A number without a decimal point ranging from -2 147 483 648 to 2 147 483 647 | 32-bit signed integer |
| REAL[2] (480 or 481) | A number in scientific notation ranging from -3.40282346 x $10^{38}$ to 3.40282346 x $10^{38}$ | Single-precision floating point |
| DOUBLE[3] (480 or 481) | A number in scientific notation ranging from -1.79769313 x $10^{308}$ to 1.79769313 x $10^{308}$ | Double-precision floating point |
| DECIMAL(*p*,*s*) (484 or 485) | A number with a decimal point | Packed decimal |

Table 11. SQL Column Types Mapped to REXX Declarations (continued)

| SQL Column Type[1] | REXX Data Type | SQL Column Type Description |
|---|---|---|
| CHAR(*n*) (452 or 453) | A string with a leading and trailing quotation mark (′), which has length *n* after removing the two quotation marks<br><br>A string of length *n* with any non-numeric characters, other than leading and trailing blanks or the E in scientific notation | Fixed-length character string of length *n* where *n* is from 1 to 254 |
| VARCHAR(*n*) (448 or 449) | Equivalent to CHAR(*n*) | Variable-length character string of length *n*, where *n* ranges from 1 to 4000 |
| LONG VARCHAR (456 or 457) | Equivalent to CHAR(*n*) | Variable-length character string of length *n*, where *n* ranges from 1 to 32 700 |
| CLOB(*n*) (408 or 409) | Equivalent to CHAR(*n*) | Large object variable-length character string of length *n*, where *n* ranges from 1 to 2 147 483 647 |
| CLOB locator variable[4] (964 or 965) | DECLARE *:var_name* LANGUAGE TYPE CLOB LOCATOR | Identifies CLOB entities residing on the server |
| CLOB file reference variable[4] (920 or 921) | DECLARE *:var_name* LANGUAGE TYPE CLOB FILE | Descriptor for file containing CLOB data |
| BLOB(*n*) (404 or 405) | A string with a leading and trailing apostrophe, preceded by BIN, containing *n* characters after removing the preceding BIN and the two apostrophes. | Large object variable-length binary string of length *n*, where *n* ranges from 1 to 2 147 483 647 |
| BLOB locator variable[4] (960 or 961) | DECLARE *:var_name* LANGUAGE TYPE BLOB LOCATOR | Identifies BLOB entities on the server |
| BLOB file reference variable[4] (916 or 917) | DECLARE *:var_name* LANGUAGE TYPE BLOB FILE | Descriptor for the file containing BLOB data |
| DATE (384 or 385) | Equivalent to CHAR(*10*) | 10-byte character string |
| TIME (388 or 389) | Equivalent to CHAR(*8*) | 8-byte character string |
| TIMESTAMP (392 or 393) | Equivalent to CHAR(*26*) | 26-byte character string |
| XML (988 or 989) | SQL_TYP_XML | There is no XML support for REXX; applications are able to get the describe type back but will not be able to make use of it. |

The following data types are only available in the DBCS environment.

Table 12. SQL Column Types Mapped to REXX Declarations

| SQL Column Type[1] | REXX Data Type | SQL Column Type Description |
|---|---|---|
| GRAPHIC(*n*) (468 or 469) | A string with a leading and trailing apostrophe preceded by a G or N, containing *n* DBCS characters after removing the preceding character and the two apostrophes | Fixed-length graphic string of length *n*, where *n* is from 1 to 127 |
| VARGRAPHIC(*n*) (464 or 465) | Equivalent to GRAPHIC(*n*) | Variable-length graphic string of length *n*, where *n* ranges from 1 to 2 000 |
| LONG VARGRAPHIC (472 or 473) | Equivalent to GRAPHIC(*n*) | Long variable-length graphic string of length *n*, where *n* ranges from 1 to 16 350 |
| DBCLOB(*n*) (412 or 413) | Equivalent to GRAPHIC(*n*) | Large object variable-length graphic string of length *n*, where *n* ranges from 1 to 1 073 741 823 |
| DBCLOB locator variable[4] (968 or 969) | DECLARE *:var_name* LANGUAGE TYPE DBCLOB LOCATOR | Identifies DBCLOB entities residing on the server |
| DBCLOB file reference variable[4] (924 or 925) | DECLARE *:var_name* LANGUAGE TYPE DBCLOB FILE | Descriptor for file containing DBCLOB data |

*Table 12. SQL Column Types Mapped to REXX Declarations (continued)*

| SQL Column Type[1] | REXX Data Type | SQL Column Type Description |
|---|---|---|

**Notes:**

1. The first number under **Column Type** indicates that an indicator variable is not provided, and the second number indicates that an indicator variable is provided. An indicator variable is needed to indicate NULL values, or to hold the length of a truncated string.
2. FLOAT($n$) where $0 < n < 25$ is a synonym for REAL. The difference between REAL and DOUBLE in the SQLDA is the length value (4 or 8).
3. The following SQL types are synonyms for DOUBLE:
   - FLOAT
   - FLOAT($n$) where $24 < n < 54$ is a synonym for DOUBLE.
   - DOUBLE PRECISION
4. This is not a column type but a host variable type.

**Related concepts:**

- "Null or truncation indicator variables in REXX embedded SQL applications" on page 135
- "Host variable names in REXX embedded SQL applications" on page 128
- "Host variable references in REXX embedded SQL applications" on page 129

**Related reference:**

- "Declaration of file reference type host variables in REXX embedded SQL applications" on page 134
- "REXX samples" on page 377

## Host variables in embedded SQL applications

### Host Variables in embedded SQL applications

*Host variables* are variables referenced by embedded SQL statements. They are used to exchange data values between the database server and the embedded SQL application. Embedded SQL applications can also include host variable declarations for relational SQL queries. Furthermore, a host variable can be used to contain an XQuery expression to be executed. There is, however, no mechanism for passing values to parameters in XQuery expressions.

Host variables are declared using the host language specific variable declaration syntax in a declaration section.

A declaration section is the portion of an embedded SQL application found near the top of an embedded SQL source code file, and is bounded by two non-executable SQL statements:

- BEGIN DECLARE SECTION
- END DECLARE SECTION

These statements enable the precompiler to find the variable declarations. Each host variable declaration must appear in between these two statements, otherwise the variables are considered to be only regular variables.

The following rules apply to host variable declaration sections:

- All host variables must be declared in the source file within a well formed declaration section before they are referenced, except for host variables referring to SQLDA structures.

- Multiple declare sections can be used in one source file.
- Host variable names must be unique within a source file. This is because the DB2 precompiler does not account for host language-specific variable scoping rules. As such, there is only one scope for host variables.

  **Note:** This does not mean that the DB2 precompiler changes the scope of host variables to global so that they can be accessed outside the scope in which they are defined.

Consider the following example:

```
foo1(){
  .
  .
  .
  BEGIN SQL DECLARE SECTION;
  int x;
  END SQL DECLARE SECTION;
x=10;
  .
  .
  .
}


foo2(){
  .
  .
  .
  y=x;
  .
  .
  .
}
```

Depending on the language, the above example will either fail to compile because variable `x` is not declared in function `foo2()`, or the value of `x` would not be set to 10 in `foo2()`. To avoid this problem, you must either declare `x` as a global variable, or pass `x` as a parameter to function `foo2()` as follows:

```
  foo1(){
  .
  .
  .
    BEGIN SQL DECLARE SECTION;
    int x;
    END SQL DECLARE SECTION;
    x=10;
    foo2(x);
  .
  .
  .
  }


  foo2(int x){
  .
  .
  .
    y=x;
  .
  .
  .
  }
```

**Related concepts:**

- "Host variables in C and C++ embedded SQL applications" on page 78
- "Embedding SQL statements in a host language" on page 4
- "Embedded SQL application template in C" on page 39

**Related tasks:**

- "Declaring host variables in embedded SQL applications" on page 68
- "Declaring Host Variables with the db2dclgn Declaration Generator" on page 69
- "Referencing host variables in embedded SQL applications" on page 76

**Related reference:**

- "END DECLARE SECTION statement" in *SQL Reference, Volume 2*
- "BEGIN DECLARE SECTION statement" in *SQL Reference, Volume 2*

## Declaring host variables in embedded SQL applications

To transmit data between the database server and the application, you need to declare host variables in your application source code for things such as relational SQL queries and host variable declarations for XQuery expressions.

**Procedure:**

The following table provides examples of host variable declarations for embedded SQL host languages.

*Table 13. Host Variable Declarations by Host Language*

| Language | Example Source Code |
|---|---|
| C and C++ | `EXEC SQL BEGIN DECLARE SECTION;`<br>`  short     dept=38, age=26;`<br>`  double    salary;`<br>`  char      CH;`<br>`  char      name1[9], NAME2[9];`<br>`  short     nul_ind;`<br>`EXEC SQL END DECLARE SECTION;` |
| COBOL | ` EXEC SQL BEGIN DECLARE SECTION END-EXEC.`<br>`   01 age        PIC S9(4) COMP-5 VALUE 26.`<br>`   01 DEPT       PIC S9(9) COMP-5 VALUE 38.`<br>`   01 salary     PIC S9(6)V9(3) COMP-3.`<br>`   01 CH         PIC X(1).`<br>`   01 name1      PIC X(8).`<br>`   01 NAME2      PIC X(8).`<br>`   01 nul-ind    PIC S9(4) COMP-5.`<br>` EXEC SQL END DECLARE SECTION END-EXEC.` |
| FORTRAN | `       EXEC SQL BEGIN DECLARE SECTION`<br>`         integer*2     age   /26/`<br>`         integer*4     dept  /38/`<br>`         real*8        salary`<br>`         character     ch`<br>`         character*8   name1,NAME2`<br>`         integer*2     nul_ind`<br>`       EXEC SQL END DECLARE SECTION` |

**Related concepts:**

- "Embedding SQL statements in a host language" on page 4

**Related tasks:**

- "Declaring Host Variables with the db2dclgn Declaration Generator" on page 69
- "Referencing host variables in embedded SQL applications" on page 76

## Declaring Host Variables with the db2dclgn Declaration Generator

You can use the Declaration Generator to generate declarations for a given table in a database. It creates embedded SQL declaration source files which you can easily insert into your applications. `db2dclgn` supports the C/C++, Java, COBOL, and FORTRAN languages.

**Procedure:**

To generate declaration files, enter the `db2dclgn` command in the following format:

```
db2dclgn -d database-name -t table-name [options]
```

For example, to generate the declarations for the STAFF table in the SAMPLE database in C in the output file `staff.h`, issue the following command:

```
db2dclgn -d sample -t staff -l C
```

The resulting `staff.h` file contains:

```
struct
{
  short id;
  struct
  {
    short length;
    char data[9];
  } name;
  short dept;
  char job[6];
  short years;
  double salary;
  double comm;
} staff;
```

**Related reference:**

- "db2dclgn - Declaration generator command" in *Command Reference*

## Column data types and host variables in embedded SQL applications

Each column of every DB2 table is given an *SQL data type* when the column is created. For information about how these types are assigned to columns, see the CREATE TABLE statement.

**Notes:**

1. Every supported data type can have the NOT NULL attribute. This is treated as another type.
2. Data types can be extended by defining user-defined distinct types (UDT). UDTs are separate data types that use the representation of one of the built-in SQL types.

Supported embedded SQL host languages have data types that correspond to the majority of the database manager data types. Only these host language data types can be used in host variable declarations. When the precompiler finds a host variable declaration, it determines the appropriate SQL data type value. The database manager uses this value to convert the data exchanged between itself and the application.

As the application programmer, it is important for you to understand how the database manager handles comparisons and assignments between different data types. Simply put, data types must be compatible with each other during assignment and comparison operations, whether the database manager is working with two SQL column data types, two host-language data types, or one of each.

The *general* rule for data type compatibility is that all supported host-language numeric data types are comparable and assignable with all database manager numeric data types, and all host-language character types are compatible with all database manager character types; numeric types are incompatible with character types. However, there are also some exceptions to this general rule, depending on host language idiosyncrasies and limitations imposed when working with large objects.

Within SQL statements, DB2 provides conversions between compatible data types. For example, in the following SELECT statement, SALARY and BONUS are DECIMAL columns; however, each employee's total compensation is returned as DOUBLE data:

```
SELECT EMPNO, DOUBLE(SALARY+BONUS) FROM EMPLOYEE
```

Note that the execution of the above statement includes conversion between DECIMAL and DOUBLE data types.

To make the query results more readable on your screen, you could use the following SELECT statement:

```
SELECT EMPNO, DIGIT(SALARY+BONUS) FROM EMPLOYEE
```

The `DIGITS` function used in the example above returns a character-string representation of a number.

To convert data within your application, contact your compiler vendor for additional routines, classes, built-in types, or APIs that support this conversion.

If your application code page is not the same as your database code page, character data types may also be subject to character conversion.

**Related concepts:**

- "Data conversion considerations" in *Developing SQL and External Routines*
- "Character conversion between different code pages" in *Developing SQL and External Routines*
- "Data types that map to SQL data types in embedded SQL applications" on page 53
- "Embedding SQL statements in a host language" on page 4

**Related reference:**

- "CREATE TABLE statement" in *SQL Reference, Volume 2*
- "Supported SQL data types in C and C++ embedded SQL applications" on page 53
- "Supported SQL data types in COBOL embedded SQL applications" on page 60

- “Supported SQL data types in FORTRAN embedded SQL applications” on page 63
- “Supported SQL data types in REXX embedded SQL applications” on page 64

## Declaring XML host variables in embedded SQL applications

To exchange XML data between the database server and an embedded SQL application, you need to declare host variables in your application source code.

DB2 V9.1 introduces an XML data type that stores XML data in a structured set of nodes in a tree format. Columns with this XML data type are described as an SQL_TYP_XML column SQLTYPE, and applications can bind various language-specific data types for input to and output from these columns or parameters. XML columns can be accessed directly using SQL, the SQL/XML extensions, or XQuery. The XML data type applies to more than just columns. Functions can have XML value arguments and produce XML values as well. Similarly, stored procedures can take XML values as both input and output parameters. Finally, XQuery expressions produce XML values regardless of whether or not they access XML columns.

XML data is character in nature and has an encoding that specifies the character set used. The encoding of XML data can be determined externally, derived from the base application type containing the serialized string representation of the XML document. It can also be determined internally, which requires interpretation of the data. For Unicode encoded documents, a byte order mark (BOM), consisting of a Unicode character code at the beginning of a data stream is recommended. The BOM is used as a signature that defines the byte order and Unicode encoding form.

Existing character and binary types, which include CHAR, VARCHAR, CLOB, and BLOB may be used in addition to XML host variables for fetching and inserting data. However, they will not be subject to implicit XML parsing, as XML host variables would. Instead, an explicit XMLPARSE function with default whitespace stripping is injected and applied.

**Restrictions:**

XML and XQuery restrictions on developing embedded SQL applications

**Procedure:**

To declare XML host variables in embedded SQL applications:

In the declaration section of the application, declare the XML host variables as LOB data types:

- `SQL TYPE IS XML AS CLOB(n) <hostvar_name>`

  where <hostvar_name> is a CLOB host variable that contains XML data encoded in the mixed codepage of the application.
- `SQL TYPE IS XML AS DBCLOB(n) <hostvar_name>`

  where <hostvar_name> is a DBCLOB host variable that contains XML data encoded in the application graphic codepage.
- `SQL TYPE IS XML AS BLOB(n) <hostvar_name>`

where <hostvar_name> is a BLOB host variable that contains XML data internally encoded[1].

- `SQL TYPE IS XML AS CLOB_FILE <hostvar_name>`

  where <hostvar_name> is a CLOB file that contains XML data encoded in the application mixed codepage.
- `SQL TYPE IS XML AS DBCLOB_FILE <hostvar_name>`

  where <hostvar_name> is a DBCLOB file that contains XML data encoded in the application graphic codepage.
- `SQL TYPE IS XML AS BLOB_FILE <hostvar_name>`

  where <hostvar_name> is a BLOB file that contains XML data internally encoded[1].

**Notes:**

1. Refer to the algorithm for determining encoding with XML 1.0 specifications (`http://www.w3.org/TR/REC-xml/#sec-guessing-no-ext-info`).

**Related concepts:**

- "Data types that map to SQL data types in embedded SQL applications" on page 53
- "Example: Referencing XML host variables in embedded SQL applications" on page 76
- "Background information on XML internal encoding" in *XML Guide*
- "XML data encoding" in *XML Guide*

**Related tasks:**

- "Executing XQuery expressions in embedded SQL applications" on page 151

**Related reference:**

- "Recommendations for developing embedded SQL applications with XML and XQuery" on page 27

## Identifying XML values in an SQLDA

To indicate that a base type holds XML data, the sqlname field of the SQLVAR must be updated as follows:

- `sqlname.length` must be 8
- The first two bytes of sqlname.data must be `X'0000'`
- The third and fourth bytes of sqlname.data should be `X'0000'`
- The fifth byte of `sqlname.data` must be `X'01'` (referred to as the XML subtype indicator only when the first two conditions are met)
- The remaining bytes should be `X'000000'`

If the XML subtype indicator is set in an SQLVAR whose SQLTYPE is non-LOB, an SQL0804 error (rc=115) will be returned at runtime.

**Note:** SQL_TYP_XML can only be returned from the DESCRIBE statement. This type cannot be used for any other requests. The application must modify the SQLDA to contain a valid character or binary type, and set the `sqlname` field appropriately to indicate that the data is XML.

**Related concepts:**
- "Host Variables in embedded SQL applications" on page 66

**Related tasks:**
- "Referencing host variables in embedded SQL applications" on page 76
- "Declaring host variables in embedded SQL applications" on page 68
- "Declaring the SQLDA structure in a dynamically executed SQL program" on page 138

## Identifying null SQL values with null indicator variables

Embedded SQL applications must prepare for receiving null values by associating a *null-indicator variable* with any host variable that can receive a null. A indicator variable is shared by both the database manager and the host application. Therefore, this variable must be declared in the application as a host variable, which corresponds to the SQL data type SMALLINT.

A null-indicator variable is placed in an SQL statement immediately after the host variable, and is prefixed with a colon. A space can separate the null-indicator variable from the host variable, but is not required. However, do not put a comma between the host variable and the null-indicator variable. You can also specify a null-indicator variable by using the optional INDICATOR keyword, which you place between the host variable and its null indicator.

The null-indicator variable is examined for a negative value. If the value is not negative, the application can use the returned value of the host variable. If the value is negative, the fetched value is null and the host variable should not be used. The database manager does not change the value of the host variable in this case.

**Note:** If the database configuration parameter *dft_sqlmathwarn* is set to 'YES', the null-indicator variable value may be -2. This value indicates a null that was either caused by evaluating an expression with an arithmetic error, or by an overflow while attempting to convert the numeric result value to the host variable.

If the data type can handle nulls, the application must provide a null indicator. Otherwise, an error may occur. If a null indicator is not used, an SQLCODE -305 (SQLSTATE 22002) is returned.

If the SQLCA structure indicates a truncation warning, the null-indicator variables can be examined for truncation. If a null-indicator variable has a positive value, a truncation occurred.
- If the seconds' portion of a TIME data type is truncated, the null-indicator value contains the seconds portion of the truncated data.
- For all other string data types, except large objects (LOB), the null-indicator value represents the actual length of the data returned. User-defined distinct types (UDT) are handled in the same way as their base type.

When processing INSERT or UPDATE statements, the database manager checks the null-indicator variable, if one exists. If the indicator variable is negative, the database manager sets the target column value to null, if nulls are allowed.

If the null-indicator variable is zero or positive, the database manager uses the value of the associated host variable.

The SQLWARN1 field in the SQLCA structure might contain an `X` or `W` if the value of a string column is truncated when it is assigned to a host variable. The field contains an `N` if a null terminator is truncated.

A value of `X` is returned by the database manager only if all of the following conditions are met:

- A mixed code page connection exists where conversion of character string data from the database code page to the application code page involves a change in the length of the data.
- A cursor is blocked.
- A null-indicator variable is provided by your application.

The value returned in the null-indicator variable will be the length of the resultant character string in the application's code page.

In all other cases involving data truncation (as opposed to null terminator truncation), the database manager returns a `W`. In this case, the database manager returns a value in the null-indicator variable to the application that is the length of the resultant character string in the code page of the select list item (either the application code page, the database code page, or nothing).

Before you can use null-indicator variables in the host language, you need to declare the null-indicator variables. In the following example, suitable for C and C++ programs, the null-indicator variable cmind can be declared as:

```
EXEC SQL BEGIN DECLARE SECTION;
  char cm[3];
  short cmind;
EXEC SQL END DECLARE SECTION;
```

The following table provides examples for the supported host languages:

*Table 14. Null-Indicator Variables by Host Language*

| Language | Example Source Code |
|---|---|
| C and C++ | `EXEC SQL FETCH C1 INTO :cm INDICATOR :cmind;` |
| | `if ( cmind < 0 )` |
| | `   printf( "Commission is NULL\n" );` |
| COBOL | `EXEC SQL FETCH C1 INTO :cm INDICATOR :cmind END-EXEC` |
| | `IF cmind LESS THAN 0` |
| | `   DISPLAY 'Commission is NULL'` |
| FORTRAN | `EXEC SQL FETCH C1 INTO :cm INDICATOR :cmind` |
| | `IF ( cmind .LT. 0 ) THEN` |
| | `   WRITE(*,*) 'Commission is NULL'` |
| | `ENDIF` |
| REXX | `   CALL SQLEXEC 'FETCH C1 INTO :cm INDICATOR :cmind'` |
| | `   IF ( cmind < 0 )` |
| | `      SAY 'Commission is NULL'` |

**Related concepts:**

- "Data types that map to SQL data types in embedded SQL applications" on page 53
- "Error information in the SQLCODE, SQLSTATE, and SQLWARN fields" on page 173

- “Error message retrieval in embedded SQL applications” on page 174

**Related tasks:**

- “Declaring host variables in embedded SQL applications” on page 68
- “Declaring Host Variables with the db2dclgn Declaration Generator” on page 69
- “Referencing host variables in embedded SQL applications” on page 76

**Related reference:**

- “Column data types and host variables in embedded SQL applications” on page 69

## Including SQLSTATE and SQLCODE host variables in embedded SQL applications

Error information is returned in the SQLCODE and SQLSTATE fields of the SQLCA structure, which is updated after every executable SQL statement and most database manager API calls. If your application is compliant with the FIPS 127-2 standard, you can declare host variables named SQLSTATE and SQLCODE instead of explicitly declaring the SQLCA structure in embedded SQL applications.

**Prerequisites:**

- The PREP option LANGLEVEL SQL92E needs to be specified

**Procedure:**

In the following example, the application checks the `SQLCODE` field of the `SQLCA` structure to determine whether the update was successful.

*Table 15. Embedding SQL Statements in a Host Language*

| Language | Sample Source Code |
|---|---|
| C and C++ | `EXEC SQL UPDATE staff SET job = 'Clerk' WHERE job = 'Mgr';`<br>`if ( SQLCODE < 0 )`<br>`   printf( "Update Error:  SQLCODE = %ld \n", SQLCODE );` |
| COBOL | `EXEC SQL UPDATE staff SET job = 'Clerk' WHERE job = 'Mgr' END_EXEC.`<br>`IF SQLCODE LESS THAN 0`<br>`   DISPLAY 'UPDATE ERROR:  SQLCODE = ', SQLCODE.` |
| FORTRAN | `EXEC SQL UPDATE staff SET job = 'Clerk' WHERE job = 'Mgr'`<br>`if ( sqlcode .lt. 0 ) THEN`<br>`   write(*,*) 'Update error:  sqlcode = ', sqlcode` |

**Related concepts:**

- “Embedded SQL application template in C” on page 39
- “Executing SQL statements in embedded SQL applications” on page 136
- “Host Variables in embedded SQL applications” on page 66

**Related tasks:**

- “Declaring the SQLCA for Error Handling” on page 50

**Related reference:**

- “PREPARE statement” in *SQL Reference, Volume 2*

## Referencing host variables in embedded SQL applications

Once you have declared a host variable in your embedded SQL application code, you can reference it later in the application. When you use a host variable in an SQL statement, prefix its name with a colon (:). If you use a host variable in host language programming syntax, omit the colon.

**Procedure:**

Reference the host variables using the syntax for the host language that you are using. The following table provides examples.

*Table 16. Host Variable References by Host Language*

| Language | Example Source Code |
|---|---|
| C or C++ | `EXEC SQL FETCH C1 INTO :cm;`<br>`printf( "Commission = %f\n", cm );` |
| COBOL | `EXEC SQL FETCH C1 INTO :cm END-EXEC`<br>`DISPLAY 'Commission = ' cm` |
| FORTRAN | `EXEC SQL FETCH C1 INTO :cm`<br>`WRITE(*,*) 'Commission = ', cm` |
| REXX | `CALL SQLEXEC 'FETCH C1 INTO :cm'`<br>`SAY 'Commission = ' cm` |

**Related concepts:**

- "Host Variables in embedded SQL applications" on page 66
- "Embedded SQL application template in C" on page 39
- "Embedding SQL statements in a host language" on page 4

**Related tasks:**

- "Declaring host variables in embedded SQL applications" on page 68
- "Declaring Host Variables with the db2dclgn Declaration Generator" on page 69

## Example: Referencing XML host variables in embedded SQL applications

The following sample applications demostrate how to reference XML host variables in C and COBOL.

**Example: Embedded SQL C application::**

```
The following code example has been formatted for clarity:
EXEC SQL BEGIN DECLARE;
  SQL TYPE IS XML AS CLOB( 10K ) xmlBuf;
  SQL TYPE IS XML AS BLOB( 10K ) xmlblob;
  SQL TYPE IS CLOB( 10K ) clobBuf;
EXEC SQL END DECLARE SECTION;

// as XML AS CLOB
// The XML value written to xmlBuf will be prefixed by an XML declaration
// similar to: <?xml version = "1.0" encoding = "ISO-8859-1" ?>
// Note: The encoding name will depend upon the application codepage
EXEC SQL SELECT xmlCol INTO :xmlBuf
   FROM myTable
   WHERE id = '001';
EXEC SQL UPDATE myTable
   SET xmlCol = :xmlBuf
   WHERE id = '001';
```

```
// as XML AS BLOB
// The XML value written to xmlblob will be prefixed by an XML declaration
// similar to: <?xml version = "1.0" encoding = "UTF-8"?>
EXEC SQL SELECT xmlCol INTO :xmlblob
   FROM myTable
   WHERE id = '001';
EXEC SQL UPDATE myTable
   SET xmlCol = :xmlblob
   WHERE id = '001';

// as CLOB
// The output will be encoded in the application character codepage,
// but will not contain an XML declaration
EXEC SQL SELECT XMLSERIALIZE (xmlCol AS CLOB(10K)) INTO :clobBuf
   FROM myTable
   WHERE id = '001';
EXEC SQL UPDATE myTable
   SET xmlCol = XMLPARSE (:clobBuf PRESERVE WHITESPACE)
   WHERE id = '001';
```

### Example: Embedded SQL COBOL application::

```
The following code example has been formatted for clarity:
EXEC SQL BEGIN DECLARE SECTION END-EXEC.
  01 xmlBuf USAGE IS SQL TYPE IS XML as CLOB(5K).
  01 clobBuf USAGE IS SQL TYPE IS CLOB(5K).
  01 xmlblob  USAGE IS SQL TYPE IS BLOB(5K).
EXEC SQL END DECLARE SECTION END-EXEC.

* as XML
EXEC SQL SELECT xmlCol INTO :xmlBuf
   FROM myTable
   WHERE id = '001'.
EXEC SQL UPDATE myTable
   SET xmlCol = :xmlBuf
   WHERE id = '001'.

* as BLOB
EXEC SQL SELECT xmlCol INTO :xmlblob
   FROM myTable
   WHERE id = '001'.
EXEC SQL UPDATE myTable
   SET xmlCol = :xmlblob
   WHERE id = '001'.

* as CLOB
EXEC SQL SELECT XMLSERIALIZE(xmlCol AS CLOB(10K)) INTO :clobBuf
   FROM myTable
   WHERE id= '001'.
EXEC SQL UPDATE myTable
   SET xmlCol = XMLPARSE(:clobBuf) PRESERVE WHITESPACE
   WHERE id = '001'.
```

**Related tasks:**

- “Declaring XML host variables in embedded SQL applications” on page 71
- “Executing XQuery expressions in embedded SQL applications” on page 151

**Related reference:**

- “Recommendations for developing embedded SQL applications with XML and XQuery” on page 27

## Host variables in C and C++ embedded SQL applications

### Host variables in C and C++ embedded SQL applications

Host variables are C or C++ language variables that are referenced within SQL statements. They allow an application to exchange data with the database manager. After the application is precompiled, host variables are used by the compiler as any other C or C++ variable. Follow the rules described in the following sections when naming, declaring, and using host variables.

**Long variable considerations:**

In applications that manually construct the SQLDA, long variables cannot be used when `sqlvar::sqltype==SQL_TYP_INTEGER`. Instead, `sqlint32` types must be used. This problem is identical to using long variables in host variable declarations, except that with a manually constructed SQLDA, the precompiler will not uncover this error and run time errors will occur.

Any long and unsigned long casts that are used to access sqlvar::sqldata information must be changed to `sqlint32` and `sqluint32` respectively. `Val` members for the `sqloptions` and `sqla_option` structures are declared as `sqluintptr`. Therefore, assignment of pointer members into `sqla_option::val` or `sqloptions::val` members should use `sqluintptr` casts rather than unsigned long casts. This change will not cause run-time problems in 64-bit UNIX and Linux operating systems, but should be made in preparation for 64-bit Windows applications, where the long type is only 32-bit.

**Multi-byte encoding considerations:**

Some character encoding schemes, particularly those from east-Asian countries, require multiple bytes to represent a character. This external representation of data is called the *multi-byte character code* representation of a character, and includes double-byte characters (characters represented by two bytes). Host variables will be chosen accordingly since graphic data in DB2 consists of double-byte characters.

To manipulate character strings with double-byte characters, it may be convenient for an application to use an internal representation of data. This internal representation is called the *wide-character code* representation of the double-byte characters, and is the format customarily used in the `wchar_t` C or C++ data type. Subroutines that conform to ANSI C and X/OPEN Portability Guide 4 (XPG4) are available to process wide-character data, and to convert data in wide-character format to and from multi-byte format.

Note that although an application can process character data in either multi-byte format or wide-character format, interaction with the database manager is done with DBCS (multi-byte) character codes only. That is, data is stored in and retrieved from GRAPHIC columns in DBCS format. The WCHARTYPE precompiler option is provided to allow application data in wide-character format to be converted to/from multi-byte format when it is exchanged with the database engine.

**Related concepts:**

- "Declaration of graphic host variables in C and C++ embedded SQL applications" on page 86

- "Host structure support in the declare section of C and C++ embedded SQL applications" on page 103
- "Declare section for host variables in C and C++ embedded SQL applications" on page 80
- "Initialization of host variables in C and C++ embedded SQL applications" on page 101
- "Host variable names in C and C++ embedded SQL applications" on page 79
- "Example: SQL declare section template for C and C++ embedded SQL applications" on page 81
- "Declaration of fixed-length, null-terminated and variable-length character host variables in C and C++ embedded SQL applications" on page 84

**Related reference:**
- "Declaration of file reference type host variables in C and C++ embedded SQL applications" on page 96
- "Declaration of large object type host variables in C and C++ embedded SQL applications" on page 93
- "Declaration of large object locator type host variables in C and C++ embedded SQL applications" on page 95
- "Declaration of numeric host variables in C and C++ embedded SQL applications" on page 83

## Host variable names in C and C++ embedded SQL applications

The SQL precompiler identifies host variables by their declared name. The following rules apply:
- Host variable names must be no longer than 255 characters in length.
- Host variable names must not use the prefix SQL, sql, DB2, and db2, which are reserved for system use. For example:
```
EXEC SQL BEGIN DECLARE SECTION;
  char  varsql;     /* allowed */
  char  sqlvar;     /* not allowed */
  char  SQL_VAR;    /* not allowed */
EXEC SQL END DECLARE SECTION;
```
- The precompiler considers host variable names as global to a module. This does not mean, however, that host variables have to be declared as global variables; it is perfectly acceptable to declare host variables as local variables within functions. For example, the following code will work correctly:
```
void f1(int i)
{
EXEC SQL BEGIN DECLARE SECTION;
  short host_var_1;
EXEC SQL END DECLARE SECTION;
EXEC SQL SELECT COL1 INTO :host_var_1 from TBL1;
}
void f2(int i)
{
EXEC SQL BEGIN DECLARE SECTION;
  short host_var_2;
EXEC SQL END DECLARE SECTION;
EXEC SQL INSERT INTO TBL1 VALUES (:host_var_2);
}
```
  It is also possible to have several local host variables with the same name, as long as they all have the same type and size. To do this, declare the first occurrence of the host variable to the precompiler between BEGIN DECLARE

SECTION and END DECLARE SECTION statements, and leave subsequent declarations of the variable out of declare sections. The following code shows an example of this:

```
void f3(int i)
{
EXEC SQL BEGIN DECLARE SECTION;
  char host_var_3[25];
EXEC SQL END DECLARE SECTION;
EXEC SQL SELECT COL2 INTO :host_var_3 FROM TBL2;
}
void f4(int i)
{
char host_var_3[25];
EXEC SQL INSERT INTO TBL2 VALUES (:host_var_3);
}
```

Because `f3` and `f4` are in the same module, and `host_var_3` has the same type and length in both functions, a single declaration to the precompiler is sufficient to use it in both places.

**Related concepts:**

- "Declare section for host variables in C and C++ embedded SQL applications" on page 80
- "Data types that map to SQL data types in embedded SQL applications" on page 53

**Related reference:**

- "C samples" on page 369
- "C++ samples" in *Samples Topics*
- "Data types for procedures, functions, and methods in C and C++ embedded SQL applications" on page 58

## Declare section for host variables in C and C++ embedded SQL applications

An SQL declare section must be used to identify host variable declarations. This alerts the precompiler to any host variables that can be referenced in subsequent SQL statements. For example:

```
EXEC SQL BEGIN DECLARE SECTION;
  char  varsql;     /* allowed */
EXEC SQL END DECLARE SECTION;
```

The C or C++ precompiler only recognizes a subset of valid C or C++ declarations as valid host variable declarations. These declarations define either numeric or character variables. Type definitions for host variable types are not allowed. Host variables can be grouped into a single host structure. You can declare C++ class data members as host variables.

A numeric host variable can be used as an input or output variable for any numeric SQL input or output value. A character host variable can be used as an input or output variable for any character, date, time, or timestamp SQL input or output value. The application must ensure that output variables are long enough to contain the values that they receive.

**Related concepts:**

- "Declaration of class data members as host variables in C++ embedded SQL applications" on page 98

- “Declaration of graphic host variables in C and C++ embedded SQL applications” on page 86
- “Host structure support in the declare section of C and C++ embedded SQL applications” on page 103
- “Declaration of fixed-length, null-terminated and variable-length character host variables in C and C++ embedded SQL applications” on page 84

**Related tasks:**
- “Declaring Host Variables with the db2dclgn Declaration Generator” on page 69
- “Declaring structured type host variables” in *Developing SQL and External Routines*

**Related reference:**
- “Declaration of numeric host variables in C and C++ embedded SQL applications” on page 83

## Example: SQL declare section template for C and C++ embedded SQL applications

The following is a sample SQL declare section with host variables declared for supported SQL data types:

```
   EXEC SQL BEGIN DECLARE SECTION;

⋮
        short     age = 26;              /* SQL type  500 */
        short     year;                  /* SQL type  500 */
        sqlint32  salary;                /* SQL type  496 */
        sqlint32  deptno;                /* SQL type  496 */
        float     bonus;                 /* SQL type  480 */
        double    wage;                  /* SQL type  480 */
        char      mi;                    /* SQL type  452 */
        char      name[6];               /* SQL type  460 */
        struct   {
                  short len;
                  char data[24];
                 } address;              /* SQL type  448 */
        struct   {
                  short len;
                  char data[32695];
                 } voice;                /* SQL type  456 */
        sql type is clob(1m)
                  chapter;               /* SQL type  408 */
        sql type is clob_locator
                  chapter_locator;       /* SQL type  964 */
        sql type is clob_file
                  chapter_file_ref;      /* SQL type  920 */
        sql type is blob(1m)
                  video;                 /* SQL type  404 */
        sql type is blob_locator
                  video_locator;         /* SQL type  960 */
        sql type is blob_file
                  video_file_ref;        /* SQL type  916 */
        sql type is dbclob(1m)
                  tokyo_phone_dir;       /* SQL type  412 */
        sql type is dbclob_locator
                  tokyo_phone_dir_lctr;  /* SQL type  968 */
        sql type is dbclob_file
                  tokyo_phone_dir_flref; /* SQL type  924 */
        struct   {
                  short len;
                  sqldbchar data[100];
                 } vargraphic1;          /* SQL type  464 */
                                          /* Precompiled with
                                          WCHARTYPE NOCONVERT option */
        struct   {
                  short len;
                  wchar_t data[100];
                 } vargraphic2;          /* SQL type  464 */
                                          /* Precompiled with
                                          WCHARTYPE CONVERT option */
        struct   {
                  short len;
                  sqldbchar data[10000];
                 } long_vargraphic1;     /* SQL type  472 */
                                          /* Precompiled with
                                          WCHARTYPE NOCONVERT option */
        struct   {
                  short len;
                  wchar_t data[10000];
                 } long_vargraphic2;     /* SQL type  472 */
                                          /* Precompiled with
                                          WCHARTYPE CONVERT option */
        sqldbchar graphic1[100];         /* SQL type  468 */
                                          /* Precompiled with
                                          WCHARTYPE NOCONVERT option */
        wchar_t   graphic2[100];         /* SQL type  468 */
                                          /* Precompiled with
                                          WCHARTYPE CONVERT option */
        char      date[11];              /* SQL type  384 */
        char      time[9];               /* SQL type  388 */
        char      timestamp[27];         /* SQL type  392 */
        short     wage_ind;              /* Null indicator */

.
.
.
   EXEC SQL END DECLARE SECTION;
```

**Related concepts:**

- "Declare section for host variables in C and C++ embedded SQL applications" on page 80
- "Host variable names in C and C++ embedded SQL applications" on page 79
- "Host variables in C and C++ embedded SQL applications" on page 78
- "Initialization of host variables in C and C++ embedded SQL applications" on page 101

**Related tasks:**

- "Declaring host variables in embedded SQL applications" on page 68

## SQLSTATE and SQLCODE variables in C and C++ embedded SQL application

When using the LANGLEVEL precompile option with a value of SQL92E, the following two declarations can be included as host variables:

```
EXEC SQL BEGIN DECLARE SECTION;
  char      SQLSTATE[6]
  sqlint32  SQLCODE;


EXEC SQL END DECLARE SECTION;
```

The SQLCODE declaration is assumed during the precompile step. Note that when using this option, the INCLUDE SQLCA statement should not be specified.

In an application that is made up of multiple source files, the SQLCODE and SQLSTATE variables can be defined in the first source file as above. Subsequent source files should modify the definitions as follows:

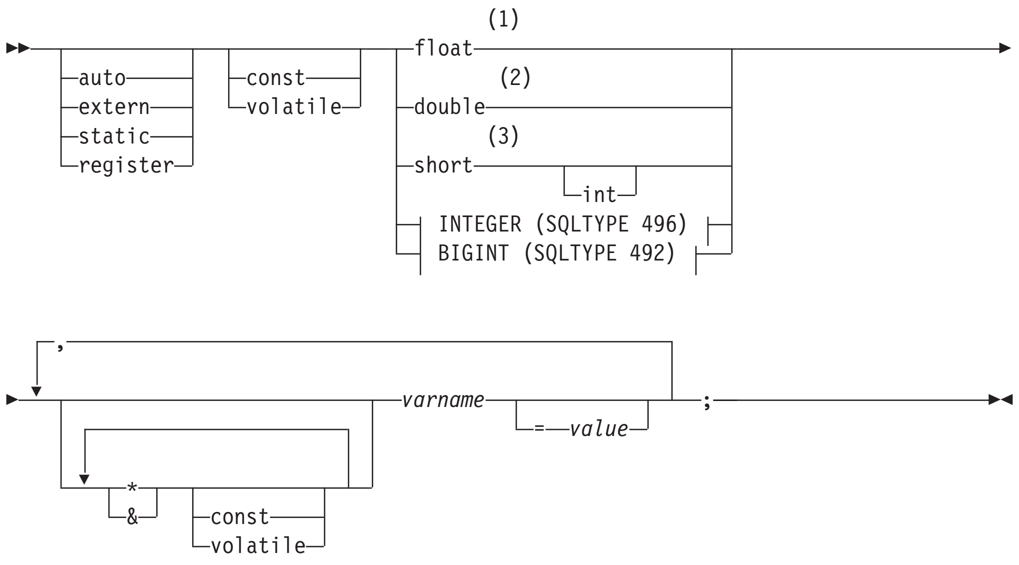
```
extern sqlint32 SQLCODE;
extern char     SQLSTATE[6];
```

**Related concepts:**

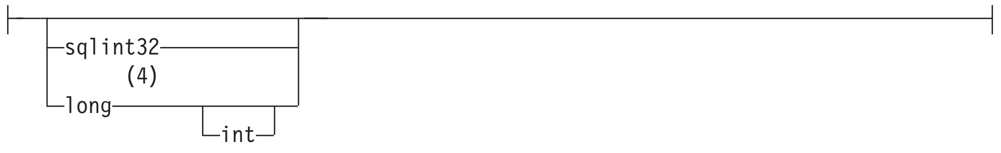
- "Declare section for host variables in C and C++ embedded SQL applications" on page 80
- "Error information in the SQLCODE, SQLSTATE, and SQLWARN fields" on page 173
- "Host variables in C and C++ embedded SQL applications" on page 78

## Declaration of numeric host variables in C and C++ embedded SQL applications

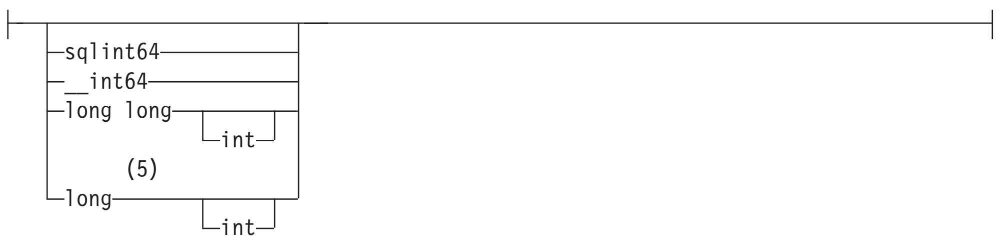
Following is the syntax for declaring numeric host variables in C or C++.



### INTEGER (SQLTYPE 496)



### BIGINT (SQLTYPE 492)



**Notes:**

1 REAL (SQLTYPE 480), length 4

2 DOUBLE (SQLTYPE 480), length 8

3 SMALLINT (SQLTYPE 500)

4 For maximum application portability, use `sqlint32` and `sqlint64` for INTEGER and BIGINT host variables, respectively. By default, the use of long host variables results in the precompiler error SQL0402 on platforms where long is a 64 bit quantity, such as 64 BIT UNIX. Use the PREP option LONGERROR NO to force DB2 to accept long variables as acceptable host variable types and treat them as BIGINT variables.

5 For maximum application portability, use `sqlint32` and `sqlint64` for INTEGER and BIGINT host variables, respectively. To use the BIGINT data type, your platform must support 64 bit integer values. By default, the use of long host variables results in the precompiler error SQL0402 on platforms where long is a 64 bit quantity, such as 64 BIT UNIX. Use the PREP option LONGERROR NO to force DB2 to accept long variables as acceptable host variable types and treat them as BIGINT variables.

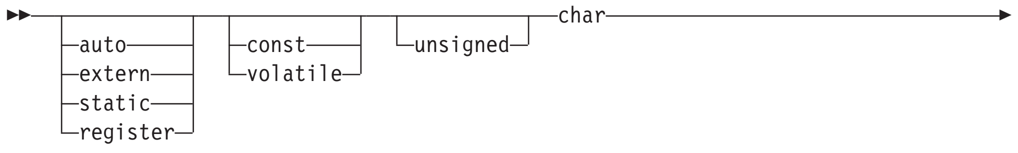
**Related concepts:**

- "Data types that map to SQL data types in embedded SQL applications" on page 53
- "Host variables in C and C++ embedded SQL applications" on page 78

## Declaration of fixed-length, null-terminated and variable-length character host variables in C and C++ embedded SQL applications

Following are the syntax for declaring fixed, null-terminated (form 1) and Variable-Length (form 2) character host variables in C or C++.

### Form 1: Syntax for fixed and null-terminated character host variables in C or C++ embedded SQL applications



```
   ┌─,──────────────────────────────┐
   ▼                                │
►──┬─┤ CHAR ├─────┬──┬──────────┬───┴─;──────────────────────────────►◄
   └─┤ C String ├─┘  └─=──value─┘
```

**CHAR**

```
                                     (1)
├──┬────────────────────────┬─varname─────────────────────────────────┤
   │ ┌────────────────────┐ │
   │ ▼                    │ │
   └───┬─*─┬──┬──────────┬┴─┘
       └─&─┘  ├─const────┤
              └─volatile─┘
```

**C String**

```
                                                            (2)
├──┬─varname──────────────────────────────────────┬─[length]─────────┤
   └─(──┬────────────────────────┬─varname──)─────┘
        │ ┌────────────────────┐ │
        │ ▼                    │ │
        └───┬─*─┬──┬──────────┬┴─┘
            └─&─┘  ├─const────┤
                   └─volatile─┘
```

**Notes:**

1 CHAR (SQLTYPE 452), length 1

2 Null-terminated C string (SQLTYPE 460); length can be any valid constant expression

**Form 2: Syntax for variable-length character host variables in C and C++ embedded SQL applications**

```
►►──┬──────────┬──┬──────────┬─struct──┬─────┬────────────────────────►
    ├─auto─────┤  ├─const────┤         └─tag─┘
    ├─extern───┤  └─volatile─┘
    ├─static───┤
    └─register─┘
```

```
                                                               (1)
►─{──short──┬─────┬─var1──;──┬──────────┬─char──var2──[length]──────;──}──────►
            └─int─┘          └─unsigned─┘
```

```
   ┌─,─────────────────────────────────────────────────┐
   ▼                                                   │
►──┬────────────────────────┬─varname──┬────────────┬──┴─;────────►◄
   │ ┌────────────────────┐ │          └─┤ Values ├─┘
   │ ▼                    │ │
   └───┬─*─┬──┬──────────┬┴─┘
       └─&─┘  ├─const────┤
              └─volatile─┘
```

**Values**

```
├──=──{──value-1──,──value-2──}──────────────────────────────────────────┤
```

**Notes:**

1 In form 2, length can be any valid constant expression. Its value after evaluation determines if the host variable is VARCHAR (SQLTYPE 448) or LONG VARCHAR (SQLTYPE 456).

**Variable-Length Character Host Variable Considerations:**

1. Although the database manager converts character data to either **form 1** or **form 2** whenever possible, **form 1** corresponds to column types CHAR or VARCHAR, while **form 2** corresponds to column types VARCHAR and LONG VARCHAR.
2. If **form 1** is used with a length specifier *[n]*, the value for the length specifier after evaluation must be no greater than 32 672, and the string contained by the variable should be null-terminated.
3. If **form 2** is used, the value for the length specifier after evaluation must be no greater than 32 700.
4. In **form 2**, *var1* and *var2* must be simple variable references (no operators), and cannot be used as host variables (*varname* is the host variable).
5. *varname* can be a simple variable name, or it can include operators such as **varname*. See the description of pointer data types in C and C++ for more information.
6. The precompiler determines the SQLTYPE and SQLLEN of all host variables. If a host variable appears in an SQL statement with an indicator variable, the SQLTYPE is assigned to be the base SQLTYPE plus one for the duration of that statement.
7. The precompiler permits some declarations which are not syntactically valid in C or C++. Refer to your compiler documentation if in doubt about a particular declaration syntax.

**Related concepts:**

- "Null terminated strings in C and C++ embedded SQL applications" on page 105
- "Null or truncation indicator variables and indicator tables in C and C++ embedded SQL applications" on page 104

## Declaration of graphic host variables in C and C++ embedded SQL applications

To handle graphic data in C or C++ applications, use host variables based on either the `wchar_t` C or C++ data type or the `sqldbchar` data type provided by DB2. You can assign these types of host variables to columns of a table that are GRAPHIC, VARGRAPHIC, or DBCLOB. For example, you can update or select DBCS data from GRAPHIC or VARGRAPHIC columns of a table.

There are three valid forms for a graphic host variable:

- Single-graphic form

  Single-graphic host variables have an SQLTYPE of 468/469 that is equivalent to the GRAPHIC(1) SQL data type.
- Null-terminated graphic form

Null-terminated refers to the situation where all the bytes of the last character of the graphic string contain binary zeros ('\0's). They have an SQLTYPE of 400/401.

- VARGRAPHIC structured form

  VARGRAPHIC structured host variables have an SQLTYPE of 464/465 if their length is between 1 and 16 336 bytes. They have an SQLTYPE of 472/473 if their length is between 2 000 and 16 350 bytes.

**Related concepts:**

- "Host structure support in the declare section of C and C++ embedded SQL applications" on page 103
- "Declare section for host variables in C and C++ embedded SQL applications" on page 80
- "Initialization of host variables in C and C++ embedded SQL applications" on page 101
- "Host variable names in C and C++ embedded SQL applications" on page 79
- "Null or truncation indicator variables and indicator tables in C and C++ embedded SQL applications" on page 104
- "wchar_t and sqldbchar data types for graphic data in C and C++ embedded SQL applications" on page 87
- "WCHARTYPE precompiler option for graphic data in C and C++ embedded SQL applications" on page 88

**Related reference:**

- "Declaration of file reference type host variables in C and C++ embedded SQL applications" on page 96
- "Declaration of GRAPHIC type host variables in single-graphic and null-terminated graphic forms in C and C++ embedded SQL applications" on page 92
- "Declaration of VARGRAPHIC type host variables in the structured form in C or C++ embedded SQL applications" on page 91
- "Declaration of large object type host variables in C and C++ embedded SQL applications" on page 93
- "Declaration of large object locator type host variables in C and C++ embedded SQL applications" on page 95

## wchar_t and sqldbchar data types for graphic data in C and C++ embedded SQL applications

While the size and encoding of DB2 graphic data is constant from one platform to another for a particular code page, the size and internal format of the ANSI C or C++ `wchar_t` data type depends on which compiler you use and which platform you are on. The `sqldbchar` data type, however, is defined by DB2 to be two bytes in size, and is intended to be a portable way of manipulating DBCS and UCS-2 data in the same format in which it is stored in the database.

You can define all DB2 C graphic host variable types using either `wchar_t` or `sqldbchar`. You must use `wchar_t` if you build your application using the WCHARTYPE CONVERT precompile option.

**Note:** When specifying the WCHARTYPE CONVERT option on a Windows operating system, you should note that `wchar_t` on Windows operating systems is Unicode. Therefore, if your C or C++ compiler's `wchar_t` is not

Unicode, the `wcstombs()` function call can fail with SQLCODE -1421 (SQLSTATE=22504). If this happens, you can specify the WCHARTYPE NOCONVERT option, and explicitly call the `wcstombs()` and `mbstowcs()` functions from within your program.

If you build your application with the WCHARTYPE NOCONVERT precompile option, you should use `sqldbchar` for maximum portability between different DB2 client and server platforms. You can use `wchar_t` with WCHARTYPE NOCONVERT, but only on platforms where `wchar_t` is defined as two bytes in length.

If you incorrectly use either `wchar_t` or `sqldbchar` in host variable declarations, you will receive an SQLCODE 15 (no SQLSTATE) at precompile time.

**Related concepts:**

- "Japanese and Traditional Chinese EUC and UCS-2 code set considerations" in *Developing SQL and External Routines*
- "WCHARTYPE precompiler option for graphic data in C and C++ embedded SQL applications" on page 88

## WCHARTYPE precompiler option for graphic data in C and C++ embedded SQL applications

Using the WCHARTYPE precompiler option, you can specify which graphic character format you want to use in your C or C++ application. This option provides you with the flexibility to choose between having your graphic data in multi-byte format or in wide-character format. There are two possible values for the WCHARTYPE option:

**CONVERT**

If you select the WCHARTYPE CONVERT option, character codes are converted between the graphic host variable and the database manager. For graphic input host variables, the character code conversion from wide-character format to multi-byte DBCS character format is performed before the data is sent to the database manager, using the ANSI C function `wcstombs()`. For graphic output host variables, the character code conversion from multi-byte DBCS character format to wide-character format is performed before the data received from the database manager is stored in the host variable, using the ANSI C function `mbstowcs()`.

The advantage to using WCHARTYPE CONVERT is that it allows your application to fully exploit the ANSI C mechanisms for dealing with wide-character strings (L-literals, 'wc' string functions, and so on) without having to explicitly convert the data to multi-byte format before communicating with the database manager. The disadvantage is that the implicit conversions may have an impact on the performance of your application at run time, and may increase memory requirements.

If you select WCHARTYPE CONVERT, declare all graphic host variables using `wchar_t` instead of `sqldbchar`.

If you want WCHARTYPE CONVERT behavior, but your application does not need to be precompiled (for example, a CLI application), then define the C preprocessor macro `SQL_WCHART_CONVERT` at compile time. This ensures that certain definitions in the DB2 header files use the data type `wchar_t` instead of `sqldbchar`.

**Note:** The WCHARTYPE CONVERT precompile option is not supported in programs running on the DB2 Windows 3.1 client. For those programs, use the default (WCHARTYPE NOCONVERT).

**NOCONVERT (default)**
: If you choose the WCHARTYPE NOCONVERT option, or do not specify any WCHARTYPE option, no implicit character code conversion occurs between the application and the database manager. Data in a graphic host variable is sent to and received from the database manager as unaltered DBCS characters. This has the advantage of improved performance, but the disadvantage that your application must either refrain from using wide-character data in `wchar_t` host variables, or must explicitly call the `wcstombs()` and `mbstowcs()` functions to convert the data to and from multi-byte format when interfacing with the database manager.

  If you select WCHARTYPE NOCONVERT, declare all graphic host variables using the `sqldbchar` type for maximum portability to other DB2 client/server platforms.

Other guidelines you need to observe are:

- Because `wchar_t` or `sqldbchar` support is used to handle DBCS data, its use requires DBCS or EUC capable hardware and software. This support is only available in the DBCS environment of DB2 Database for Linux, UNIX, and Windows, or for dealing with GRAPHIC data in any application (including single-byte applications) connected to a UCS-2 database.
- Non-DBCS characters, and wide-characters that can be converted to non-DBCS characters, should not be used in graphic strings. *Non-DBCS characters* refers to single-byte characters, and non-double byte characters. Graphic strings are not validated to ensure that their values contain only double-byte character code points. Graphic host variables must contain only DBCS data, or, if WCHARTYPE CONVERT is in effect, wide-character data that converts to DBCS data. You should store mixed double-byte and single-byte data in character host variables. Note that mixed data host variables are unaffected by the setting of the WCHARTYPE option.
- In applications where the WCHARTYPE NOCONVERT precompile option is used, L-literals should not be used in conjunction with graphic host variables, because L-literals are in wide-character format. An L-literal is a C wide-character string literal prefixed by the letter L which has the data type `"array of wchar_t"`. For example, `L"dbcs-string"` is an L-literal.
- In applications where the WCHARTYPE CONVERT precompile option is used, L-literals can be used to initialize `wchar_t` host variables, but cannot be used in SQL statements. Instead of using L-literals, SQL statements should use graphic string constants, which are independent of the WCHARTYPE setting.
- The setting of the WCHARTYPE option affects graphic data passed to and from the database manager using the SQLDA structure as well as host variables. If WCHARTYPE CONVERT is in effect, graphic data received from the application through an SQLDA will be presumed to be in wide-character format, and will be converted to DBCS format via an implicit call to `wcstombs()`. Similarly, graphic output data received by an application will have been converted to wide-character format before being placed in application storage.
- Not-fenced stored procedures must be precompiled with the WCHARTYPE NOCONVERT option. Ordinary fenced stored procedures may be precompiled with either the CONVERT or NOCONVERT options, which will affect the format of graphic data manipulated by SQL statements contained in the stored procedure. In either case, however, any graphic data passed into the stored

procedure through the SQLDA will be in DBCS format. Likewise, data passed out of the stored procedure through the SQLDA must be in DBCS format.

- If an application calls a stored procedure through the Database Application Remote Interface (DARI) interface (the `sqleproc()` API), any graphic data in the input SQLDA must be in DBCS format, or in UCS-2 if connected to a UCS-2 database, regardless of the state of the calling application's WCHARTYPE setting. Likewise, any graphic data in the output SQLDA will be returned in DBCS format, or in UCS-2 if connected to a UCS-2 database, regardless of the WCHARTYPE setting.
- If an application calls a stored procedure through the SQL CALL statement, graphic data conversion will occur on the SQLDA, depending on the calling application's WCHARTYPE setting.
- Graphic data passed to user-defined functions (UDFs) will always be in DBCS format. Likewise, any graphic data returned from a UDF will be assumed to be in DBCS format for DBCS databases, and UCS-2 format for EUC and UCS-2 databases.
- Data stored in DBCLOB files through the use of DBCLOB file reference variables is stored in either DBCS format, or, in the case of UCS-2 databases, in UCS-2 format. Likewise, input data from DBCLOB files is retrieved either in DBCS format, or, in the case of UCS-2 databases, in UCS-2 format.

**Notes:**

1. For DB2 for Windows operating systems, the `WCHARTYPE CONVERT` option is supported for applications compiled with the Microsoft Visual C++ compiler. However, do not use the `CONVERT` option with this compiler if your application inserts data into a DB2 database in a code page that is different from the database code page. DB2 server normally performs a code page conversion in this situation; however, the Microsoft C run-time environment does not handle substitution characters for certain double byte characters. This could result in run time conversion errors.
2. If you precompile C applications using the WCHARTYPE CONVERT option, DB2 validates the applications' graphic data on both input and output as the data is passed through the conversion functions. If you do *not* use the CONVERT option, no conversion of graphic data, and hence no validation occurs. In a mixed CONVERT/NOCONVERT environment, this may cause problems if invalid graphic data is inserted by a NOCONVERT application and then fetched by a CONVERT application. This data fails the conversion with an SQLCODE -1421 (SQLSTATE 22504) on a FETCH in the CONVERT application.

**Related concepts:**

- "Declaration of graphic host variables in C and C++ embedded SQL applications" on page 86
- "Performance of embedded SQL applications" on page 22

**Related reference:**

- "PREPARE statement" in *SQL Reference, Volume 2*
- "Declaration of GRAPHIC type host variables in single-graphic and null-terminated graphic forms in C and C++ embedded SQL applications" on page 92
- "Declaration of large object type host variables in C and C++ embedded SQL applications" on page 93
- "Include files for C and C++ embedded SQL applications" on page 42

## Declaration of VARGRAPHIC type host variables in the structured form in C or C++ embedded SQL applications

Following is the syntax for declaring a graphic host variable using the VARGRAPHIC structured form.

```
►►──┬──────────┬──┬──────────┬──struct──┬─────┬──────────────────────►
    ├─auto─────┤  ├─const────┤          └─tag─┘
    ├─extern───┤  └─volatile─┘
    ├─static───┤
    └─register─┘

                         (1)                           (2)
►─{──short──┬─────┬──var-1──;──────┬─sqldbchar─┬──var-2──[length─────]──;──}──────►
            └─int─┘                └─wchar_t───┘

   ┌─,──────────────────────────────────────────┐
   ▼                                            │
►──┬──────────────────────────────┬─┤ Variable ├─;─┴──────────────────────►◄
   │  ┌────────────────────────┐  │
   │  ▼                        │  │
   └────┬─*─┬──┬──────────┬────┴──┘
        └─&─┘  ├─const────┤
               └─volatile─┘
```

**Variable:**

```
├──variable-name──┬──────────────────────────────┬──────────────────┤
                  └─=──{──value-1──,──value-2──}─┘
```

**Notes:**

1 To determine which of the two graphic types should be used, see the description of the `wchar_t` and `sqldbchar` data types in C and C++.

2 *length* can be any valid constant expression. Its value after evaluation determines if the host variable is VARGRAPHIC (SQLTYPE 464) or LONG VARGRAPHIC (SQLTYPE 472). The value of length must be greater than or equal to 1, and not greater than the maximum length of LONG VARGRAPHIC which is 16 350.

**Graphic declaration (VARGRAPHIC structured form) Considerations:**

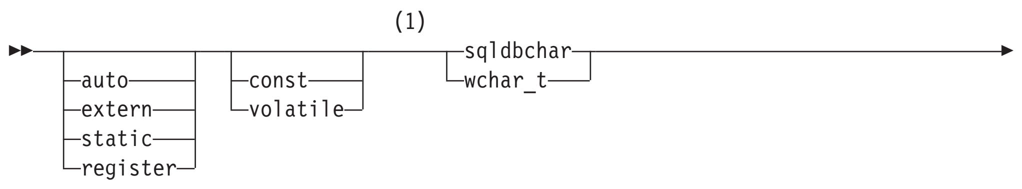
1. *var-1* and *var-2* must be simple variable references (no operators) and cannot be used as host variables.
2. *value-1* and *value-2* are initializers for *var-1* and *var-2*. *value-1* must be an integer and *value-2* should be a wide-character string literal (L-literal) if the WCHARTYPE CONVERT precompiler option is used.
3. The struct *tag* can be used to define other data areas, but itself cannot be used as a host variable.

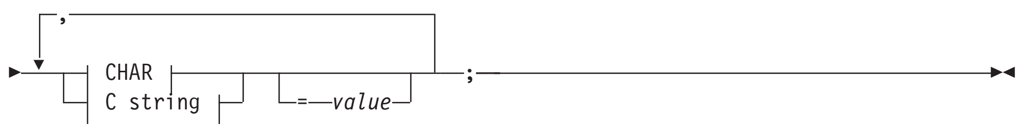
**Related concepts:**

- "wchar_t and sqldbchar data types for graphic data in C and C++ embedded SQL applications" on page 87

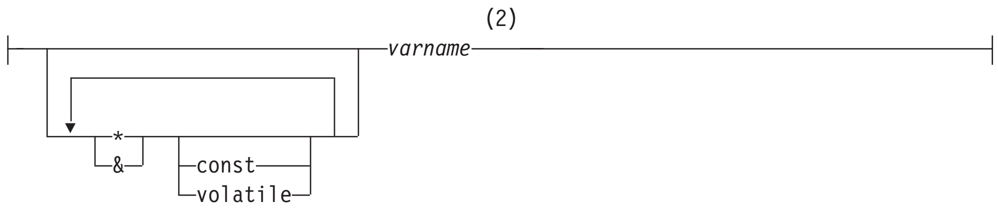
## Declaration of GRAPHIC type host variables in single-graphic and null-terminated graphic forms in C and C++ embedded SQL applications

Following is the syntax for declaring a graphic host variable using the single-graphic form and the null-terminated graphic form.

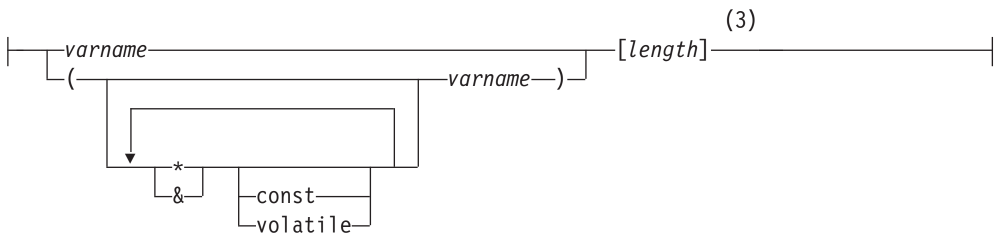




**CHAR**



**C string**



**Notes:**

1 To determine which of the two graphic types should be used, see the description of the `wchar_t` and `sqldbchar` data types in C and C++.

2 GRAPHIC (SQLTYPE 468), length 1

3 Null-terminated graphic string (SQLTYPE 400)

**Graphic host variable considerations:**

1. The single-graphic form declares a fixed-length graphic string host variable of length 1 with SQLTYPE of 468 or 469.
2. *value* is an initializer. A wide-character string literal (L-literal) should be used if the WCHARTYPE CONVERT precompiler option is used.

3. *length* can be any valid constant expression, and its value after evaluation must be greater than or equal to 1, and not greater than the maximum length of VARGRAPHIC, which is 16 336.
4. Null-terminated graphic strings are handled differently, depending on the value of the standards level precompile option setting.

**Related concepts:**
- "Null terminated strings in C and C++ embedded SQL applications" on page 105
- "wchar_t and sqldbchar data types for graphic data in C and C++ embedded SQL applications" on page 87

## Declaration of large object type host variables in C and C++ embedded SQL applications

Following is the syntax for declaring large object (LOB) host variables in C or C++.

```
►►─┬──────────┬─┬──────────┬─SQL TYPE IS─┬────────┬─┬─BLOB───┬───────────────►
   ├─auto─────┤ ├─const────┤             └─XML AS─┘ ├─CLOB───┤
   ├─extern───┤ └─volatile─┘                        └─DBCLOB─┘
   ├─static───┤
   └─register─┘

                   ┌─,──────────────────────────────────────────────────┐
          (1)      ▼                                                    │
►─(length────)──┬──────────────────────────┬─variable-name─┤ LOB data ├─┴──►
                │ ┌──────────────────────┐ │
                │ ▼                      │ │
                └───┬─*─┬─┬──────────┬───┴─┘
                    └─&─┘ ├─const────┤
                          └─volatile─┘

►─;──────────────────────────────────────────────────────────────────────►◄
```

**LOB data**

```
├─┬─────────────────────────────────┬──────────────────────────────────────┤
  ├─={init-len,"init-data"}─────────┤
  ├─=SQL_BLOB_INIT("init-data")─────┤
  ├─=SQL_CLOB_INIT("init-data")─────┤
  └─=SQL_DBCLOB_INIT("init-data")───┘
```

**Notes:**

1 *length* can be any valid constant expression, in which the constant K, M, or G can be used. The value of length after evaluation for BLOB and CLOB must be 1 <= length <= 2 147 483 647. The value of *length* after evaluation for DBCLOB must be 1 <= length <= 1 073 741 823.

**LOB host variable considerations:**
1. The SQL TYPE IS clause is needed to distinguish the three LOB-types from each other so that type checking and function resolution can be carried out for LOB-type host variables that are passed to functions.

2. SQL TYPE IS, BLOB, CLOB, DBCLOB, K, M, G may be in mixed case.
3. The maximum length allowed for the initialization string "*init-data*" is 32 702 bytes, including string delimiters (the same as the existing limit on C and C++ strings within the precompiler).
4. The initialization length, *init-len*, must be a numeric constant (for example, it cannot include K, M, or G).
5. A length for the LOB must be specified; that is, the following declaration is not permitted:
   ```
   SQL TYPE IS BLOB my_blob;
   ```
6. If the LOB is not initialized within the declaration, no initialization will be done within the precompiler-generated code.
7. If a DBCLOB is initialized, it is the user's responsibility to prefix the string with an 'L' (indicating a wide-character string).

   **Note:** Wide-character literals, for example, `L"Hello"`, should only be used in a precompiled program if the WCHARTYPE CONVERT precompile option is selected.
8. The precompiler generates a structure tag which can be used to cast to the host variable's type.

**BLOB example:**

Declaration:

```
static Sql Type is Blob(2M) my_blob=SQL_BLOB_INIT("mydata");
```

Results in the generation of the following structure:

```
static struct my_blob_t {
      sqluint32       length;
      char            data[2097152];
 } my_blob=SQL_BLOB_INIT("mydata");
```

**CLOB example:**

Declaration:

```
 volatile sql type is clob(125m) *var1, var2 = {10, "data5data5"};
```

Results in the generation of the following structure:

```
 volatile struct var1_t {
      sqluint32       length;
      char            data[131072000];
 } * var1, var2 = {10, "data5data5"};
```

**DBCLOB example:**

Declaration:

```
SQL TYPE IS DBCLOB(30000) my_dbclob1;
```

Precompiled with the WCHARTYPE NOCONVERT option, results in the generation of the following structure:

```
struct my_dbclob1_t {
     sqluint32       length;
     sqldbchar       data[30000];
} my_dbclob1;
```

Declaration:

```
SQL TYPE IS DBCLOB(30000) my_dbclob2 = SQL_DBCLOB_INIT(L"mydbdata");
```

Precompiled with the WCHARTYPE CONVERT option, results in the generation of the following structure:

```
struct my_dbclob2_t {
     sqluint32       length;
     wchar_t         data[30000];
} my_dbclob2 = SQL_DBCLOB_INIT(L"mydbdata");
```

**Related concepts:**

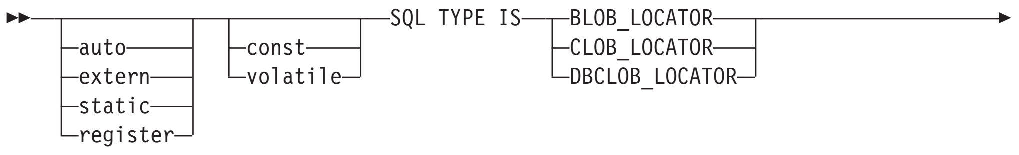
- "Declaration of host variables as pointers in C and C++ embedded SQL applications" on page 97
- "Declare section for host variables in C and C++ embedded SQL applications" on page 80
- "Initialization of host variables in C and C++ embedded SQL applications" on page 101
- "Precompiler generated timestamps" on page 188

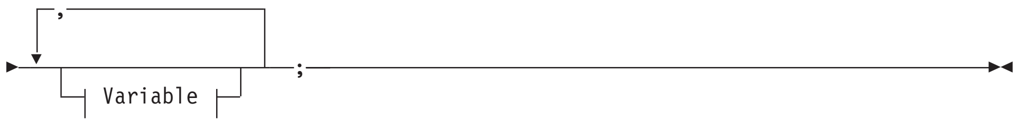
**Related tasks:**

- "Declaring XML host variables in embedded SQL applications" on page 71

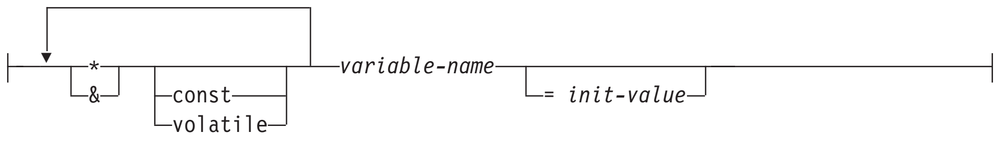
## Declaration of large object locator type host variables in C and C++ embedded SQL applications

Following is the syntax for declaring large object (LOB) locator host variables in C or C++.





**Variable**



**LOB locator host variable considerations:**

1. SQL TYPE IS, BLOB_LOCATOR, CLOB_LOCATOR, DBCLOB_LOCATOR may be in mixed case.
2. *init-value* permits the initialization of pointer and reference locator variables. Other types of initialization will have no meaning.

**CLOB locator example** (other LOB locator type declarations are similar):

Declaration:

```
SQL TYPE IS CLOB_LOCATOR my_locator;
```

Results in the generation of the following declaration:

```
sqluint32 my_locator;
```

**Related concepts:**

- “Declare section for host variables in C and C++ embedded SQL applications” on page 80

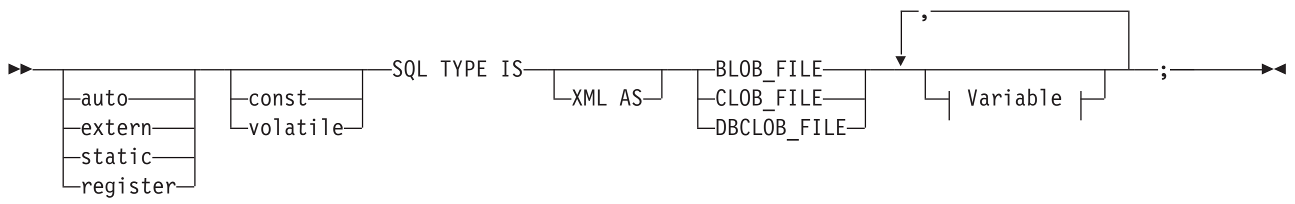
**Related reference:**

- “Declaration of file reference type host variables in C and C++ embedded SQL applications” on page 96

## Declaration of file reference type host variables in C and C++ embedded SQL applications

The following is the syntax for declaring file reference host variables in C or C++.

### Syntax for file reference host variables in C or C++



### Variable

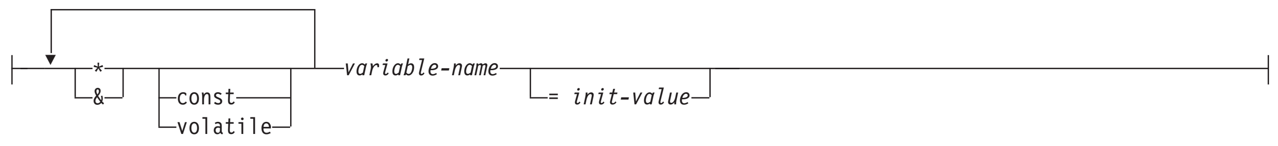


*Figure 2. Syntax Diagram*

**Note:** SQL TYPE IS, BLOB_FILE, CLOB_FILE, DBCLOB_FILE may be in mixed case.

**CLOB file reference example** (other LOB file reference type declarations are similar):

Declaration:

```
static volatile SQL TYPE IS BLOB_FILE my_file;
```

Results in the generation of the following structure:

```
static volatile struct {
     sqluint32       name_length;
     sqluint32       data_length;
     sqluint32       file_options;
              char   name[255];
} my_file;
```

**Note:** The structure above is equivalent to the sqlfile structure located in the sql.h header. See Figure 2 on page 96 to refer to the syntax diagram.

**Related concepts:**

- "Initialization of host variables in C and C++ embedded SQL applications" on page 101

## Declaration of host variables as pointers in C and C++ embedded SQL applications

Host variables can be declared as pointers to specific data types with the following restrictions:

- If a host variable is declared as a pointer, no other host variable can be declared with that same name within the same source file. The following example is not allowed:

  ```
  char mystring[20];
  char (*mystring)[20];
  ```

- Use parentheses when declaring a pointer to a null-terminated character array. In all other cases, parentheses are not allowed. For example:

  ```
  EXEC SQL BEGIN DECLARE SECTION;
    char  (*arr)[10];  /* correct   */
    char  *(arr);      /* incorrect */
    char  *arr[10];    /* incorrect */
  EXEC SQL END DECLARE SECTION;
  ```

  The first declaration is a pointer to a 10-byte character array. This is a valid host variable. The second is an invalid declaration. The parentheses are not allowed in a pointer to a character. The third declaration is an array of pointers. This is not a supported data type.

  The host variable declaration:

  ```
  char *ptr;
  ```

  is accepted, but it does not mean *null-terminated character string of undetermined length*. Instead, it means a *pointer to a fixed-length, single-character host variable*. This might not be what is intended. To define a pointer host variable that can indicate different character strings, use the first declaration form above.

- When pointer host variables are used in SQL statements, they should be prefixed by the same number of asterisks as they were declared with, as in the following example:

  ```
  EXEC SQL BEGIN DECLARE SECTION;
    char (*mychar)[20];    /* Pointer to character array of 20 bytes */
  EXEC SQL END DECLARE SECTION;

  EXEC SQL SELECT column INTO :*mychar FROM table;   /* Correct */
  ```

- Only the asterisk can be used as an operator over a host variable name.
- The maximum length of a host variable name is not affected by the number of asterisks specified, because asterisks are not considered part of the name.
- Whenever using a pointer variable in an SQL statement, you should leave the optimization level precompile option (OPTLEVEL) at the default setting of 0 (no optimization). This means that no SQLDA optimization will be done by the database manager.

**Related concepts:**

- "Host variable names in C and C++ embedded SQL applications" on page 79
- "Null terminated strings in C and C++ embedded SQL applications" on page 105

- "Null or truncation indicator variables and indicator tables in C and C++ embedded SQL applications" on page 104
- "Declaration of fixed-length, null-terminated and variable-length character host variables in C and C++ embedded SQL applications" on page 84

### Declaration of class data members as host variables in C++ embedded SQL applications

You can declare class data members as host variables (but not classes or objects themselves). The following example illustrates the method to use:

```
class STAFF
{
    private:
         EXEC SQL BEGIN DECLARE SECTION;
           char       staff_name[20];
           short int  staff_id;
           double     staff_salary;
         EXEC SQL END DECLARE SECTION;
         short      staff_in_db;
    .
    .
};
```

Data members are only directly accessible in SQL statements through the implicit *this* pointer provided by the C++ compiler in class member functions. You **cannot** explicitly qualify an object instance (such as `SELECT name INTO :my_obj.staff_name ...`) in an SQL statement.

If you directly refer to class data members in SQL statements, the database manager resolves the reference using the *this* pointer. For this reason, you should leave the optimization level precompile option (OPTLEVEL) at the default setting of 0 (no optimization).

The following example shows how you might directly use class data members which you have declared as host variables in an SQL statement.

```
class STAFF
{
⋮
    public:

⋮
       short int hire( void )
       {
         EXEC SQL INSERT INTO staff ( name,id,salary )
           VALUES ( :staff_name, :staff_id, :staff_salary );
         staff_in_db = (sqlca.sqlcode == 0);
         return sqlca.sqlcode;
       }
};
```

In this example, class data members `staff_name`, `staff_id`, and `staff_salary` are used directly in the INSERT statement. Because they have been declared as host variables (see the first example in this section), they are implicitly qualified to the current object with the *this* pointer. In SQL statements, you can also refer to data members that are not accessible through the *this* pointer. You do this by referring to them indirectly using pointer or reference host variables.

The following example shows a new method, *asWellPaidAs* that takes a second object, *otherGuy*. This method references its members indirectly through a local pointer or reference host variable, as you cannot reference its members directly within the SQL statement.

```
short int STAFF::asWellPaidAs( STAFF otherGuy )
{
    EXEC SQL BEGIN DECLARE SECTION;
      short &otherID = otherGuy.staff_id
      double otherSalary;
    EXEC SQL END DECLARE SECTION;
    EXEC SQL SELECT SALARY INTO :otherSalary
      FROM STAFF WHERE id = :otherID;
      if( sqlca.sqlcode == 0 )
         return staff_salary >= otherSalary;
      else
         return 0;
}
```

**Related concepts:**

- "Host variables in C and C++ embedded SQL applications" on page 78

**Related reference:**

- "Data types for procedures, functions, and methods in C and C++ embedded SQL applications" on page 58
- "INSERT statement" in *SQL Reference, Volume 2*

## Scope resolution and class member operators in C and C++ embedded SQL applications

You *cannot* use the C++ scope resolution operator '::', nor the C and C++ member operators '.' or '->' in embedded SQL statements. You can easily accomplish the same thing through use of local pointer or reference variables, which are set outside the SQL statement, to point to the desired scoped variable, then used inside the SQL statement to refer to it. The following example shows the correct method to use:

```
EXEC SQL BEGIN DECLARE SECTION;
  char (& localName)[20] = ::name;
EXEC SQL END DECLARE SECTION;
EXEC SQL
  SELECT name INTO :localName FROM STAFF
  WHERE name = 'Sanders';
```

**Related concepts:**

- "Declaration of host variables as pointers in C and C++ embedded SQL applications" on page 97

**Related reference:**

- "Declaration of file reference type host variables in C and C++ embedded SQL applications" on page 96

## Japanese or Traditional Chinese EUC, and UCS-2 Considerations in C and C++ embedded SQL applications

If your application code page is Japanese or Traditional Chinese EUC, or if your application connects to a UCS-2 database, you can access GRAPHIC columns at a database server by using either the CONVERT or the NOCONVERT option and `wchar_t` or `sqldbchar` graphic host variables, or input/output SQLDAs. In this section, *DBCS format* refers to the UCS-2 encoding scheme for EUC data. Consider the following cases:

- CONVERT option used

  The DB2 client converts graphic data from the wide character format to your application code page, then to UCS-2 before sending the input SQLDA to the database server. Any graphic data is sent to the database server tagged with the UCS-2 code page identifier. Mixed character data is tagged with the application code page identifier. When graphic data is retrieved from a database by a client, it is tagged with the UCS-2 code page identifier. The DB2 client converts the data from UCS-2 to the client application code page, then to the wide character format. If an input SQLDA is used instead of a host variable, you are required to ensure that graphic data is encoded using the wide character format. This data will be converted to UCS-2, then sent to the database server. These conversions will impact performance.
- NOCONVERT option used

  The graphic data is assumed by DB2 to be encoded using UCS-2 and is tagged with the UCS-2 code page, and no conversions are done. DB2 assumes that the graphic host variable is being used simply as a bucket. When the NOCONVERT option is chosen, graphic data retrieved from the database server is passed to the application encoded using UCS-2. Any conversions from the application code page to UCS-2 and from UCS-2 to the application code page are your responsibility. Data tagged as UCS-2 is sent to the database server without any conversions or alterations.

To minimize conversions you can either use the NOCONVERT option and handle the conversions in your application, or not use GRAPHIC columns. For the client environments where `wchar_t` encoding is in two-byte Unicode, for example Windows 2000® or AIX version 5.1 and higher, you can use the NOCONVERT option and work directly with UCS-2. In such cases, your application should handle the difference between big-endian and little-endian architectures. With the NOCONVERT option, DB2 database systems use `sqldbchar`, which is always two-byte big-endian.

Do not assign IBM-eucJP/IBM-eucTW CS0 (7-bit ASCII) and IBM-eucJP CS2 (Katakana) data to graphic host variables either after conversion to UCS-2 (if NOCONVERT is specified) or by conversion to the wide character format (if CONVERT is specified). The reason is that characters in both of these EUC code sets become single-byte when converted from UCS-2 to PC DBCS.

In general, although eucJP and eucTW store GRAPHIC data as UCS-2, the GRAPHIC data in these databases is still non-ASCII eucJP or eucTW data. Specifically, any space padded to such GRAPHIC data is DBCS space (also known as ideographic space in UCS-2, U+3000). For a UCS-2 database, however, GRAPHIC data can contain any UCS-2 character, and space padding is done with UCS-2 space, U+0020. Keep this difference in mind when you code applications to retrieve UCS-2 data from a UCS-2 database versus UCS-2 data from eucJP and eucTW databases.

**Related concepts:**

- "Japanese and Traditional Chinese EUC and UCS-2 code set considerations" in *Developing SQL and External Routines*
- "Declaration of graphic host variables in C and C++ embedded SQL applications" on page 86

- "wchar_t and sqldbchar data types for graphic data in C and C++ embedded SQL applications" on page 87
- "Performance of embedded SQL applications" on page 22

**Related tasks:**
- "Transferring data in a dynamically executed SQL program using an SQLDA structure" on page 148

## Binary storage of variable values using the FOR BIT DATA clause in C and C++ embedded SQL applications

The standard C or C++ string type 460 should not be used for columns designated FOR BIT DATA. The database manager truncates this data type when a null character is encountered. Use either the VARCHAR (SQL type 448) or CLOB (SQL type 408) structures.

**Related concepts:**
- "Example: SQL declare section template for C and C++ embedded SQL applications" on page 81

**Related reference:**
- "Supported SQL data types in C and C++ embedded SQL applications" on page 53

## Initialization of host variables in C and C++ embedded SQL applications

In C and C++ declare sections, you can declare and initialize multiple variables on a single line. However, variables must be initialized using the ″=″ symbol and not by using parentheses. The following example shows the correct and incorrect methods of initialization in a declare section:

```
EXEC SQL BEGIN DECLARE SECTION;
  short my_short_2 = 5;       /* correct   */
  short my_short_1(5);        /* incorrect */
EXEC SQL END DECLARE SECTION;
```

**Related concepts:**
- "Host variables in C and C++ embedded SQL applications" on page 78
- "Declare section for host variables in C and C++ embedded SQL applications" on page 80

## Macro expansion and the DECLARE SECTION of C and C++ embedded SQL applications

The C or C++ precompiler cannot directly process any C macro used in a declaration within a declare section. Instead, you must first preprocess the source file with an external C preprocessor. To do this, specify the exact command for invoking a C preprocessor to the precompiler through the PREPROCESSOR option.

When you specify the PREPROCESSOR option, the precompiler first processes all the SQL INCLUDE statements by incorporating the contents of all the files referred to in the SQL INCLUDE statement into the source file. The precompiler then invokes the external C preprocessor using the command you specify with the modified source file as input. The preprocessed file, which the precompiler always expects to have an extension of `.i`, is used as the new source file for the rest of the precompiling process.

Any #line macro generated by the precompiler no longer references the original source file, but instead references the preprocessed file. To relate any compiler errors back to the original source file, retain comments in the preprocessed file. This helps you to locate various sections of the original source files, including the header files. The option to retain comments is commonly available in C preprocessors, and you can include the option in the command you specify through the PREPROCESSOR option. You should not have the C preprocessor output any #line macros itself, as they may be incorrectly mixed with ones generated by the precompiler.

**Notes on using macro expansion:**

1. The command you specify through the PREPROCESSOR option should include all the desired options, but not the name of the input file. For example, for IBM C on AIX you can use the option:
   ```
   xlC -P -DMYMACRO=1
   ```
2. The precompiler expects the command to generate a preprocessed file with a `.i` extension. However, you cannot use redirection to generate the preprocessed file. For example, you **cannot** use the following option to generate a preprocessed file:
   ```
   xlC -E > x.i
   ```
3. Any errors the external C preprocessor encounters are reported in a file with a name corresponding to the original source file, but with a `.err` extension.

For example, you can use macro expansion in your source code as follows:

```
#define SIZE 3

EXEC SQL BEGIN DECLARE SECTION;
  char a[SIZE+1];
  char b[(SIZE+1)*3];
  struct
  {
    short length;
    char  data[SIZE*6];
  } m;
  SQL TYPE IS BLOB(SIZE+1) x;
  SQL TYPE IS CLOB((SIZE+2)*3) y;
  SQL TYPE IS DBCLOB(SIZE*2K) z;
EXEC SQL END DECLARE SECTION;
```

The previous declarations resolve to the following after you use the PREPROCESSOR option:

```
EXEC SQL BEGIN DECLARE SECTION;
  char a[4];
  char b[12];
  struct
  {
    short length;
    char  data[18];
  } m;
  SQL TYPE IS BLOB(4) x;
  SQL TYPE IS CLOB(15) y;
  SQL TYPE IS DBCLOB(6144) z;
EXEC SQL END DECLARE SECTION;
```

**Related concepts:**

- “Error-checking utilities” on page 202

**Related reference:**

- “Include files for C and C++ embedded SQL applications” on page 42

- "INCLUDE statement" in *SQL Reference, Volume 2*

## Host structure support in the declare section of C and C++ embedded SQL applications

With host structure support, the C or C++ precompiler allows host variables to be grouped into a single host structure. This feature provides a shorthand for referencing that same set of host variables in an SQL statement. For example, the following host structure can be used to access some of the columns in the STAFF table of the SAMPLE database:

```
struct tag
   {
     short id;
     struct
     {
       short length;
       char  data[10];
     } name;
     struct
     {
       short   years;
       double salary;
     } info;
   } staff_record;
```

The fields of a host structure can be any of the valid host variable types. Valid types include all numeric, character, and large object types. Nested host structures are also supported up to 25 levels. In the example above, the field `info` is a sub-structure, whereas the field `name` is not, as it represents a VARCHAR field. The same principle applies to LONG VARCHAR, VARGRAPHIC and LONG VARGRAPHIC. Pointer to host structure is also supported.

There are two ways to reference the host variables grouped in a host structure in an SQL statement:

- The host structure name can be referenced in an SQL statement.

  ```
  EXEC SQL SELECT id, name, years, salary
       INTO :staff_record
       FROM staff
       WHERE id = 10;
  ```

  The precompiler converts the reference to `staff_record` into a list, separated by commas, of all the fields declared within the host structure. Each field is qualified with the host structure names of all levels to prevent naming conflicts with other host variables or fields. This is equivalent to the following method.
- Fully qualified host variable names can be referenced in an SQL statement.

  ```
  EXEC SQL SELECT id, name, years, salary
       INTO :staff_record.id, :staff_record.name,
            :staff_record.info.years, :staff_record.info.salary
       FROM staff
       WHERE id = 10;
  ```

  References to field names must be fully qualified, even if there are no other host variables with the same name. Qualified sub-structures can also be referenced. In the example above, `:staff_record.info` can be used to replace `:staff_record.info.years, :staff_record.info.salary`.

Because a reference to a host structure (first example) is equivalent to a comma-separated list of its fields, there are instances where this type of reference may lead to an error. For example:

```
EXEC SQL DELETE FROM :staff_record;
```

Here, the DELETE statement expects a single character-based host variable. By giving a host structure instead, the statement results in a precompile-time error:

```
SQL0087N  Host variable "staff_record" is a structure used where structure
references are not permitted.
```

Other uses of host structures, which can cause an SQL0087N error to occur, include PREPARE, EXECUTE IMMEDIATE, CALL, indicator variables and SQLDA references. Host structures with exactly one field are permitted in such situations, as are references to individual fields (second example).

**Related concepts:**

- "Null or truncation indicator variables and indicator tables in C and C++ embedded SQL applications" on page 104
- "Declaration of graphic host variables in C and C++ embedded SQL applications" on page 86

**Related reference:**

- "Declaration of large object type host variables in C and C++ embedded SQL applications" on page 93
- "Declaration of numeric host variables in C and C++ embedded SQL applications" on page 83

## Null or truncation indicator variables and indicator tables in C and C++ embedded SQL applications

For each host variable that can be receive null values, declare **indicator variables** as a `short` data type.

An **indicator table** is a collection of indicator variables to be used with a host structure. It must be declared as an array of short integers. For example:

```
short ind_tab[10];
```

The example above declares an indicator table with 10 elements. The following shows the way it can be used in an SQL statement:

```
EXEC SQL SELECT id, name, years, salary
     INTO :staff_record INDICATOR :ind_tab
     FROM staff
     WHERE id = 10;
```

The following lists each host structure field with its corresponding indicator variable in the table:

| | |
|---|---|
| **staff_record.id** | ind_tab[0] |
| **staff_record.name** | ind_tab[1] |
| **staff_record.info.years** | ind_tab[2] |
| **staff_record.info.salary** | ind_tab[3] |

**Note:** An indicator table element, for example `ind_tab[1]`, cannot be referenced individually in an SQL statement. The keyword INDICATOR is optional. The number of structure fields and indicators do not have to match; any extra indicators are unused, as are extra fields that do not have indicators assigned to them.

A scalar indicator variable can also be used in the place of an indicator table to provide an indicator for the first field of the host structure. This is equivalent to having an indicator table with only one element. For example:

```
short scalar_ind;

EXEC SQL SELECT id, name, years, salary
          INTO :staff_record INDICATOR :scalar_ind
          FROM staff
          WHERE id = 10;
```

If an indicator table is specified along with a host variable instead of a host structure, only the first element of the indicator table, for example `ind_tab[0]`, will be used:

```
EXEC SQL SELECT id
          INTO :staff_record.id INDICATOR :ind_tab
          FROM staff
          WHERE id = 10;
```

If an array of short integers is declared within a host structure:

```
struct tag
{
  short i[2];
} test_record;
```

The array will be expanded into its elements when `test_record` is referenced in an SQL statement making `:test_record` equivalent to `:test_record.i[0]`, `:test_record.i[1]`.

**Related concepts:**

- "Host structure support in the declare section of C and C++ embedded SQL applications" on page 103
- "Host variables in C and C++ embedded SQL applications" on page 78
- "Declare section for host variables in C and C++ embedded SQL applications" on page 80
- "Initialization of host variables in C and C++ embedded SQL applications" on page 101

## Null terminated strings in C and C++ embedded SQL applications

C and C++ null-terminated strings have their own SQLTYPE (460/461 for character and 468/469 for graphic).

C and C++ null-terminated strings are handled differently, depending on the value of the LANGLEVEL precompiler option. If a host variable of one of these SQLTYPE values and declared length *n* is specified within an SQL statement, and the number of bytes (for character types) or double-byte characters (for graphic types) of data is *k*, then:

- If the LANGLEVEL option on the PREP command is SAA1 (the default):

  **For Output:**

| **If...** | **Then...** |
|---|---|
| *k* > *n* | *n* characters are moved to the target host variable, SQLWARN1 is set to 'W', and SQLCODE 0 (SQLSTATE 01004). No null-terminator is placed in the string. If an |

| | |
|---|---|
| | indicator variable was specified with the host variable, the value of the indicator variable is set to *k*. |
| *k* = *n* | *k* characters are moved to the target host variable, SQLWARN1 is set to 'N', and SQLCODE 0 (SQLSTATE 01004). No null-terminator is placed in the string. If an indicator variable was specified with the host variable, the value of the indicator variable is set to 0. |
| *k* < *n* | *k* characters are moved to the target host variable and a null character is placed in character *k + 1*. If an indicator variable was specified with the host variable, the value of the indicator variable is set to 0. |

**For input:** When the database manager encounters an input host variable of one of these SQLTYPE values that does not end with a null-terminator, it will assume that character *n+1* will contain the null-terminator character.

- If the LANGLEVEL option on the PREP command is MIA:

  **For output:**

  | **If...** | **Then...** |
  |---|---|
  | *k* >= *n* | *n - 1* characters are moved to the target host variable, SQLWARN1 is set to 'W', and SQLCODE 0 (SQLSTATE 01501). The *n*th character is set to the null-terminator. If an indicator variable was specified with the host variable, the value of the indicator variable is set to *k*. |
  | *k + 1* = *n* | *k* characters are moved to the target host variable, and the null-terminator is placed in character *n*. If an indicator variable was specified with the host variable, the value of the indicator variable is set to 0. |
  | *k + 1* < *n* | *k* characters are moved to the target host variable, *n - k -1* blanks are appended on the right starting at character *k + 1*, then the null-terminator is placed in character *n*. If an indicator variable was specified with the host variable, the value of the indicator variable is set to 0. |

  **For input:** When the database manager encounters an input host variable of one of these SQLTYPE values that does not end with a null character, SQLCODE -302 (SQLSTATE 22501) is returned.

When specified in any other SQL context, a host variable of SQLTYPE 460 with length *n* is treated as a VARCHAR data type with length *n*, as defined above. When specified in any other SQL context, a host variable of SQLTYPE 468 with length *n* is treated as a VARGRAPHIC data type with length *n*, as defined above.

**Related concepts:**

- "Embedded SQL statements in C and C++ applications" on page 5

**Related tasks:**
- "Declaring the SQLCA for Error Handling" on page 50

**Related reference:**
- "PREPARE statement" in *SQL Reference, Volume 2*

## Host variables in COBOL

### Host variables in COBOL

Host variables are COBOL language variables that are referenced within SQL statements. They allow an application to exchange data with the database manager. After the application is precompiled, host variables are used by the compiler as any other COBOL variable. Follow the rules described in the following sections when naming, declaring, and using host variables.

**Related concepts:**
- "Declare section for host variables in COBOL embedded SQL applications" on page 108
- "Example: SQL declare section template for COBOL embedded SQL applications" on page 108
- "Host variable names in COBOL" on page 107

### Host variable names in COBOL

The SQL precompiler identifies host variables by their declared name. The following rules apply:
- Specify variable names up to 255 characters in length.
- Begin host variable names with prefixes other than `SQL`, `sql`, `DB2`, or `db2`, which are reserved for system use.
- FILLER items using the declaration syntaxes described below are permitted in group host variable declarations, and will be ignored by the precompiler. However, if you use FILLER more than once within an SQL DECLARE section, the precompiler fails. You can not include FILLER items in VARCHAR, LONG VARCHAR, VARGRAPHIC or LONG VARGRAPHIC declarations.
- You can use hyphens in host variable names.

  SQL interprets a hyphen enclosed by spaces as a subtraction operator. Use hyphens without spaces in host variable names.
- The REDEFINES clause is permitted in host variable declarations.
- Level-88 declarations are permitted in the host variable declare section, but are ignored.

**Related concepts:**
- "Declare section for host variables in COBOL embedded SQL applications" on page 108

**Related reference:**
- "Declaration of file reference type host variables in COBOL embedded SQL applications" on page 115

- "Declaration of fixed length and variable length character host variables in COBOL embedded SQL applications" on page 111
- "Declaration of fixed length and variable length graphic host variables in COBOL embedded SQL applications" on page 113
- "Declaration of large object type host variables in COBOL embedded SQL applications" on page 114
- "Declaration of large object locator type host variables in COBOL embedded SQL applications" on page 115
- "Declaration of numeric host variables in COBOL embedded SQL applications" on page 110

## Declare section for host variables in COBOL embedded SQL applications

An SQL declare section must be used to identify host variable declarations. This section alerts the precompiler to any host variables that can be referenced in subsequent SQL statements. For example:

```
    EXEC SQL BEGIN DECLARE SECTION END-EXEC.
77 dept              pic s9(4) comp-5.
01 userid            pic x(8).
01 passwd.
    EXEC SQL END DECLARE SECTION END-EXEC.
```

The COBOL precompiler only recognizes a subset of valid COBOL declarations.

**Related tasks:**

- "Declaring structured type host variables" in *Developing SQL and External Routines*

**Related reference:**

- "Declaration of file reference type host variables in COBOL embedded SQL applications" on page 115
- "Declaration of fixed length and variable length character host variables in COBOL embedded SQL applications" on page 111
- "Declaration of fixed length and variable length graphic host variables in COBOL embedded SQL applications" on page 113
- "Declaration of large object type host variables in COBOL embedded SQL applications" on page 114
- "Declaration of large object locator type host variables in COBOL embedded SQL applications" on page 115
- "Declaration of numeric host variables in COBOL embedded SQL applications" on page 110

## Example: SQL declare section template for COBOL embedded SQL applications

The following is a sample SQL declare section with a host variable declared for each supported SQL data type.

```
EXEC SQL BEGIN DECLARE SECTION END-EXEC.
*
  01 age        PIC S9(4) COMP-5.          /* SQL type  500 */
  01 divis      PIC S9(9) COMP-5.          /* SQL type  496 */
  01 salary     PIC S9(6)V9(3) COMP-3.     /* SQL type  484 */
  01 bonus      USAGE IS COMP-1.           /* SQL type  480 */
  01 wage       USAGE IS COMP-2.           /* SQL type  480 */
```

```
      01 nm        PIC X(5).                  /* SQL type  452 */
      01 varchar.
         49 leng    PIC S9(4) COMP-5.          /* SQL type  448 */
         49 strg    PIC X(14).                 /* SQL type  448 */
      01 longvchar.
         49 len     PIC S9(4) COMP-5.          /* SQL type  456 */
         49 str     PIC X(6027).               /* SQL type  456 */
      01 MY-CLOB USAGE IS SQL TYPE IS CLOB(1M).                /* SQL type  408 */
      01 MY-CLOB-LOCATOR USAGE IS SQL TYPE IS CLOB-LOCATOR.    /* SQL type  964 */
      01 MY-CLOB-FILE USAGE IS SQL TYPE IS CLOB-FILE.          /* SQL type  920 */
      01 MY-BLOB USAGE IS SQL TYPE IS BLOB(1M).                /* SQL type  404 */
      01 MY-BLOB-LOCATOR USAGE IS SQL TYPE IS BLOB-LOCATOR.    /* SQL type  960 */
      01 MY-BLOB-FILE USAGE IS SQL TYPE IS BLOB-FILE.          /* SQL type  916 */
      01 MY-DBCLOB USAGE IS SQL TYPE IS DBCLOB(1M).            /* SQL type  412 */
      01 MY-DBCLOB-LOCATOR USAGE IS SQL TYPE IS DBCLOB-LOCATOR. /* SQL type  968 */
      01 MY-DBCLOB-FILE USAGE IS SQL TYPE IS DBCLOB-FILE.      /* SQL type  924 */
      01 MY-PICTURE PIC G(16000) USAGE IS DISPLAY-1.           /* SQL type  464 */
      01 dt        PIC X(10).                 /* SQL type  384 */
      01 tm        PIC X(8).                  /* SQL type  388 */
      01 tmstmp    PIC X(26).                 /* SQL type  392 */
      01 wage-ind  PIC S9(4) COMP-5.          /* SQL type  464 */
    *
    EXEC SQL END DECLARE SECTION END-EXEC.
```

**Related concepts:**

- "Declare section for host variables in COBOL embedded SQL applications" on page 108

**Related reference:**

- "Supported SQL data types in COBOL embedded SQL applications" on page 60

## BINARY/COMP-4 data types in COBOL embedded SQL applications

The DB2 COBOL precompiler supports the use of BINARY, COMP, and COMP-4 data types wherever integer host variables and indicators are permitted, as long as the target COBOL compiler views (or can be made to view) the BINARY, COMP, or COMP-4 data types as equivalent to the COMP-5 data type. In the examples provided, such host variables and indicators are shown with the type COMP-5. Target compilers supported by DB2 that treat COMP, COMP-4, BINARY COMP and COMP-5 as equivalent are:

- IBM COBOL Set for AIX
- Micro Focus COBOL for AIX

**Related concepts:**

- "Null-indicator variables and null or truncation indicator variable tables in COBOL embedded SQL applications" on page 119
- "Host variable names in COBOL" on page 107
- "Declare section for host variables in COBOL embedded SQL applications" on page 108
- "Host variables in COBOL" on page 107

## SQLSTATE and SQLCODE Variables in COBOL embedded SQL application

When using the LANGLEVEL precompile option with a value of SQL92E, the following two declarations may be included as host variables:

```
EXEC SQL BEGIN DECLARE SECTION END-EXEC.
01 SQLSTATE PIC X(5).
01 SQLCODE  PIC S9(9) USAGE COMP.
.
.
.
EXEC SQL END DECLARE SECTION END-EXEC.
```

If neither of these is specified, the SQLCODE declaration is assumed during the precompile step. The SQLCODE and SQLSTATE variables can be declared using level 01 (as shown above) or level 77. Note that when using this option, the INCLUDE SQLCA statement should not be specified.

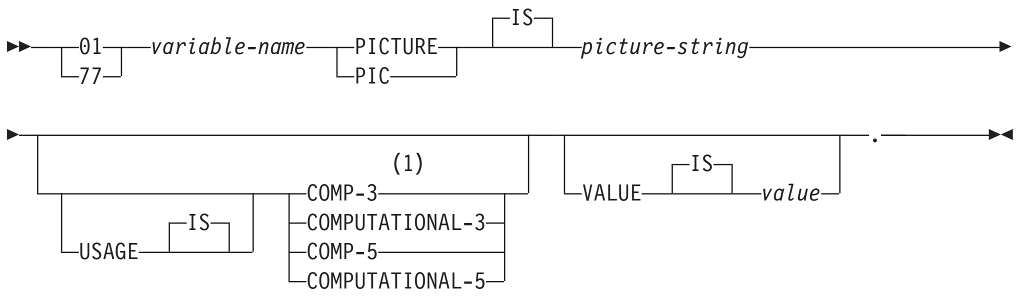
For applications made up of multiple source files, the SQLCODE and SQLSTATE declarations may be included in each source file as shown above.

**Related concepts:**

- "Declare section for host variables in COBOL embedded SQL applications" on page 108
- "Embedded SQL statements in COBOL applications" on page 6
- "Host variables in COBOL" on page 107

## Declaration of numeric host variables in COBOL embedded SQL applications

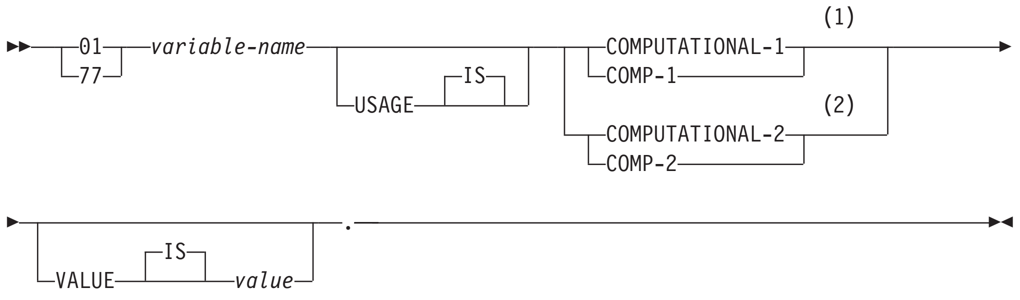
Following is the syntax for numeric host variables.



**Notes:**

1 An alternative for COMP-3 is PACKED-DECIMAL.

**Floating point**



**Notes:**

1 REAL (SQLTYPE 480), Length 4

2 DOUBLE (SQLTYPE 480), Length 8

**Numeric host variable considerations:**

1. *Picture-string* must have one of the following forms:
   - S9(m)V9(n)
   - S9(m)V
   - S9(m)
2. Nines may be expanded (for example., ″S999″ instead of S9(3)″)
3. *m* and *n* must be positive integers.

**Related concepts:**

- "Declare section for host variables in COBOL embedded SQL applications" on page 108
- "Host structure support in the declare section of COBOL embedded SQL applications" on page 117
- "Host variable names in COBOL" on page 107

## Declaration of fixed length and variable length character host variables in COBOL embedded SQL applications

Following is the syntax for character host variables.

**Fixed Length**

```
                                                 ┌─IS─┐
►►──┬─01─┬──variable-name──┬─PICTURE─┬──┴────┴──picture-string──────────────►
    └─77─┘                 └─PIC─────┘

►──┬───────────────────────────┬─.──────────────────────────────────────────►◄
   │         ┌─IS─┐            │
   └─VALUE───┴────┴───value────┘
```

**Variable length**

```
►►──01──variable-name──.────────────────────────────────────────────────────►◄


                                        ┌─IS─┐
►►──49──identifier-1──┬─PICTURE─┬──┴────┴──S9(4)─────────────────────────────►
                      └─PIC─────┘

►──┬────────────────────────────────────────┬──┬───────────────────────────┬─.──►◄
   │                   ┌─COMP-5───────────┐ │  │         ┌─IS─┐            │
   │                   └─COMPUTATIONAL-5──┘ │  └─VALUE───┴────┴───value────┘
   │          ┌─IS─┐   │                    │
   └─USAGE────┴────┴───┘                    


                                        ┌─IS─┐
►►──49──identifier-2──┬─PICTURE─┬──┴────┴──picture-string────────────────────►
                      └─PIC─────┘
```

```
►──┬─────────────────────┬──.──────────────────────────────────────────►◄
   │          ┌─IS─┐     │
   └─VALUE────┴────┴─value─┘
```

**Character host variable consideration:**

1. *Picture-string* must have the form *X(m)*. Alternatively, X's may be expanded (for example, ″XXX″ instead of ″X(3)″).
2. *m* is from 1 to 254 for fixed-length strings.
3. *m* is from 1 to 32 700 for variable-length strings.
4. If *m* is greater than 32 672, the host variable will be treated as a LONG VARCHAR string, and its use may be restricted.
5. Use X and 9 as the picture characters in any PICTURE clause. Other characters are not allowed.
6. Variable-length strings consist of a length item and a value item. You can use acceptable COBOL names for the length item and the string item. However, refer to the variable-length string by the collective name in SQL statements.
7. In a CONNECT statement, such as shown below, COBOL character string host variables `dbname` and `userid` will have any trailing blanks removed before processing:

   ```
   EXEC SQL CONNECT TO :dbname USER :userid USING :p-word
   END-EXEC.
   ```

   However, because blanks can be significant in passwords, the `p-word` host variable should be declared as a VARCHAR data item, so that your application can explicitly indicate the significant password length for the CONNECT statement as follows:

   ```
   EXEC SQL BEGIN DECLARE SECTION END-EXEC.
   01 dbname PIC X(8).
   01 userid PIC X(8).
   01 p-word.
       49 L PIC S9(4) COMP-5.
       49 D PIC X(18).
   EXEC SQL END DECLARE SECTION END-EXEC.
   PROCEDURE DIVISION.
       MOVE "sample" TO dbname.
       MOVE "userid" TO userid.
       MOVE "password" TO D OF p-word.
       MOVE 8          TO L of p-word.
   EXEC SQL CONNECT TO :dbname USER :userid USING :p-word
   END-EXEC.
   ```

**Related concepts:**

- “Connecting to DB2 databases in embedded SQL applications” on page 52
- “Host variable names in COBOL” on page 107
- “Declare section for host variables in COBOL embedded SQL applications” on page 108

**Related reference:**

- “CONNECT (Type 1) statement” in *SQL Reference, Volume 2*

## Declaration of fixed length and variable length graphic host variables in COBOL embedded SQL applications

Following is the syntax for graphic host variables.

**Fixed Length**

```
                                                 ┌─IS─┐
►►──┬─01─┬──variable-name──┬─PICTURE─┬──────┴────┴──picture-string──USAGE──────────►
    └─77─┘                 └─PIC─────┘

   ┌─IS─┐
►──┴────┴──DISPLAY-1──┬─────────────────────────────┬─.──────────────────────────►◄
                      │          ┌─IS─┐             │
                      └─VALUE────┴────┴─────value───┘
```

**Variable Length**

```
►►──01──variable-name──.─────────────────────────────────────────────────────────►◄
```

```
                                          ┌─IS─┐
►►──49──identifier-1──┬─PICTURE─┬─────────┴────┴──S9(4)──────────────────────────►
                      └─PIC─────┘

►──┬──────────────────────────────────────────────────┬──┬───────────────────────────┬─.──►◄
   └─┬───────────────────┬─┬─COMP-5───────────────┬─┘  │         ┌─IS─┐            │
     │         ┌─IS─┐    │ └─COMPUTATIONAL-5──────┘    └─VALUE───┴────┴────value───┘
     └─USAGE───┴────┴────┘
```

```
                                          ┌─IS─┐
►►──49──identifier-2──┬─PICTURE─┬─────────┴────┴──picture-string──USAGE──────────►
                      └─PIC─────┘

   ┌─IS─┐
►──┴────┴──DISPLAY-1──┬─────────────────────────────┬─.──────────────────────────►◄
                      │          ┌─IS─┐             │
                      └─VALUE────┴────┴─────value───┘
```

**Graphic Host Variable Considerations:**

1. *Picture-string* must have the form *G(m)*. Alternatively, G's may be expanded (for example, "GGG" instead of "G(3)").
2. *m* is from 1 to 127 for fixed-length strings.
3. *m* is from 1 to 16 350 for variable-length strings.
4. If *m* is greater than 16 336, the host variable will be treated as a LONG VARGRAPHIC string, and its use may be restricted.

**Related concepts:**

- "Example: SQL declare section template for COBOL embedded SQL applications" on page 108

**Related reference:**

• "Supported SQL data types in COBOL embedded SQL applications" on page 60

## Declaration of large object type host variables in COBOL embedded SQL applications

Following is the syntax for declaring large object (LOB) host variables in COBOL.

```
►►──01──variable-name──┬──────────────┬──SQL TYPE IS──┬─BLOB───┬──────────────►
                       └─USAGE─┬────┬─┘               ├─CLOB───┤
                               └─IS─┘                 └─DBCLOB─┘

►─(──length──┬───┬──)──.──────────────────────────────────────────────────────►◄
             ├─K─┤
             ├─M─┤
             └─G─┘
```

**LOB host variable considerations:**

1. For BLOB and CLOB 1 <= lob-length <= 2 147 483 647.
2. For DBCLOB 1 <= lob-length <= 1 073 741 823.
3. SQL TYPE IS, BLOB, CLOB, DBCLOB, K, M, G can be in either uppercase, lowercase, or mixed.
4. Initialization within the LOB declaration is not permitted.
5. The host variable name prefixes LENGTH and DATA in the precompiler generated code.

**BLOB example**:

Declaring:

```
01 MY-BLOB USAGE IS SQL TYPE IS BLOB(2M).
```

Results in the generation of the following structure:

```
01 MY-BLOB.
   49 MY-BLOB-LENGTH PIC S9(9) COMP-5.
   49 MY-BLOB-DATA PIC X(2097152).
```

**CLOB example**:

Declaring:

```
01 MY-CLOB USAGE IS SQL TYPE IS CLOB(125M).
```

Results in the generation of the following structure:

```
01 MY-CLOB.
   49 MY-CLOB-LENGTH PIC S9(9) COMP-5.
   49 MY-CLOB-DATA PIC X(131072000).
```

**DBCLOB example**:

Declaring:

```
01 MY-DBCLOB USAGE IS SQL TYPE IS DBCLOB(30000).
```

Results in the generation of the following structure:

```
01 MY-DBCLOB.
   49 MY-DBCLOB-LENGTH PIC S9(9) COMP-5.
   49 MY-DBCLOB-DATA PIC G(30000) DISPLAY-1.
```

**Related concepts:**

- "Declare section for host variables in COBOL embedded SQL applications" on page 108
- "Host structure support in the declare section of COBOL embedded SQL applications" on page 117
- "Host variable names in COBOL" on page 107

## Declaration of large object locator type host variables in COBOL embedded SQL applications

Following is the syntax for declaring large object (LOB) locator host variables in COBOL.

```
►►──01──variable-name──┬──────────────┬──SQL TYPE IS──┬─BLOB-LOCATOR───┬──.──────►◄
                       └─USAGE─┬────┬─┘               ├─CLOB-LOCATOR───┤
                               └─IS─┘                 └─DBCLOB-LOCATOR─┘
```

**LOB locator host variable considerations:**

1. SQL TYPE IS, BLOB-LOCATOR, CLOB-LOCATOR, DBCLOB-LOCATOR can be either uppercase, lowercase, or mixed.
2. Initialization of locators is not permitted.

**BLOB locator example** (other LOB locator types are similar):

Declaring:

```
01 MY-LOCATOR USAGE SQL TYPE IS BLOB-LOCATOR.
```

Results in the generation of the following declaration:

```
01 MY-LOCATOR PIC S9(9) COMP-5.
```

**Related concepts:**

- "Example: SQL declare section template for COBOL embedded SQL applications" on page 108
- "Declare section for host variables in COBOL embedded SQL applications" on page 108

## Declaration of file reference type host variables in COBOL embedded SQL applications

Following is the syntax for declaring file reference host variables in COBOL.

```
►►──01──variable-name──┬──────────────┬──SQL TYPE IS──┬─BLOB-FILE───┬──.──────►◄
                       └─USAGE─┬────┬─┘               ├─CLOB-FILE───┤
                               └─IS─┘                 └─DBCLOB-FILE─┘
```

- SQL TYPE IS, BLOB-FILE, CLOB-FILE, DBCLOB-FILE can be either uppercase, lowercase, or mixed.

**BLOB file reference example** (other LOB types are similar):

Declaring:

```
01 MY-FILE USAGE IS SQL TYPE IS BLOB-FILE.
```

Results in the generation of the following declaration:

```
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-OPTIONS PIC S9(9) COMP-5.
   49 MY-FILE-NAME PIC X(255).
```

**Related concepts:**

- "Host variable names in COBOL" on page 107
- "Example: SQL declare section template for COBOL embedded SQL applications" on page 108

## Grouping data items using REDEFINES in COBOL embedded SQL applications

You can use the REDEFINES clause when declaring host variables. If you declare a member of a group data item with the REDEFINES clause, and that group data item is referred to as a whole in an SQL statement, any subordinate items containing the REDEFINES clause are not expanded. For example:

```
01 foo.
 10 a pic s9(4) comp-5.
 10 a1 redefines a pic x(2).
 10 b pic x(10).
```

Referring to `foo` in an SQL statement as follows:

```
... INTO :foo ...
```

The above statement is equivalent to:

```
... INTO :foo.a, :foo.b ...
```

That is, the subordinate item `a1` that is declared with the REDEFINES clause, is not automatically expanded out in such situations. If `a1` is unambiguous, you can explicitly refer to a subordinate with a REDEFINES clause in an SQL statement, as follows:

```
... INTO :foo.a1 ...
```

or

```
... INTO :a1 ...
```

**Related concepts:**

- "Embedded SQL statements in COBOL applications" on page 6

## Japanese or Traditional Chinese EUC, and UCS-2 considerations for COBOL embedded SQL applications

Any graphic data sent from your application running under an eucJp or eucTW code set, or connected to a UCS-2 database, is tagged with the UCS-2 code page identifier. Your application must convert a graphic-character string to UCS-2 before sending it to the database server. Likewise, graphic data retrieved from a UCS-2 database by any application, or from any database by an application running under an EUC eucJP or eucTW code page, is encoded using UCS-2. This requires your application to convert from UCS-2 to your application code page internally, unless the user is to be presented with UCS-2 data.

Your application is responsible for converting to and from UCS-2 because this conversion must be conducted before the data is copied to, and after it is copied from, the SQLDA. DB2 Database for Linux, UNIX, and Windows does not supply any conversion routines that are accessible to your application. Instead, you must use the system calls available from your operating system. In the case of a UCS-2 database, you may also consider using the VARCHAR and VARGRAPHIC scalar functions.

**Related concepts:**

- "Japanese and Traditional Chinese EUC and UCS-2 code set considerations" in *Developing SQL and External Routines*

**Related reference:**

- "VARCHAR scalar function" in *SQL Reference, Volume 1*
- "VARGRAPHIC scalar function" in *SQL Reference, Volume 1*
- "Declaration of fixed length and variable length graphic host variables in COBOL embedded SQL applications" on page 113

## Binary storage of variable values using the FOR BIT DATA clause in COBOL embedded SQL applications

Certain database columns can be declared FOR BIT DATA. These columns, which generally contain characters, are used to hold binary information. The CHAR(*n*), VARCHAR, LONG VARCHAR, and BLOB data types are the COBOL host variable types that can contain binary data. Use these data types when working with columns with the FOR BIT DATA attribute.

**Related reference:**

- "Supported SQL data types in COBOL embedded SQL applications" on page 60
- "Declaration of large object type host variables in COBOL embedded SQL applications" on page 114

## Host structure support in the declare section of COBOL embedded SQL applications

The COBOL precompiler supports declarations of group data items in the host variable declare section. Among other things, this provides a shorthand for referring to a set of elementary data items in an SQL statement. For example, the following group data item can be used to access some of the columns in the STAFF table of the SAMPLE database:

```
01 staff-record.
    05 staff-id       pic s9(4) comp-5.
    05 staff-name.
        49 l          pic s9(4) comp-5.
        49 d          pic x(9).
    05 staff-info.
        10 staff-dept pic s9(4) comp-5.
        10 staff-job  pic x(5).
```

Group data items in the declare section can have any of the valid host variable types described above as subordinate data items. This includes all numeric and character types, as well as all large object types. You can nest group data items up to 10 levels. Note that you must declare VARCHAR character types with the subordinate items at level 49, as in the above example. If they are not at level 49, the VARCHAR is treated as a group data item with two subordinates, and is subject to the rules of declaring and using group data items. In the example above, `staff-info` is a group data item, whereas `staff-name` is a VARCHAR. The same principle applies to LONG VARCHAR, VARGRAPHIC, and LONG VARGRAPHIC. You may declare group data items at any level between 02 and 49.

You can use group data items and their subordinates in four ways:

**Method 1.**

The entire group may be referenced as a single host variable in an SQL statement:

```
    EXEC SQL SELECT id, name, dept, job
      INTO :staff-record
      FROM staff WHERE id = 10 END-EXEC.
```

The precompiler converts the reference to staff-record into a list, separated by commas, of all the subordinate items declared within `staff-record`. Each elementary item is qualified with the group names of all levels to prevent naming conflicts with other items. This is equivalent to the following method.

**Method 2.**

The second way of using group data items:

```
   EXEC SQL SELECT id, name, dept, job
     INTO
     :staff-record.staff-id,
     :staff-record.staff-name,
     :staff-record.staff-info.staff-dept,
     :staff-record.staff-info.staff-job
     FROM staff WHERE id = 10 END-EXEC.
```

**Note:** The reference to `staff-id` is qualified with its group name using the prefix `staff-record.`, and not `staff-id` of `staff-record` as in pure COBOL.

Assuming there are no other host variables with the same names as the subordinates of `staff-record`, the above statement can also be coded as in method 3, eliminating the explicit group qualification.

**Method 3.**

Here, subordinate items are referenced in a typical COBOL fashion, without being qualified to their particular group item:

```
   EXEC SQL SELECT id, name, dept, job
     INTO
     :staff-id,
     :staff-name,
     :staff-dept,
     :staff-job
     FROM staff WHERE id = 10 END-EXEC.
```

As in pure COBOL, this method is acceptable to the precompiler as long as a given subordinate item can be uniquely identified. If, for example, `staff-job` occurs in more than one group, the precompiler issues an error indicating an ambiguous reference:

```
   SQL0088N Host variable "staff-job" is ambiguous.
```

**Method 4.**

To resolve the ambiguous reference, you can use partial qualification of the subordinate item, for example:

```
   EXEC SQL SELECT id, name, dept, job
     INTO
     :staff-id,
     :staff-name,
     :staff-info.staff-dept,
     :staff-info.staff-job
     FROM staff WHERE id = 10 END-EXEC.
```

Because a reference to a group item alone, as in method 1, is equivalent to a comma-separated list of its subordinates, there are instances where this type of reference leads to an error. For example:

```
EXEC SQL CONNECT TO :staff-record END-EXEC.
```

Here, the CONNECT statement expects a single character-based host variable. By giving the `staff-record` group data item instead, the host variable results in the following precompile-time error:

```
SQL0087N Host variable "staff-record" is a structure used where
         structure references are not permitted.
```

Other uses of group items that cause an SQL0087N to occur include PREPARE, EXECUTE IMMEDIATE, CALL, indicator variables, and SQLDA references. Groups with only one subordinate are permitted in such situations, as are references to individual subordinates, as in methods 2, 3, and 4 above.

**Related reference:**

- "Declaration of large object type host variables in COBOL embedded SQL applications" on page 114
- "Declaration of numeric host variables in COBOL embedded SQL applications" on page 110
- "Supported SQL data types in COBOL embedded SQL applications" on page 60
- "Declaration of fixed length and variable length character host variables in COBOL embedded SQL applications" on page 111

## Null-indicator variables and null or truncation indicator variable tables in COBOL embedded SQL applications

Null-indicator variables should be declared as a `PIC S9(4) COMP-5` data type.

The COBOL precompiler supports the declaration of *null-indicator variable tables* (known as indicator tables), which are convenient to use with group data items. They are declared as follows:

```
01 <indicator-table-name>.
    05 <indicator-name> pic s9(4) comp-5
                        occurs <table-size> times.
```

For example:

```
01 staff-indicator-table.
    05 staff-indicator pic s9(4) comp-5
                       occurs 7 times.
```

This indicator table can be used effectively with the first format of group item reference above:

```
EXEC SQL SELECT id, name, dept, job
  INTO :staff-record :staff-indicator
  FROM staff WHERE id = 10 END-EXEC.
```

Here, the precompiler detects that `staff-indicator` was declared as an indicator table, and expands it into individual indicator references when it processes the

SQL statement. `staff-indicator`(1) is associated with `staff-id` of `staff-record`, `staff-indicator`(2) is associated with `staff-name` of `staff-record`, and so on.

**Note:** If there are `k` more indicator entries in the indicator table than there are subordinates in the data item (for example, if `staff-indicator` has 10 entries, making `k`=6), the `k` extra entries at the end of the indicator table are ignored. Likewise, if there are `k` fewer indicator entries than subordinates, the last `k` subordinates in the group item do not have indicators associated with them. *Note that you can refer to individual elements in an indicator table in an SQL statement*.

**Related concepts:**
- "Declare section for host variables in COBOL embedded SQL applications" on page 108

**Related reference:**
- "Supported SQL data types in COBOL embedded SQL applications" on page 60

## Host variables in FORTRAN

### Host variables in FORTRAN

Host variables are FORTRAN language variables that are referenced within SQL statements. They allow an application to exchange data with the database manager. After the application is precompiled, host variables are used by the compiler as any other FORTRAN variable. Follow the rules described in the following sections when naming, declaring, and using host variables.

**Related concepts:**
- "Declare section for host variables in FORTRAN embedded SQL applications" on page 121
- "Example: SQL declare section template for FORTRAN embedded SQL applications" on page 121
- "Host variable names in FORTRAN embedded SQL applications" on page 120

### Host variable names in FORTRAN embedded SQL applications

The SQL precompiler identifies host variables by their declared name. The following suggestions apply:
- Specify variable names up to 255 characters in length.
- Begin host variable names with prefixes other than `SQL`, `sql`, `DB2`, or `db2`, which are reserved for system use.

**Related concepts:**
- "Declare section for host variables in FORTRAN embedded SQL applications" on page 121

**Related reference:**
- "Declaration of fixed-length and variable length character host variables in FORTRAN embedded SQL applications" on page 123
- "Declaration of file reference type host variables in FORTRAN embedded SQL applications" on page 126

- "Declaration of large object type host variables in FORTRAN embedded SQL applications" on page 125
- "Declaration of large object locator type host variables in FORTRAN embedded SQL applications" on page 126
- "Declaration of numeric host variables in FORTRAN embedded SQL applications" on page 122

## Declare section for host variables in FORTRAN embedded SQL applications

An SQL declare section must be used to identify host variable declarations. This alerts the precompiler to any host variables that can be referenced in subsequent SQL statements.

The FORTRAN precompiler only recognizes a subset of valid FORTRAN declarations as valid host variable declarations. These declarations define either numeric or character variables. A numeric host variable can be used as an input or output variable for any numeric SQL input or output value. A character host variable can be used as an input or output variable for any character, date, time or timestamp SQL input or output value. The programmer must ensure that output variables are long enough to contain the values that they will receive.

**Related tasks:**
- "Declaring structured type host variables" in *Developing SQL and External Routines*

**Related reference:**
- "Declaration of fixed-length and variable length character host variables in FORTRAN embedded SQL applications" on page 123
- "Declaration of file reference type host variables in FORTRAN embedded SQL applications" on page 126
- "Declaration of large object type host variables in FORTRAN embedded SQL applications" on page 125
- "Declaration of large object locator type host variables in FORTRAN embedded SQL applications" on page 126
- "Declaration of numeric host variables in FORTRAN embedded SQL applications" on page 122

## Example: SQL declare section template for FORTRAN embedded SQL applications

The following is a sample SQL declare section with a host variable declared for each supported data type:

```
EXEC SQL BEGIN DECLARE SECTION
  INTEGER*2     AGE  /26/                  /* SQL type  500 */
  INTEGER*4     DEPT                      /* SQL type  496 */
  REAL*4        BONUS                     /* SQL type  480 */
  REAL*8        SALARY                    /* SQL type  480 */
  CHARACTER     MI                        /* SQL type  452 */
  CHARACTER*112 ADDRESS                   /* SQL type  452 */
  SQL TYPE IS VARCHAR (512) DESCRIPTION   /* SQL type  448 */
  SQL TYPE IS VARCHAR (32000) COMMENTS    /* SQL type  448 */
  SQL TYPE IS CLOB (1M) CHAPTER           /* SQL type  408 */
  SQL TYPE IS CLOB_LOCATOR CHAPLOC        /* SQL type  964 */
  SQL TYPE IS CLOB_FILE  CHAPFL           /* SQL type  920 */
  SQL TYPE IS BLOB (1M) VIDEO             /* SQL type  404 */
```

```
    SQL TYPE IS BLOB_LOCATOR VIDLOC            /* SQL type  960 */
    SQL TYPE IS BLOB_FILE VIDFL                /* SQL type  916 */
    CHARACTER*10  DATE                         /* SQL type  384 */
    CHARACTER*8   TIME                         /* SQL type  388 */
    CHARACTER*26  TIMESTAMP                    /* SQL type  392 */
    INTEGER*2     WAGE_IND                     /* SQL type  500 */
  EXEC SQL END DECLARE SECTION
```

**Related concepts:**

- "Declare section for host variables in FORTRAN embedded SQL applications" on page 121

**Related reference:**

- "Supported SQL data types in FORTRAN embedded SQL applications" on page 63

## SQLSTATE and SQLCODE variables in FORTRAN embedded SQL application

When using the LANGLEVEL precompile option with a value of SQL92E, the following two declarations may be included as host variables:

```
  EXEC SQL BEGIN DECLARE SECTION;
    CHARACTER*5 SQLSTATE
    INTEGER     SQLCOD
    .
    .
    .
  EXEC SQL END DECLARE SECTION
```

The `SQLCOD` declaration is assumed during the precompile step. The variable named `SQLSTATE` may also be `SQLSTA`. Note that when using this option, the INCLUDE SQLCA statement should not be specified.

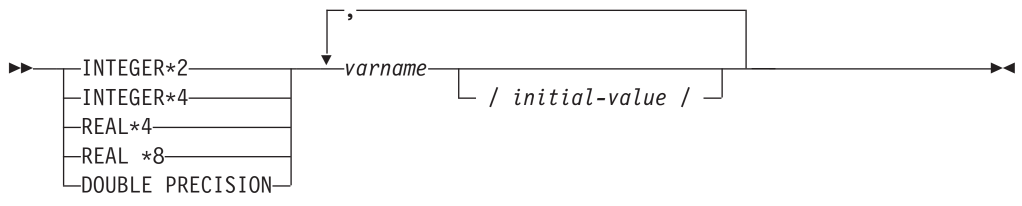
For applications that contain multiple source files, the declarations of `SQLCOD` and `SQLSTATE` may be included in each source file, as shown above.

**Related reference:**

- "PRECOMPILE command" in *Command Reference*
- "INCLUDE statement" in *SQL Reference, Volume 2*

## Declaration of numeric host variables in FORTRAN embedded SQL applications

Following is the syntax for numeric host variables in FORTRAN.



**Numeric host variable considerations:**

1. REAL*8 and DOUBLE PRECISION are equivalent.
2. Use an E rather than a D as the exponent indicator for REAL*8 constants.

**Related concepts:**

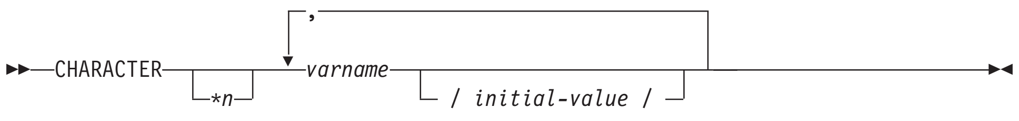
- "Host variable names in FORTRAN embedded SQL applications" on page 120
- "Declare section for host variables in FORTRAN embedded SQL applications" on page 121
- "Example: SQL declare section template for FORTRAN embedded SQL applications" on page 121

## Declaration of fixed-length and variable length character host variables in FORTRAN embedded SQL applications

Following is the syntax for fixed-length character host variables.

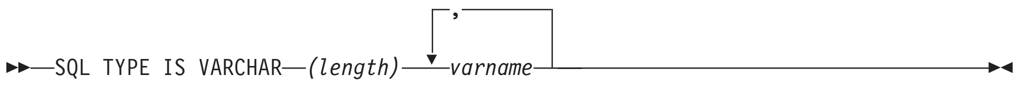
**`Fixed length`**

### Syntax for character host variables in FORTRAN: fixed length



Following is the syntax for variable-length character host variables.

**`Variable length`**



**Character host variable considerations:**

1. **n* has a maximum value of 254.
2. When length is between 1 and 32 672 inclusive, the host variable has type VARCHAR(SQLTYPE 448).
3. When length is between 32 673 and 32 700 inclusive, the host variable has type LONG VARCHAR(SQLTYPE 456).
4. Initialization of VARCHAR and LONG VARCHAR host variables is not permitted within the declaration.

**VARCHAR example:**

Declaring:

```
sql type is varchar(1000) my_varchar
```

Results in the generation of the following structure:

```
 character    my_varchar(1000+2)
 integer*2    my_varchar_length
 character    my_varchar_data(1000)
 equivalence( my_varchar(1),
+             my_varchar_length )
 equivalence( my_varchar(3),
+             my_varchar_data )
```

The application may manipulate both `my_varchar_length` and `my_varchar_data`; for example, to set or examine the contents of the host variable. The base name (in this case, `my_varchar`), is used in SQL statements to refer to the VARCHAR as a whole.

**LONG VARCHAR example:**

Declaring:

```
sql type is varchar(10000) my_lvarchar
```

Results in the generation of the following structure:

```
 character    my_lvarchar(10000+2)
 integer*2    my_lvarchar_length
 character    my_lvarchar_data(10000)
 equivalence( my_lvarchar(1),
+             my_lvarchar_length )
 equivalence( my_lvarchar(3),
+             my_lvarchar_data )
```

The application may manipulate both `my_lvarchar_length` and `my_lvarchar_data`; for example, to set or examine the contents of the host variable. The base name (in this case, `my_lvarchar`), is used in SQL statements to refer to the LONG VARCHAR as a whole.

**Note:** In a CONNECT statement, such as in the following example, the FORTRAN character string host variables `dbname` and `userid` will have any trailing blanks removed before processing.

```
EXEC SQL CONNECT TO :dbname USER :userid USING :passwd
```

However, because blanks can be significant in passwords, you should declare host variables for passwords as VARCHAR, and have the length field set to reflect the actual password length:

```
EXEC SQL BEGIN DECLARE SECTION
  character*8 dbname, userid
  sql type is varchar(18) passwd
EXEC SQL END DECLARE SECTION
character*18 passwd_string
equivalence(passwd_data,passwd_string)
dbname = 'sample'
userid = 'userid'
passwd_length= 8
passwd_string = 'password'
EXEC SQL CONNECT TO :dbname USER :userid USING :passwd
```

**Related concepts:**

- "Host variable names in FORTRAN embedded SQL applications" on page 120
- "Declare section for host variables in FORTRAN embedded SQL applications" on page 121
- "Example: SQL declare section template for FORTRAN embedded SQL applications" on page 121

**Related reference:**

- "CONNECT (Type 1) statement" in *SQL Reference, Volume 2*

## Declaration of large object type host variables in FORTRAN embedded SQL applications

Following is the syntax for declaring large object (LOB) host variables in FORTRAN.

```
                                                  ┌─,─────────────┐
                                                  ▼               │
►►──SQL TYPE IS──┬─BLOB─┬──(length──┬───┬──)──────variable-name───┴──────────────►◄
                 └─CLOB─┘           ├─K─┤
                                    ├─M─┤
                                    └─G─┘
```

**LOB host variable considerations:**

1. GRAPHIC types are not supported in FORTRAN.
2. SQL TYPE IS, BLOB, CLOB, K, M, G can be in either uppercase, lowercase, or mixed.
3. For BLOB and CLOB 1 <= lob-length <= 2 147 483 647.
4. The initialization of a LOB within a LOB declaration is not permitted.
5. The host variable name prefixes ’length’ and ’data’ in the precompiler generated code.

**BLOB example:**

Declaring:

```
sql type is blob(2m) my_blob
```

Results in the generation of the following structure:

```
 character    my_blob(2097152+4)
 integer*4    my_blob_length
 character    my_blob_data(2097152)
 equivalence( my_blob(1),
+             my_blob_length )
 equivalence( my_blob(5),
+             my_blob_data )
```

**CLOB example:**

Declaring:

```
sql type is clob(125m) my_clob
```

Results in the generation of the following structure:

```
 character    my_clob(131072000+4)
 integer*4    my_clob_length
 character    my_clob_data(131072000)
 equivalence( my_clob(1),
+             my_clob_length )
 equivalence( my_clob(5),
+             my_clob_data )
```

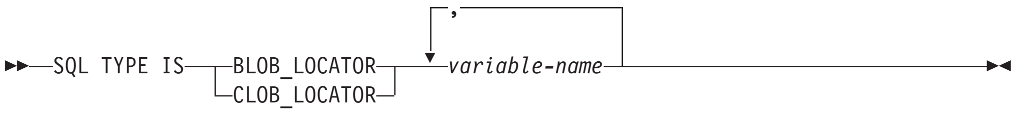
**Related concepts:**

- “Host variable names in FORTRAN embedded SQL applications” on page 120
- “Declare section for host variables in FORTRAN embedded SQL applications” on page 121

- "Example: SQL declare section template for FORTRAN embedded SQL applications" on page 121

## Declaration of large object locator type host variables in FORTRAN embedded SQL applications

Following is the syntax for declaring large object (LOB) locator host variables in FORTRAN.



**LOB locator host variable considerations:**

1. GRAPHIC types are not supported in FORTRAN.
2. SQL TYPE IS, BLOB_LOCATOR, CLOB_LOCATOR can be either uppercase, lowercase, or mixed.
3. Initialization of locators is not permitted.

**CLOB locator example** (BLOB locator is similar):

Declaring:

```
SQL TYPE IS CLOB_LOCATOR my_locator
```

Results in the generation of the following declaration:

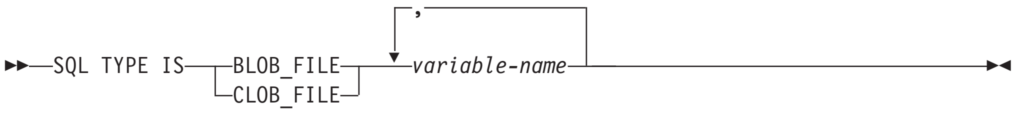
```
integer*4 my_locator
```

**Related concepts:**

- "Declare section for host variables in FORTRAN embedded SQL applications" on page 121
- "Host variable names in FORTRAN embedded SQL applications" on page 120

## Declaration of file reference type host variables in FORTRAN embedded SQL applications

Following is the syntax for declaring file reference host variables in FORTRAN.



**File reference host variable considerations:**

1. Graphic types are not supported in FORTRAN.
2. SQL TYPE IS, BLOB_FILE, CLOB_FILE can be either uppercase, lowercase, or mixed.

**Example of a BLOB file reference variable** (CLOB file reference variable is similar):

```
SQL TYPE IS BLOB_FILE my_file
```

Results in the generation of the following declaration:

```
  character     my_file(267)
  integer*4     my_file_name_length
  integer*4     my_file_data_length
  integer*4     my_file_file_options
  character*255 my_file_name
  equivalence(  my_file(1),
 +              my_file_name_length )
  equivalence(  my_file(5),
 +              my_file_data_length )
  equivalence(  my_file(9),
 +              my_file_file_options )
  equivalence(  my_file(13),
 +              my_file_name )
```

**Related concepts:**

- "Considerations for graphic (multi-byte) character sets in FORTRAN embedded SQL applications" on page 127

**Related reference:**

- "Declaration of large object type host variables in FORTRAN embedded SQL applications" on page 125

## Considerations for graphic (multi-byte) character sets in FORTRAN embedded SQL applications

There are no graphic (multi-byte) host variable data types supported in FORTRAN. Only mixed-character host variables are supported through the `character` data type. However, it is possible to create a user SQLDA that contains graphic data.

**Related concepts:**

- "Host variable names in FORTRAN embedded SQL applications" on page 120
- "Declare section for host variables in FORTRAN embedded SQL applications" on page 121
- "Example: SQL declare section template for FORTRAN embedded SQL applications" on page 121

**Related tasks:**

- "Transferring data in a dynamically executed SQL program using an SQLDA structure" on page 148

## Japanese or Traditional Chinese EUC, and UCS-2 considerations for FORTRAN embedded SQL applications

Any graphic data sent from your application running under an eucJp or eucTW code set, or connected to a UCS-2 database, is tagged with the UCS-2 code page identifier. Your application must convert a graphic-character string to UCS-2 before sending it to a the database server. Likewise, graphic data retrieved from a UCS-2 database by any application, or from any database by an application running under an EUC eucJP or eucTW code page, is encoded using UCS-2. This requires your application to convert from UCS-2 to your application code page internally, unless the user is to be presented with UCS-2 data.

Your application is responsible for converting to and from UCS-2 because this conversion must be conducted before the data is copied to, and after it is copied from, the SQLDA. DB2 database systems do not supply any conversion routines that are accessible to your application. Instead, you must use the system calls available from your operating system. In the case of a UCS-2 database, you can also consider using the VARCHAR and VARGRAPHIC scalar functions.

**Related concepts:**

- "Japanese and Traditional Chinese EUC and UCS-2 code set considerations" in *Developing SQL and External Routines*

**Related reference:**

- "VARCHAR scalar function" in *SQL Reference, Volume 1*
- "VARGRAPHIC scalar function" in *SQL Reference, Volume 1*

### Null or truncation indicator variables in FORTRAN embedded SQL applications

Indicator variables should be declared as an `INTEGER*2` data type.

**Related concepts:**

- "Declare section for host variables in FORTRAN embedded SQL applications" on page 121
- "Example: SQL declare section template for FORTRAN embedded SQL applications" on page 121
- "Host variables in FORTRAN" on page 120

**Related reference:**

- "Supported SQL data types in FORTRAN embedded SQL applications" on page 63

## Host variables in REXX embedded SQL applications

### Host variables in REXX

Host variables are REXX language variables that are referenced within SQL statements. They allow an application to exchange data with the database manager. After the application is precompiled, host variables are used by the compiler as any other REXX variable. Follow the rules described in the following sections when naming, declaring, and using host variables.

**Related concepts:**

- "Host variable names in REXX embedded SQL applications" on page 128
- "Host variable references in REXX embedded SQL applications" on page 129

**Related tasks:**

- "Considerations while programming REXX embedded SQL applications" on page 131

**Related reference:**

- "Predefined REXX Variables" on page 129

### Host variable names in REXX embedded SQL applications

Any properly named REXX variable can be used as a host variable. A variable name can be up to 64 characters long. Do not end the name with a period. A host variable name can consist of numbers, alphabetic characters, and the characters @, _, !, ., ?, and $.

**Related concepts:**

- "Host variable references in REXX embedded SQL applications" on page 129

**Related reference:**

- "REXX samples" on page 377
- "Supported SQL data types in REXX embedded SQL applications" on page 64

## Host variable references in REXX embedded SQL applications

The REXX interpreter examines every string without quotation marks in a procedure. If the string represents a variable in the current REXX variable pool, REXX replaces the string with the current value. The following is an example of how you can reference a host variable in REXX:

```
CALL SQLEXEC 'FETCH C1 INTO :cm'
SAY 'Commission = ' cm
```

To ensure that a character string is not converted to a numeric data type, enclose the string with single quotation marks as in the following example:

```
VAR = '100'
```

REXX sets the variable *VAR* to the 3-byte character string `100`. If single quotation marks are to be included as part of the string, follow this example:

```
VAR = "'100'"
```

When inserting numeric data into a CHARACTER field, the REXX interpreter treats numeric data as integer data, thus you must concatenate numeric strings explicitly and surround them with single quotation marks.

**Related reference:**

- "Supported SQL data types in REXX embedded SQL applications" on page 64
- "REXX samples" on page 377

## Predefined REXX Variables

SQLEXEC, SQLDBS, and SQLDB2 set predefined REXX variables as a result of certain operations. These variables are:

**RESULT**
Each operation sets this return code. Possible values are:

*n* Where *n* is a positive value indicating the number of bytes in a formatted message. The GET ERROR MESSAGE API alone returns this value.

**0** The API was executed. The REXX variable SQLCA contains the completion status of the API. If SQLCA.SQLCODE is not zero, SQLMSG contains the text message associated with that value.

**–1** There is not enough memory available to complete the API. The requested message was not returned.

**–2** SQLCA.SQLCODE is set to 0. No message was returned.

**–3** SQLCA.SQLCODE contained an invalid SQLCODE. No message was returned.

| | |
|---|---|
| **–6** | The SQLCA REXX variable could not be built. This indicates that there was not enough memory available or the REXX variable pool was unavailable for some reason. |
| **–7** | The SQLMSG REXX variable could not be built. This indicates that there was not enough memory available or the REXX variable pool was unavailable for some reason. |
| **–8** | The SQLCA.SQLCODE REXX variable could not be fetched from the REXX variable pool. |
| **–9** | The SQLCA.SQLCODE REXX variable was truncated during the fetch. The maximum length for this variable is 5 bytes. |
| **–10** | The SQLCA.SQLCODE REXX variable could not be converted from ASCII to a valid long integer. |
| **–11** | The SQLCA.SQLERRML REXX variable could not be fetched from the REXX variable pool. |
| **–12** | The SQLCA.SQLERRML REXX variable was truncated during the fetch. The maximum length for this variable is 2 bytes. |
| **–13** | The SQLCA.SQLERRML REXX variable could not be converted from ASCII to a valid short integer. |
| **–14** | The SQLCA.SQLERRMC REXX variable could not be fetched from the REXX variable pool. |
| **–15** | The SQLCA.SQLERRMC REXX variable was truncated during the fetch. The maximum length for this variable is 70 bytes. |
| **–16** | The REXX variable specified for the error text could not be set. |
| **–17** | The SQLCA.SQLSTATE REXX variable could not be fetched from the REXX variable pool. |
| **–18** | The SQLCA.SQLSTATE REXX variable was truncated during the fetch. The maximum length for this variable is 2 bytes. |

**Note:** The values –8 through –18 are returned only by the GET ERROR MESSAGE API.

**SQLMSG**
If SQLCA.SQLCODE is not 0, this variable contains the text message associated with the error code.

**SQLISL**
The isolation level. Possible values are:

| | |
|---|---|
| **RR** | Repeatable read. |
| **RS** | Read stability. |
| **CS** | Cursor stability. This is the default. |
| **UR** | Uncommitted read. |
| **NC** | No commit. (NC is only supported by some host, AS/400®, or iSeries servers.) |

**SQLCA**
The SQLCA structure updated after SQL statements are processed and DB2 APIs are called.

**SQLRODA**
The input/output SQLDA structure for stored procedures invoked using

the CALL statement. It is also the output SQLDA structure for stored procedures invoked using the Database Application Remote Interface (DARI) API.

**SQLRIDA**
The input SQLDA structure for stored procedures invoked using the Database Application Remote Interface (DARI) API.

**SQLRDAT**
An SQLCHAR structure for server procedures invoked using the Database Application Remote Interface (DARI) API.

**Related concepts:**

- "API Syntax for REXX" on page 157
- "Errors and warnings from precompilation of embedded SQL applications" on page 189

**Related tasks:**

- "Declaring the SQLCA for Error Handling" on page 50

**Related reference:**

- "SQLCA data structure" in *Administrative API Reference*
- "sqlchar data structure" in *Administrative API Reference*
- "SQLDA data structure" in *Administrative API Reference*

## Considerations while programming REXX embedded SQL applications

REXX is an interpreted language. Thus no precompiler, compiler, or linker is used. Instead, three DB2 APIs are used to create DB2 applications in REXX. Use these APIs to access different elements of DB2.

**SQLEXEC**
Supports the SQL language.

**SQLDBS**
Supports command-like versions of DB2 APIs.

**SQLDB2**
Supports a REXX specific interface to the command-line processor. See the description of the API syntax for REXX for details and restrictions on how this interface can be used.

Before using any of the DB2 APIs or issuing SQL statements in an application, you must register the SQLDBS, SQLDB2 and SQLEXEC routines. This notifies the REXX interpreter of the REXX/SQL entry points. The method you use for registering varies slightly between Windows-based and AIX platforms.

**Procedure:**

Use the following examples for correct syntax for registering each routine:

**Sample registration on Windows operating systems**

```
/* ------------ Register SQLDBS with REXX  -------------------------*/
If Rxfuncquery('SQLDBS')  <> 0 then
    rcy = Rxfuncadd('SQLDBS','DB2AR','SQLDBS')
If rcy \= 0 then
    do
```

```
      say 'SQLDBS was not successfully added to the REXX environment'
      signal rxx_exit
    end

/* ------------ Register SQLDB2 with REXX  -------------------------*/
If Rxfuncquery('SQLDB2')  <> 0 then
    rcy = Rxfuncadd('SQLDB2','DB2AR','SQLDB2')
If rcy \= 0 then
    do
      say 'SQLDB2 was not successfully added to the REXX environment'
      signal rxx_exit
    end

/* ----------------- Register SQLEXEC with REXX  --------------------*/
If Rxfuncquery('SQLEXEC') <> 0 then
    rcy = Rxfuncadd('SQLEXEC','DB2AR','SQLEXEC')
If rcy \= 0 then
    do
      say 'SQLEXEC was not successfully added to the REXX environment'
      signal rxx_exit
    end
```

**Sample registration on AIX**

```
  /* ------------ Register SQLDBS, SQLDB2 and SQLEXEC with REXX --------*/
 rcy = SysAddFuncPkg("db2rexx")
 If rcy \= 0 then
   do
     say 'db2rexx was not successfully added to the REXX environment'
     signal rxx_exit
   end
```

On Windows-based platforms, the RxFuncAdd commands need to be executed only once for all sessions.

On AIX, the SysAddFuncPkg should be executed in every REXX/SQL application.

Details on the Rxfuncadd and SysAddFuncPkg APIs are available in the REXX documentation for Windows-based platforms and AIX, respectively.

It is possible that tokens within statements or commands that are passed to the SQLEXEC, SQLDBS, and SQLDB2 routines could correspond to REXX variables. In this case, the REXX interpreter substitutes the variable's value before calling SQLEXEC, SQLDBS, or SQLDB2.

To avoid this situation, enclose statement strings in quotation marks (' ' or " "). If you do not use quotation marks, any conflicting variable names are resolved by the REXX interpreter, instead of being passed to the SQLEXEC, SQLDBS or SQLDB2 routines.

**Related concepts:**
- "API Syntax for REXX" on page 157

## Declaration of large object type host variables in REXX embedded SQL applications

When you fetch a LOB column into a REXX host variable, it will be stored as a simple (that is, uncounted) string. This is handled in the same manner as all character-based SQL types (such as CHAR, VARCHAR, GRAPHIC, LONG, and so on). On input, if the size of the contents of your host variable is larger than 32K, or if it meets other criteria set out below, it will be assigned the appropriate LOB type.

In REXX SQL, LOB types are determined from the string content of your host variable as follows:

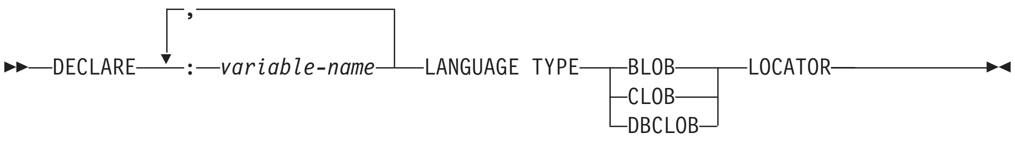
| Host variable string content | Resulting LOB type |
|---|---|
| :hv1='ordinary quoted string longer than 32K ...' | CLOB |
| :hv2="'string with embedded delimiting quotation marks ", "longer than 32K...'" | CLOB |
| :hv3="G'DBCS string with embedded delimiting single ", "quotation marks, beginning with G, longer than 32K...'" | DBCLOB |
| :hv4="BIN'string with embedded delimiting single ", "quotation marks, beginning with BIN, any length...'" | BLOB |

**Related concepts:**
- "LOB Host Variable Clearing in REXX embedded SQL applications" on page 135

**Related reference:**
- "Declaration of large object locator type host variables in REXX embedded SQL applications" on page 133

## Declaration of large object locator type host variables in REXX embedded SQL applications

The following shows the syntax for declaring LOB locator host variables in REXX:

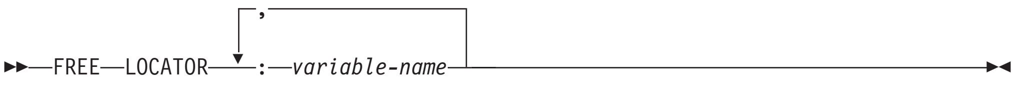


You must declare LOB locator host variables in your application. When REXX/SQL encounters these declarations, it treats the declared host variables as locators for the remainder of the program. Locator values are stored in REXX variables in an internal format.

Example:
```
CALL SQLEXEC 'DECLARE :hv1, :hv2 LANGUAGE TYPE CLOB LOCATOR'
```

Data represented by LOB locators returned from the engine can be freed in REXX/SQL using the FREE LOCATOR statement which has the following format:

### Syntax for FREE LOCATOR statement



Example:
```
CALL SQLEXEC 'FREE LOCATOR :hv1, :hv2'
```

**Related concepts:**
- "Host variable names in REXX embedded SQL applications" on page 128

**Related reference:**
- "FREE LOCATOR statement" in *SQL Reference, Volume 2*
- "Supported SQL data types in REXX embedded SQL applications" on page 64
- "REXX samples" on page 377

## Declaration of file reference type host variables in REXX embedded SQL applications

You must declare LOB file reference host variables in your application. When REXX/SQL encounters these declarations, it treats the declared host variables as LOB file references for the remainder of the program.

The following shows the syntax for declaring LOB file reference host variables in REXX:

```
              ┌─,──────────────────┐
              ▼                    │
►►──DECLARE──────:──variable-name──┴──LANGUAGE TYPE──┬─BLOB───┬──FILE────────────────►◄
                                                     ├─CLOB───┤
                                                     └─DBCLOB─┘
```

Example:

```
CALL SQLEXEC 'DECLARE :hv3, :hv4 LANGUAGE TYPE CLOB FILE'
```

File reference variables in REXX contain three fields. For the above example they are:

**`hv3.FILE_OPTIONS.`**
Set by the application to indicate how the file will be used.

**`hv3.DATA_LENGTH.`**
Set by DB2 to indicate the size of the file.

**`hv3.NAME.`**
Set by the application to the name of the LOB file.

For FILE_OPTIONS, the application sets the following keywords:

| **Keyword (integer value)** | **Meaning** |
|---|---|
| **READ (2)** | File is to be used for input. This is a regular file that can be opened, read and closed. The length of the data in the file (in bytes) is computed (by the application requestor code) upon opening the file. |
| **CREATE (8)** | On output, create a new file. If the file already exists, it is an error. The length (in bytes) of the file is returned in the `DATA_LENGTH` field of the file reference variable structure. |
| **OVERWRITE (16)** | On output, the existing file is overwritten if it exists, otherwise a new file is created. The length (in bytes) of the file is returned in the `DATA_LENGTH` field of the file reference variable structure. |
| **APPEND (32)** | The output is appended to the file if it exists, otherwise a new file is created. The length (in bytes) of the data that was added to the |

file (not the total file length) is returned in the DATA_LENGTH field of the file reference variable structure.

**Note:** A file reference host variable is a compound variable in REXX, thus you must set values for the NAME, NAME_LENGTH and FILE_OPTIONS fields in addition to declaring them.

**Related concepts:**
- "Host variable names in REXX embedded SQL applications" on page 128

**Related reference:**
- "REXX samples" on page 377
- "Supported SQL data types in REXX embedded SQL applications" on page 64

## LOB Host Variable Clearing in REXX embedded SQL applications

On Windows-based platforms it may be necessary to explicitly clear REXX SQL LOB locator and file reference host variable declarations as they remain in effect after your application program ends. This occurs because the application process does not exit until the session in which it is run is closed. If REXX SQL LOB declarations are not cleared, they may interfere with other applications that are running in the same session after a LOB application has been executed.

The syntax to clear the declaration is:

```
CALL SQLEXEC "CLEAR SQL VARIABLE DECLARATIONS"
```

You should code this statement at the end of LOB applications. Note that you can code it anywhere as a precautionary measure to clear declarations which might have been left by previous applications (for example, at the beginning of a REXX SQL application).

**Related concepts:**
- "Performance of embedded SQL applications" on page 22

**Related tasks:**
- "Building and running embedded SQL applications written in REXX" on page 239

## Null or truncation indicator variables in REXX embedded SQL applications

An indicator variable data type in REXX is a number without a decimal point. Following is an example of an indicator variable in REXX using the INDICATOR keyword.

```
CALL SQLEXEC 'FETCH C1 INTO :cm INDICATOR :cmind'
IF ( cmind < 0 )
   SAY 'Commission is NULL'
```

In the above example, cmind is examined for a negative value. If it is not negative, the application can use the returned value of cm. If it is negative, the fetched value is NULL and cm should not be used. The database manager does not change the value of the host variable in this case.

**Related concepts:**
- "Host variables in REXX" on page 128

**Related reference:**
- "Supported SQL data types in REXX embedded SQL applications" on page 64

## Executing SQL statements in embedded SQL applications

### Executing SQL statements in embedded SQL applications

Executing SQL statements in embedded SQL applications is different for statically and dynamically executed statements, although they both make use of the `EXEC SQL` command. Static statements are hard-coded into the source code of an embedded SQL application. Dynamic statements are different from static in that they are compiled at run time and may be prepared with input parameters. Information that is read may be stored in a medium called a cursor, which then allows for users to freely scroll through the data and make suitable updates. Error information from the SQLCODE, SQLSTATE, and SQLWARN are a useful tool towards assisting in troubleshooting an application.

**Related concepts:**
- "Calling stored procedures in embedded SQL applications" on page 155
- "Comments in embedded SQL applications" on page 136
- "Error message retrieval in embedded SQL applications" on page 174
- "Executing static SQL statements in embedded SQL applications" on page 137
- "Reading and scrolling through result sets in embedded SQL applications" on page 160

**Related tasks:**
- "Providing variable input to dynamically executed SQL statement using parameter markers" on page 153

### Comments in embedded SQL applications

The comments in any application are important for making the application code appear understandable. This section will discuss adding comments in embedded SQL applications.

**Comments in C and C++ embedded SQL applications:**

When working with C and C++ applications, SQL comments can be inserted within the `EXEC SQL` block. For example:

```
/* Only C or C++ comments allowed here */
 EXEC SQL
    -- SQL comments or
    /* C comments or */
    // C++ comments allowed here
    DECLARE C1 CURSOR FOR sname;
 /* Only C or C++ comments allowed here */
```

**Comments in COBOL embedded SQL applications:**

When working with COBOL applications, SQL comments can be inserted within the `EXEC SQL` block. For example:

```
*   See COBOL documentation for comment rules
*   Only COBOL comments are allowed here
    EXEC SQL
```

```
     -- SQL comments or
*    full-line COBOL comments are allowed here
     DECLARE C1 CURSOR FOR sname END-EXEC.
*    Only COBOL comments are allowed here
```

**Comments in FORTRAN embedded SQL applications:**

When working with FORTRAN applications, SQL comments can be inserted within the `EXEC SQL` block. For example:

```
C     Only FORTRAN comments are allowed here
      EXEC SQL
      + -- SQL comments, and
C     full-line FORTRAN comment are allowed here
      + DECLARE C1 CURSOR FOR sname
      I=7 ! End of line FORTRAN comments allowed here
C     Only FORTRAN comments are allowed here
```

**Comments in REXX embedded SQL applications:**

SQL comments are not supported in REXX applications.

**Related concepts:**
- "Embedding SQL statements in a host language" on page 4

## Executing static SQL statements in embedded SQL applications

SQL statements can be executed statically in a host language using the following approach:
- C or C++ (**tut_mod.sqc/tut_mod.sqC**)

  The following three examples are from the **tut_mod** sample. See this sample for a complete program that shows how to modify table data in C or C++.

  The following example shows how to insert table data:

  ```
  EXEC SQL INSERT INTO staff(id, name, dept, job, salary)
    VALUES(380, 'Pearce', 38, 'Clerk', 13217.50),
          (390, 'Hachey', 38, 'Mgr', 21270.00),
          (400, 'Wagland', 38, 'Clerk', 14575.00);
  ```

  The following example shows how to update table data:

  ```
  EXEC SQL UPDATE staff
    SET salary = salary + 10000
    WHERE id >= 310 AND dept = 84;
  ```

  The following example shows how to delete from a table:

  ```
  EXEC SQL DELETE
    FROM staff
    WHERE id >= 310 AND salary > 20000;
  ```
- COBOL (**updat.sqb**)

  The following three examples are from the **updat** sample. See this sample for a complete program that shows how to modify table data in COBOL.

  The following example shows how to insert table data:

  ```
  EXEC SQL INSERT INTO staff
      VALUES (999, 'Testing', 99, :job-update, 0, 0, 0)
      END-EXEC.
  ```

  The following example shows how to update table data where `job-update` is a reference to a host variable declared in the declaration section of the source code:

```
EXEC SQL UPDATE staff
    SET job=:job-update
    WHERE job='Mgr'
    END-EXEC.
```

The following example shows how to delete from a table:

```
EXEC SQL DELETE
    FROM staff
    WHERE job=:job-update
    END-EXEC.
```

**Related concepts:**

- "Determining when to execute SQL statements statically or dynamically in embedded SQL applications" on page 19
- "Embedding SQL statements in a host language" on page 4
- "Static and dynamic SQL statement execution in embedded SQL applications" on page 17

## Retrieving host variable information from the SQLDA structure in embedded SQL applications

### Retrieving host variable information from the SQLDA structure in embedded SQL applications

With static SQL, host variables used in embedded SQL statements are known at application compile time. With dynamic SQL, the embedded SQL statements and consequently the host variables are not known until application run time. Thus, for dynamic SQL applications, you need to deal with the list of host variables that are used in your application. You can use the DESCRIBE statement to obtain host variable information for any SELECT statement that has been prepared (using PREPARE), and store that information into the SQL descriptor area (SQLDA).

When the DESCRIBE statement gets executed in your application, the database manager defines your host variables in an SQLDA. Once the host variables are defined in the SQLDA, you can use the FETCH statement to assign values to the host variables, using a cursor.

**Related concepts:**

- "Example of a cursor in a dynamically executed SQL application" on page 170
- "Determining when to execute SQL statements statically or dynamically in embedded SQL applications" on page 19

**Related reference:**

- "DESCRIBE statement" in *SQL Reference, Volume 2*
- "FETCH statement" in *SQL Reference, Volume 2*
- "PREPARE statement" in *SQL Reference, Volume 2*
- "SQLDA data structure" in *Administrative API Reference*

### Declaring the SQLDA structure in a dynamically executed SQL program

An SQLDA contains a variable number of occurrences of SQLVAR entries, each of which contains a set of fields that describe one column in a row of data, as shown in the following figure. There are two types of SQLVAR entries: base SQLVAR entries, and secondary SQLVAR entries.

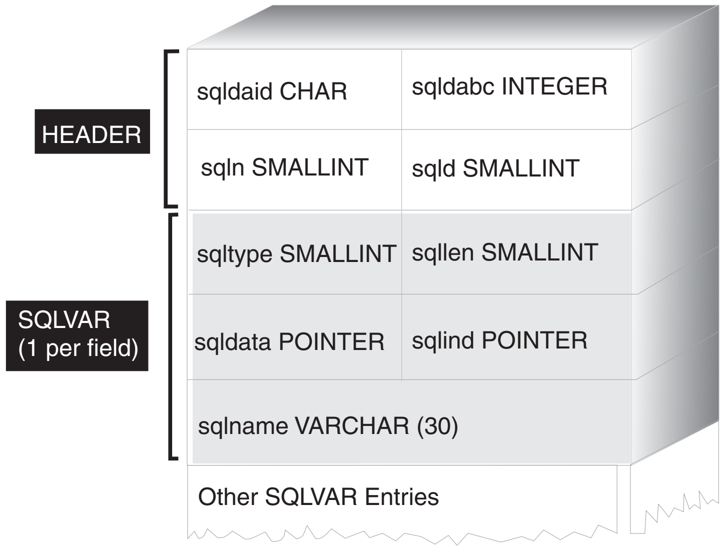


*Figure 3. The SQL Descriptor Area (SQLDA)*

**Procedure:**

Because the number of SQLVAR entries required depends on the number of columns in the result table, an application must be able to allocate an appropriate number of SQLVAR elements when needed. Use one of the following methods:

- Provide the largest SQLDA (that is, the one with the greatest number of SQLVAR entries) that is needed. The maximum number of columns that can be returned in a result table is 255. If any of the columns being returned is either a LOB type or a distinct type, the value in SQLN is doubled, and the number of SQLVAR entries needed to hold the information is doubled to 510. However, as most SELECT statements do not even retrieve 255 columns, most of the allocated space is unused.
- Provide a smaller SQLDA with fewer SQLVAR entries. In this case, if there are more columns in the result than SQLVAR entries allowed for in the SQLDA, no descriptions are returned. Instead, the database manager returns the number of select list items detected in the SELECT statement. The application allocates an SQLDA with the required number of SQLVAR entries, then uses the DESCRIBE statement to acquire the column descriptions.
- When the number of columns returned is of an LOB or user defined type, provide an SQLDA with the exact number of SQLVAR entries

For all three methods, the question arises as to how many initial SQLVAR entries you should allocate. Each SQLVAR element uses up 44 bytes of storage (not counting storage allocated for the SQLDATA and SQLIND fields). If memory is plentiful, the first method of providing an SQLDA of maximum size is easier to implement.

The second method of allocating a smaller SQLDA is only applicable to programming languages such as C and C++ that support the dynamic allocation of memory. For languages such as COBOL and FORTRAN that do not support the dynamic allocation of memory, use the first method.

**Related concepts:**

- "Identifying XML values in an SQLDA" on page 72

**Related tasks:**

- "Allocating an SQLDA structure for a dynamically executed SQL program" on page 145
- "Allocating an SQLDA structure with sufficient SQLVAR entries for dynamically executed SQL statements" on page 141
- "Declaring XML host variables in embedded SQL applications" on page 71
- "Preparing a dynamically executed SQL statement using the minimum SQLDA structure" on page 140

**Related reference:**

- "SQLDA data structure" in *Administrative API Reference*

## Preparing a dynamically executed SQL statement using the minimum SQLDA structure

Use the information provided here as an example of how to allocate the minimum SQLDA structure for a statement.

**Restrictions:**

You can only allocate a smaller SQLDA structure with programming languages, such as C and C++, that support the dynamic allocation of memory.

**Procedure:**

Suppose an application declares an SQLDA structure named `minsqlda` that contains no SQLVAR entries. The SQLN field of the SQLDA describes the number of SQLVAR entries that are allocated. In this case, SQLN must be set to 0. Next, to prepare a statement from the character string `dstring` and to enter its description into `minsqlda`, issue the following SQL statement (assuming C syntax, and assuming that `minsqlda` is declared as a pointer to an SQLDA structure):

```
EXEC SQL
  PREPARE STMT INTO :*minsqlda FROM :dstring;
```

Suppose that the statement contained in `dstring` is a SELECT statement that returns 20 columns in each row. After the PREPARE statement (or a DESCRIBE statement), the SQLD field of the SQLDA contains the number of columns of the result table for the prepared SELECT statement.

The SQLVAR entries in the SQLDA are set in the following cases:

- SQLN >= SQLD and no column is either a LOB or a distinct type.

  The first SQLD SQLVAR entries are set and SQLDOUBLED is set to blank.
- SQLN >= 2*SQLD and at least one column is a LOB or a distinct type.

  2* SQLD SQLVAR entries are set and SQLDOUBLED is set to 2.
- SQLD <= SQLN < 2*SQLD and at least one column is a distinct type, but there are no LOB columns.

  The first SQLD SQLVAR entries are set and SQLDOUBLED is set to blank. If the SQLWARN bind option is YES, a warning SQLCODE +237 (SQLSTATE 01594) is issued.

The SQLVAR entries in the SQLDA are *not* set (requiring allocation of additional space and another DESCRIBE) in the following cases:

- SQLN < SQLD and no column is either a LOB or distinct type.

  No SQLVAR entries are set and SQLDOUBLED is set to blank. If the SQLWARN bind option is YES, a warning SQLCODE +236 (SQLSTATE 01005) is issued.

  Allocate SQLD SQLVAR entries for a successful DESCRIBE.
- SQLN < SQLD and at least one column is a distinct type, but there are no LOB columns.

  No SQLVAR entries are set and SQLDOUBLED is set to blank. If the SQLWARN bind option is YES, a warning SQLCODE +239 (SQLSTATE 01005) is issued.

  Allocate 2*SQLD SQLVAR entries for a successful DESCRIBE, including the names of the distinct types.
- SQLN < 2*SQLD and at least one column is a LOB.

  No SQLVAR entries are set and SQLDOUBLED is set to blank. A warning SQLCODE +238 (SQLSTATE 01005) is issued (regardless of the setting of the SQLWARN bind option).

  Allocate 2*SQLD SQLVAR entries for a successful DESCRIBE.

The SQLWARN option of the BIND command is used to control whether the DESCRIBE (or PREPARE...INTO) will return the following warnings:

- SQLCODE +236 (SQLSTATE 01005)
- SQLCODE +237 (SQLSTATE 01594)
- SQLCODE +239 (SQLSTATE 01005).

It is recommended that your application code always consider that these SQLCODE values could be returned. The warning SQLCODE +238 (SQLSTATE 01005) is always returned when there are LOB columns in the select list and there are insufficient SQLVAR entries in the SQLDA. This is the only way the application can know that the number of SQLVAR entries must be doubled because of a LOB column in the result set.

**Related tasks:**

- "Allocating an SQLDA structure for a dynamically executed SQL program" on page 145
- "Allocating an SQLDA structure with sufficient SQLVAR entries for dynamically executed SQL statements" on page 141
- "Declaring the SQLDA structure in a dynamically executed SQL program" on page 138

## Allocating an SQLDA structure with sufficient SQLVAR entries for dynamically executed SQL statements

After you determine the number of columns in the result table, allocate storage for a second, full-size SQLDA. The first SQLDA is used for input parameters and the second full-sized SQLDA is used for output parameters.

**Procedure:**

Assume that the result table contains 20 columns (none of which are LOB columns). In this situation, you must allocate a second SQLDA structure, `fulsqlda` with at least 20 SQLVAR elements (or 40 elements if the result table contains any LOBs or distinct types). For the rest of this example, assume that no LOBs or distinct types are in the result table.

When you calculate the storage requirements for SQLDA structures, include the following:

- A fixed-length header, 16 bytes in length, containing fields such as SQLN and SQLD
- A variable-length array of SQLVAR entries, of which each element is 44 bytes in length on 32-bit platforms, and 56 bytes in length on 64-bit platforms.

The number of SQLVAR entries needed for `fulsqlda` is specified in the SQLD field of `minsqlda`. Assume this value is 20. Therefore, the storage allocation required for `fulsqlda` is:

```
16 + (20 * sizeof(struct sqlvar))
```

This value represents the size of the header plus 20 times the size of each SQLVAR entry, giving a total of 896 bytes.

You can use the SQLDASIZE macro to avoid doing your own calculations and to avoid any version-specific dependencies.

**Related tasks:**

- "Allocating an SQLDA structure for a dynamically executed SQL program" on page 145
- "Declaring the SQLDA structure in a dynamically executed SQL program" on page 138
- "Preparing a dynamically executed SQL statement using the minimum SQLDA structure" on page 140

## Describing a SELECT statement in a dynamically executed SQL program

After you allocate sufficient space for the second SQLDA (in this example, called `fulsqlda`), you must code the application to describe the SELECT statement.

**Procedure:**

Code your application to perform the following steps:

1. Store the value `20` in the SQLN field of `fulsqlda` (the assumption in this example is that the result table contains 20 columns, and none of these columns are LOB columns).
2. Obtain information about the SELECT statement using the second SQLDA structure, `fulsqlda`. Two methods are available:
   - Use another PREPARE statement, specifying `fulsqlda` instead of `minsqlda`.
   - Use the DESCRIBE statement specifying `fulsqlda`.

Using the DESCRIBE statement is preferred because the costs of preparing the statement a second time are avoided. The DESCRIBE statement simply reuses information previously obtained during the prepare operation to fill in the new SQLDA structure. The following statement can be issued:

```
EXEC SQL DESCRIBE STMT INTO :fulsqlda
```

After this statement is executed, each SQLVAR element contains a description of one column of the result table.

**Related concepts:**

- "Retrieving host variable information from the SQLDA structure in embedded SQL applications" on page 138

**Related tasks:**

- "Acquiring storage to hold a row" on page 143
- "Declaring the SQLDA structure in a dynamically executed SQL program" on page 138

**Related reference:**

- "PREPARE statement" in *SQL Reference, Volume 2*
- "SELECT statement" in *SQL Reference, Volume 2*
- "DESCRIBE statement" in *SQL Reference, Volume 2*

## Acquiring storage to hold a row

Before the application can fetch a row of the result table using an SQLDA structure, the application must first allocate storage for the row.

**Procedure:**

Code your application to do the following:

1. Analyze each SQLVAR description to determine how much space is required for the value of that column.

   Note that for LOB values, when the SELECT is described, the data type given in the SQLVAR is SQL_TYP_*x*LOB. This data type corresponds to a plain LOB host variable, that is, the whole LOB will be stored in memory at one time. This will work for small LOBs (up to a few MB), but you cannot use this data type for large LOBs (say 1 GB) because the stack is unable to allocate enough memory. It will be necessary for your application to change its column definition in the SQLVAR to be either SQL_TYP_*x*LOB_LOCATOR or SQL_TYPE_*x*LOB_FILE. (Note that changing the SQLTYPE field of the SQLVAR also necessitates changing the SQLLEN field.) After changing the column definition in the SQLVAR, your application can then allocate the correct amount of storage for the new type.
2. Allocate storage for the value of that column.
3. Store the address of the allocated storage in the SQLDATA field of the SQLDA structure.

These steps are accomplished by analyzing the description of each column and replacing the content of each SQLDATA field with the address of a storage area large enough to hold any values from that column. The length attribute is determined from the SQLLEN field of each SQLVAR entry for data items that are not of a LOB type. For items with a type of BLOB, CLOB, or DBCLOB, the length attribute is determined from the SQLLONGLEN field of the secondary SQLVAR entry.

In addition, if the specified column allows nulls, the application must replace the content of the SQLIND field with the address of an indicator variable for the column.

**Related concepts:**

- "Retrieving host variable information from the SQLDA structure in embedded SQL applications" on page 138
- "Large object usage" in *Developing SQL and External Routines*

**Related tasks:**

- "Allocating an SQLDA structure for a dynamically executed SQL program" on page 145
- "Declaring the SQLDA structure in a dynamically executed SQL program" on page 138
- "Processing the cursor in a dynamically executed SQL program" on page 144
- "Transferring data in a dynamically executed SQL program using an SQLDA structure" on page 148

## Processing the cursor in a dynamically executed SQL program

After the SQLDA structure is properly allocated, the cursor associated with the SELECT statement can be opened and rows can be fetched.

**Procedure:**

To process the cursor that is associated with a SELECT statement, first open the cursor, then fetch rows by specifying the USING DESCRIPTOR clause of the FETCH statement. For example, a C application could have the following:

```
EXEC SQL OPEN pcurs
EMB_SQL_CHECK( "OPEN" ) ;
EXEC SQL FETCH pcurs USING DESCRIPTOR :*sqldaPointer
EMB_SQL_CHECK( "FETCH" ) ;
```

For a successful FETCH, you could write the application to obtain the data from the SQLDA and display the column headings. For example:

```
display_col_titles( sqldaPointer ) ;
```

After the data is displayed, you should close the cursor and release any dynamically allocated memory. For example:

```
EXEC SQL CLOSE pcurs ;
EMB_SQL_CHECK( "CLOSE CURSOR" ) ;
```

**Related concepts:**

- "Cursor types and unit of work considerations in embedded SQL applications" on page 165
- "Example of a cursor in a dynamically executed SQL application" on page 170

**Related tasks:**

- "Declaring the SQLDA structure in a dynamically executed SQL program" on page 138
- "Transferring data in a dynamically executed SQL program using an SQLDA structure" on page 148
- "Selecting multiple rows using a cursor in embedded SQL applications" on page 164
- "Positioning a cursor on a row with a particular column value" on page 167
- "Declaring and using cursors in a dynamically executed SQL application" on page 168

## Allocating an SQLDA structure for a dynamically executed SQL program

Allocate an SQLDA structure for your application so that you can use it to pass data to and from your application.

**Procedure:**

To create an SQLDA structure with C, either embed the INCLUDE SQLDA statement in the host language or include the SQLDA include file to get the structure definition. Then, because the size of an SQLDA is not fixed, the application must declare a pointer to an SQLDA structure and allocate storage for it. The actual size of the SQLDA structure depends on the number of distinct data items being passed using the SQLDA.

In the C and C++ programming language, a macro is provided to facilitate SQLDA allocation. This macro has the following format:

```
#define SQLDASIZE(n) (offsetof(struct sqlda, sqlvar) \
        + (n) × sizeof(struct sqlvar))
```

The effect of this macro is to calculate the required storage for an SQLDA with `n` SQLVAR elements.

To create an SQLDA structure with COBOL, you can either embed an INCLUDE SQLDA statement or use the COPY statement. Use the COPY statement when you want to control the maximum number of SQLVAR entries and hence the amount of storage that the SQLDA uses. For example, to change the default number of SQLVAR entries from 1489 to 1, use the following COPY statement:

```
COPY "sqlda.cbl"
  replacing --1489--
  by --1--.
```

The FORTRAN language does not directly support self-defining data structures or dynamic allocation. No SQLDA include file is provided for FORTRAN, because it is not possible to support the SQLDA as a data structure in FORTRAN. The precompiler will ignore the INCLUDE SQLDA statement in a FORTRAN program.

However, you can create something similar to a static SQLDA structure in a FORTRAN program, and use this structure wherever an SQLDA can be used. The file `sqldact.f` contains constants that help in declaring an SQLDA structure in FORTRAN.

Execute calls to SQLGADDR to assign pointer values to the SQLDA elements that require them.

The following table shows the declaration and use of an SQLDA structure with one SQLVAR element.

| Language | Example Source Code |
|---|---|

**C and C++**

```
#include
struct sqlda *outda = (struct sqlda *)malloc(SQLDASIZE(1));

/* DECLARE LOCAL VARIABLES FOR HOLDING ACTUAL DATA */
double sal = 0;
short salind = 0;

/* INITIALIZE ONE ELEMENT OF SQLDA */
memcpy( outda->sqldaid,"SQLDA   ",sizeof(outda->sqldaid));
outda->sqln = outda->sqld = 1;
outda->sqlvar[0].sqltype = SQL_TYP_NFLOAT;
outda->sqlvar[0].sqllen  = sizeof( double );.
outda->sqlvar[0].sqldata = (unsigned char *)&sal;
outda->sqlvar[0].sqlind  = (short *)&salind;
```

**COBOL**

```
       WORKING-STORAGE SECTION.
       77 SALARY          PIC S99999V99 COMP-3.
       77 SAL-IND         PIC S9(4)     COMP-5.

          EXEC SQL INCLUDE SQLDA END-EXEC

      * Or code a useful way to save unused SQLVAR entries.
      * COPY "sqlda.cbl" REPLACING --1489-- BY --1--.

            01 decimal-sqllen pic s9(4) comp-5.
            01 decimal-parts redefines decimal-sqllen.
               05 precision pic x.
               05 scale pic x.

      * Initialize one element of output SQLDA
            MOVE 1 TO SQLN
            MOVE 1 TO SQLD
            MOVE SQL-TYP-NDECIMAL TO SQLTYPE(1)

      * Length = 7 digits precision and 2 digits scale

            MOVE x"07" TO PRECISION.
            MOVE x"02" TO SCALE.
            MOVE DECIMAL-SQLLEN TO O-SQLLEN(1).
            SET SQLDATA(1) TO ADDRESS OF SALARY
            SET SQLIND(1)  TO ADDRESS OF SAL-IND
```

| Language | Example Source Code |
|---|---|
| **FORTRAN** | (see code below) |

```
      include 'sqldact.f'

      integer*2  sqlvar1
      parameter ( sqlvar1 = sqlda_header_sz + 0*sqlvar_struct_sz )

C     Declare an Output SQLDA -- 1 Variable
      character    out_sqlda(sqlda_header_sz + 1*sqlvar_struct_sz)

      character*8  out_sqldaid     ! Header
      integer*4    out_sqldabc
      integer*2    out_sqln
      integer*2    out_sqld

      integer*2    out_sqltype1    ! First Variable
      integer*2    out_sqllen1
      integer*4    out_sqldata1
      integer*4    out_sqlind1
      integer*2    out_sqlnamel1
      character*30 out_sqlnamec1

      equivalence( out_sqlda(sqlda_sqldaid_ofs), out_sqldaid )
      equivalence( out_sqlda(sqlda_sqldabc_ofs), out_sqldabc )
      equivalence( out_sqlda(sqlda_sqln_ofs), out_sqln       )
      equivalence( out_sqlda(sqlda_sqld_ofs), out_sqld       )
      equivalence( out_sqlda(sqlvar1+sqlvar_type_ofs), out_sqltype1 )
      equivalence( out_sqlda(sqlvar1+sqlvar_len_ofs), out_sqllen1   )
      equivalence( out_sqlda(sqlvar1+sqlvar_data_ofs), out_sqldata1 )
      equivalence( out_sqlda(sqlvar1+sqlvar_ind_ofs), out_sqlind1   )
      equivalence( out_sqlda(sqlvar1+sqlvar_name_length_ofs),
     +             out_sqlnamel1                                   )
      equivalence( out_sqlda(sqlvar1+sqlvar_name_data_ofs),
     +             out_sqlnamec1                                   )

C     Declare Local Variables for Holding Returned Data.
      real*8       salary
      integer*2    sal_ind

C     Initialize the Output SQLDA (Header)
      out_sqldaid  = 'OUT_SQLDA'
      out_sqldabc  = sqlda_header_sz + 1*sqlvar_struct_sz
      out_sqln     = 1
      out_sqld     = 1
C     Initialize VAR1
      out_sqltype1 = SQL_TYP_NFLOAT
      out_sqllen1  = 8
      rc = sqlgaddr( %ref(salary), %ref(out_sqldata1) )
      rc = sqlgaddr( %ref(sal_ind), %ref(out_sqlind1) )
```

**Note:** The example above was written for 32-bit FORTRAN.

In languages not supporting dynamic memory allocation, an SQLDA with the desired number of SQLVAR elements must be explicitly declared in the host language. Be sure to declare enough SQLVAR elements as determined by the needs of the application.

**Related tasks:**

- "Allocating an SQLDA structure with sufficient SQLVAR entries for dynamically executed SQL statements" on page 141
- "Transferring data in a dynamically executed SQL program using an SQLDA structure" on page 148

- “Preparing a dynamically executed SQL statement using the minimum SQLDA structure” on page 140

## Transferring data in a dynamically executed SQL program using an SQLDA structure

Greater flexibility is available when transferring data using an SQLDA than is available using lists of host variables. For example, You can use an SQLDA to transfer data that has no native host language equivalent, such as DECIMAL data in the C language.

**Procedure:**

Use the following table as a cross-reference listing that shows how the numeric values and symbolic names are related.

*Table 17. DB2 SQLDA SQL Types.* Numeric Values and Corresponding Symbolic Names

| SQL Column Type | SQLTYPE numeric value | SQLTYPE symbolic name[1] |
|---|---|---|
| `DATE` | 384/385 | `SQL_TYP_DATE / SQL_TYP_NDATE` |
| `TIME` | 388/389 | `SQL_TYP_TIME / SQL_TYP_NTIME` |
| `TIMESTAMP` | 392/393 | `SQL_TYP_STAMP / SQL_TYP_NSTAMP` |
| `n/a`[2] | 400/401 | `SQL_TYP_CGSTR / SQL_TYP_NCGSTR` |
| `BLOB` | 404/405 | `SQL_TYP_BLOB / SQL_TYP_NBLOB` |
| `CLOB` | 408/409 | `SQL_TYP_CLOB / SQL_TYP_NCLOB` |
| `DBCLOB` | 412/413 | `SQL_TYP_DBCLOB / SQL_TYP_NDBCLOB` |
| `VARCHAR` | 448/449 | `SQL_TYP_VARCHAR / SQL_TYP_NVARCHAR` |
| `CHAR` | 452/453 | `SQL_TYP_CHAR / SQL_TYP_NCHAR` |
| `LONG VARCHAR` | 456/457 | `SQL_TYP_LONG / SQL_TYP_NLONG` |
| `n/a`[3] | 460/461 | `SQL_TYP_CSTR / SQL_TYP_NCSTR` |
| `VARGRAPHIC` | 464/465 | `SQL_TYP_VARGRAPH / SQL_TYP_NVARGRAPH` |
| `GRAPHIC` | 468/469 | `SQL_TYP_GRAPHIC / SQL_TYP_NGRAPHIC` |
| `LONG VARGRAPHIC` | 472/473 | `SQL_TYP_LONGRAPH / SQL_TYP_NLONGRAPH` |
| `FLOAT` | 480/481 | `SQL_TYP_FLOAT / SQL_TYP_NFLOAT` |
| `REAL`[4] | 480/481 | `SQL_TYP_FLOAT / SQL_TYP_NFLOAT` |
| `DECIMAL`[5] | 484/485 | `SQL_TYP_DECIMAL / SQL_TYP_DECIMAL` |
| `INTEGER` | 496/497 | `SQL_TYP_INTEGER / SQL_TYP_NINTEGER` |
| `SMALLINT` | 500/501 | `SQL_TYP_SMALL / SQL_TYP_NSMALL` |
| `n/a` | 804/805 | `SQL_TYP_BLOB_FILE / SQL_TYPE_NBLOB_FILE` |
| `n/a` | 808/809 | `SQL_TYP_CLOB_FILE / SQL_TYPE_NCLOB_FILE` |
| `n/a` | 812/813 | `SQL_TYP_DBCLOB_FILE / SQL_TYPE_NDBCLOB_FILE` |
| `n/a` | 960/961 | `SQL_TYP_BLOB_LOCATOR / SQL_TYP_NBLOB_LOCATOR` |
| `n/a` | 964/965 | `SQL_TYP_CLOB_LOCATOR / SQL_TYP_NCLOB_LOCATOR` |
| `n/a` | 968/969 | `SQL_TYP_DBCLOB_LOCATOR / SQL_TYP_NDBCLOB_LOCATOR` |
| `XML` | 988/989 | `SQL_TYP_XML / SQL_TYP_XML` |

*Table 17. DB2 SQLDA SQL Types (continued).* Numeric Values and Corresponding Symbolic Names

| SQL Column Type | SQLTYPE numeric value | SQLTYPE symbolic name[1] |
|---|---|---|

**Note:** These defined types can be found in the `sql.h` include file located in the `include` sub-directory of the `sqllib` directory. (For example, `sqllib/include/sql.h` for the C programming language.)

1. For the COBOL programming language, the SQLTYPE name does not use underscore (_) but uses a hyphen (-) instead.
2. This is a null-terminated graphic string.
3. This is a null-terminated character string.
4. The difference between REAL and DOUBLE in the SQLDA is the length value (4 or 8).
5. Precision is in the first byte. Scale is in the second byte.

**Related tasks:**
- "Acquiring storage to hold a row" on page 143
- "Describing a SELECT statement in a dynamically executed SQL program" on page 142
- "Processing the cursor in a dynamically executed SQL program" on page 144

## Processing interactive SQL statements in dynamically executed sql programs

An application using dynamic SQL can be written to process arbitrary SQL statements. For example, if an application accepts SQL statements from a user, the application must be able to execute the statements without any prior knowledge of the statements. Values that are not known until execution time can be represented by parameter marks, which are denoted by question marks. Parameter marks allow for the interaction between the user and the application and is similar to host variables for static SQL statements.

**Procedure:**

Use the PREPARE and DESCRIBE statements with an SQLDA structure so that the application can determine the type of SQL statement being executed, and act accordingly.

**Related concepts:**
- "Determination of statement type in dynamically executed SQL programs" on page 149

**Related reference:**
- "PREPARE statement" in *SQL Reference, Volume 2*
- "DESCRIBE statement" in *SQL Reference, Volume 2*

## Determination of statement type in dynamically executed SQL programs

When an SQL statement is prepared, information concerning the type of statement can be determined by examining the SQLDA structure. This information is placed in the SQLDA structure either at statement preparation time with the INTO clause, or by issuing a DESCRIBE statement against a previously prepared statement.

In either case, the database manager places a value in the SQLD field of the SQLDA structure, indicating the number of columns in the result table generated by the SQL statement. If the SQLD field contains a zero (0), the statement is *not* a SELECT statement. Since the statement is already prepared, it can immediately be executed using the EXECUTE statement.

If the statement contains parameter markers, the USING clause must be specified. The USING clause can specify either a list of host variables or an SQLDA structure.

If the SQLD field is greater than zero, the statement is a SELECT statement and must be processed as described in the following sections.

**Related reference:**
- "EXECUTE statement" in *SQL Reference, Volume 2*
- "SELECT INTO statement" in *SQL Reference, Volume 2*
- "DESCRIBE statement" in *SQL Reference, Volume 2*

## Processing variable-list SELECT statements in dynamically executed SQL programs

A *varying-list* SELECT statement is one in which the number and types of columns that are to be returned are not known at precompilation time. In this case, the application does not know in advance the exact host variables that need to be declared to hold a row of the result table.

**Procedure:**

To process a variable-list SELECT statement, code your application to do the following:
1. Declare an SQLDA.
   An SQLDA structure must be used to process varying-list SELECT statements.
2. PREPARE the statement using the INTO clause.
   The application then determines whether the SQLDA structure declared has enough SQLVAR elements. If it does not, the application allocates another SQLDA structure with the required number of SQLVAR elements, and issues an additional DESCRIBE statement using the new SQLDA.
3. Allocate the SQLVAR elements.
   Allocate storage for the host variables and indicators needed for each SQLVAR. This step involves placing the allocated addresses for the data and indicator variables in each SQLVAR element.
4. Process the SELECT statement.
   A cursor is associated with the prepared statement, opened, and rows are fetched using the properly allocated SQLDA structure.

**Related tasks:**
- "Acquiring storage to hold a row" on page 143
- "Allocating an SQLDA structure with sufficient SQLVAR entries for dynamically executed SQL statements" on page 141
- "Declaring the SQLDA structure in a dynamically executed SQL program" on page 138
- "Describing a SELECT statement in a dynamically executed SQL program" on page 142
- "Preparing a dynamically executed SQL statement using the minimum SQLDA structure" on page 140

- “Processing the cursor in a dynamically executed SQL program” on page 144

### Saving SQL requests from end users

If the users of your application can issue SQL requests from the application, you may want to save these requests.

**Procedure:**

If your application allows users to save arbitrary SQL statements, you can save them in a table with a column having a data type of VARCHAR, LONG VARCHAR, CLOB, VARGRAPHIC, LONG VARGRAPHIC or DBCLOB. Note that the VARGRAPHIC, LONG VARGRAPHIC, and DBCLOB data types are only available in double-byte character set (DBCS) and Extended UNIX Code (EUC) environments.

You must save the source SQL statements, not the prepared versions. This means that you must retrieve and then prepare each statement before executing the version stored in the table. In essence, your application prepares an SQL statement from a character string and executes this statement dynamically.

**Related concepts:**

- “Example of parameter markers in a dynamically executed SQL program” on page 154

**Related tasks:**

- “Providing variable input to dynamically executed SQL statement using parameter markers” on page 153

## Executing XQuery expressions in embedded SQL applications

You can store XML data in your tables and use embedded SQL applications to access the XML columns using XQuery expressions. To access XML data, use XML host variables instead of casting the data to character or binary data types. If you do not make use of XML host variables, the best alternative for accessing XML data is with FOR BIT DATA or BLOB data types to avoid codepage conversion.

**Prerequisites:**

- Declare XML host variables within your embedded SQL applications.

**Restrictions:**

- An XML type must be used to retrieve XML values in a static SQL `SELECT INTO` statement.
- If a CHAR, VARCHAR, CLOB, or BLOB host variable is used for input where an XML value is expected, the value will be subject to an XMLPARSE function operation with default whitespace (STRIP) handling. Otherwise, an XML host variable is required.

**Procedure:**

To execute XQuery expressions in embedded SQL application directly, prepend the expression with the ″XQUERY″ keyword. For static SQL use the XMLQUERY function. When the XMLQUERY function is called, the XQuery expression is not prefixed by ″XQUERY″.

**Example 1: Executing XQuery expressions directly in C and C++ dynamic SQL by prepending the ″XQUERY″ keyword:**

In C and C++ applications, XQuery expressions can be executed in the following way:

```
EXEC SQL INCLUDE SQLCA;
EXEC SQL BEGIN DECLARE SECTION;
  char stmt[16384];
  SQL TYPE IS XML AS BLOB( 10K ) xmlblob;
EXEC SQL END DECLARE SECTION;

sprintf( stmt, "XQUERY (10, xs:integer(1) to xs:integer(4))" );

EXEC SQL PREPARE s1 FROM :stmt;
EXEC SQL DECLARE c1 CURSOR FOR s1;
EXEC SQL OPEN c1;

while( sqlca.sqlcode == SQL_RC_OK )
{
  EXEC SQL FETCH c1 INTO :xmlblob;
  /* Display results */
}

EXEC SQL CLOSE c1;
EXEC SQL COMMIT;
```

**Example 2: Executing XQuery expressions in static SQL using the XMLQUERY function:**

SQL statements containing the XMLQUERY function can be prepared statically, as follows:

```
EXEC SQL INCLUDE SQLCA;
EXEC SQL BEGIN DECLARE SECTION;
  SQL TYPE IS XML AS BLOB( 10K ) xmlblob;
EXEC SQL END DECLARE SECTION;

EXEC SQL DECLARE C1 CURSOR FOR SELECT XMLQUERY( '(10, xs:integer(1) to
 xs:integer(4))' RETURNING SEQUENCE BY REF) from SYSIBM.SYSDUMMY1;

EXEC SQL OPEN c1;

while( sqlca.sqlcode == SQL_RC_OK )
{
  EXEC SQL FETCH c1 INTO :xmlblob;
  /* Display results */
}

EXEC SQL CLOSE c1;
EXEC SQL COMMIT;
```

**Example 3: Executing XQuery expressions in COBOL embedded SQL applications:**

In COBOL applications, XQuery expressions can be executed in the following way:

```
EXEC SQL BEGIN DECLARE SECTION END-EXEC.
     01 stmt pic x(80).
   01 xmlBuff USAGE IS SQL TYPE IS XML AS BLOB (10K).
EXEC SQL END DECLARE SECTION END-EXEC.

MOVE "XQUERY (10, xs:integer(1) to xs:integer(4))" TO stmt.
EXEC SQL PREPARE s1 FROM :stmt END-EXEC.
EXEC SQL DECLARE c1 CURSOR FOR s1 END-EXEC.
EXEC SQL OPEN c1 USING :host-var END-EXEC.
```

```
*Call the FETCH and UPDATE loop.
Perform Fetch-Loop through End-Fetch-Loop
   until SQLCODE does not equal 0.

EXEC SQL CLOSE c1 END-EXEC.
EXEC SQL COMMIT END-EXEC.

Fetch-Loop Section.
    EXEC SQL FETCH c1 INTO :xmlBuff END-EXEC.
    if SQLCODE not equal 0
       go to End-Fetch-Loop.
* Display results
End-Fetch-Loop. exit.
```

**Related concepts:**

- "Binary storage of variable values using the FOR BIT DATA clause in C and C++ embedded SQL applications" on page 101
- "Binary storage of variable values using the FOR BIT DATA clause in COBOL embedded SQL applications" on page 117
- "Data types that map to SQL data types in embedded SQL applications" on page 53
- "Example: Referencing XML host variables in embedded SQL applications" on page 76
- "XML serialization" in *XML Guide*

**Related tasks:**

- "Declaring XML host variables in embedded SQL applications" on page 71

**Related reference:**

- "Recommendations for developing embedded SQL applications with XML and XQuery" on page 27

## Providing variable input to dynamically executed SQL using parameter markers

### Providing variable input to dynamically executed SQL statement using parameter markers

A dynamic SQL statement cannot contain host variables, because host variable information (data type and length) is available only during application precompilation. At execution time, the host variable information is not available.

In dynamic SQL, parameter markers are used instead of host variables. Parameter markers are indicated by a question mark (?), and indicate where a host variable is to be substituted inside an SQL statement.

**Procedure:**

Assume that your application uses dynamic SQL, and that you want to be able to perform a DELETE. A character string containing a parameter marker might look like the following:

```
DELETE FROM TEMPL WHERE EMPNO = ?
```

When this statement is executed, a host variable or SQLDA structure is specified by the USING clause of the EXECUTE statement. The contents of the host variable are used when the statement executes.

The parameter marker takes on an assumed data type and length that is dependent on the context of its use inside the SQL statement. If the data type of a parameter marker is not obvious from the context of the statement in which it is used, use a CAST to specify the type. Such a parameter marker is considered a *typed parameter marker*. Typed parameter markers will be treated like a host variable of the given type. For example, the statement `SELECT ? FROM SYSCAT.TABLES` is not valid because DB2 does not know the type of the result column. However, the statement `SELECT CAST(? AS INTEGER) FROM SYSCAT.TABLES` is valid because the cast indicates that the parameter marker represents an INTEGER, so DB2 knows the type of the result column.

If the SQL statement contains more than one parameter marker, the USING clause of the EXECUTE statement must either specify a list of host variables (one for each parameter marker), or it must identify an SQLDA that has an SQLVAR entry for each parameter marker. (Note that for LOBs, there are two SQLVAR entries per parameter marker.) The host variable list or SQLVAR entries are matched according to the order of the parameter markers in the statement, and they must have compatible data types.

**Note:** Using a parameter marker with dynamic SQL is like using host variables with static SQL. In either case, the optimizer does not use distribution statistics, and possibly may not choose the best access plan.

The rules that apply to parameter markers are described with the PREPARE statement.

**Related concepts:**

- "Example of parameter markers in a dynamically executed SQL program" on page 154

**Related reference:**

- "PREPARE statement" in *SQL Reference, Volume 2*
- "EXECUTE statement" in *SQL Reference, Volume 2*
- "Casting between data types" in *SQL Reference, Volume 1*

## Example of parameter markers in a dynamically executed SQL program

The following examples show how to use parameter markers in a dynamic SQL program:

- C and C++ (**dbuse.sqc/dbuse.sqC**)

  The function `DynamicStmtWithMarkersEXECUTEusingHostVars()` in the C-language sample **dbuse.sqc** shows how to perform a delete using a parameter marker with a host variable:

```
EXEC SQL BEGIN DECLARE SECTION;
  char hostVarStmt1[50];
  short hostVarDeptnumb;
EXEC SQL END DECLARE SECTION;

/* prepare the statement with a parameter marker */
strcpy(hostVarStmt1, "DELETE FROM org WHERE deptnumb = ?");
EXEC SQL PREPARE Stmt1 FROM :hostVarStmt1;
```

```
/* execute the statement for hostVarDeptnumb = 15 */
hostVarDeptnumb = 15;
EXEC SQL EXECUTE Stmt1 USING :hostVarDeptnumb;
```

- COBOL (**varinp.sqb**)

  The following example is from the COBOL sample **varinp.sqb**, and shows how to use a parameter marker in search and update conditions:

```
   EXEC SQL BEGIN DECLARE SECTION END-EXEC.
01 pname            pic x(10).
01 dept             pic s9(4) comp-5.
01 st               pic x(127).
01 parm-var         pic x(5).
   EXEC SQL END DECLARE SECTION END-EXEC.

   move "SELECT name, dept FROM staff
-        "   WHERE job = ? FOR UPDATE OF job" to st.
   EXEC SQL PREPARE s1 FROM :st END-EXEC.

   EXEC SQL DECLARE c1 CURSOR FOR s1 END-EXEC.

   move "Mgr" to parm-var.
   EXEC SQL OPEN c1 USING :parm-var END-EXEC

   move "Clerk" to parm-var.
   move "UPDATE staff SET job = ? WHERE CURRENT OF c1" to st.
   EXEC SQL PREPARE s2 from :st END-EXEC.

* call the FETCH and UPDATE loop.
   perform Fetch-Loop thru End-Fetch-Loop
      until SQLCODE not equal 0.

   EXEC SQL CLOSE c1 END-EXEC.
```

**Related concepts:**
- "Error message retrieval in embedded SQL applications" on page 174

**Related samples:**
- "dbuse.out -- HOW TO USE A DATABASE (C)"
- "dbuse.sqc -- How to use a database (C)"
- "dbuse.out -- HOW TO USE A DATABASE (C++)"
- "dbuse.sqC -- How to use a database (C++)"

## Calling stored procedures in embedded SQL applications

### Calling stored procedures in embedded SQL applications

Stored procedures can be called from embedded SQL applications by formulating and executing the CALL statement with an appropriate procedure reference and parameters. The CALL statement can be executed either statically or dynamically within embedded SQL applications. However, for each programming language there are different methods to execute this command. No matter which host language, each host variable used in the stored procedure must be declared to match the data type which is required.

Client applications and the calling of routines exchange information with procedures through parameters and result sets. The parameters for procedures are defined by the direction the data is traveling (the parameter mode).

There are three types of parameters for procedures:

- IN parameters: data passed to the procedure.
- OUT parameters: data returned by the procedure.
- INOUT parameters: data passed to the procedure that is, during procedure execution, replaced by data to be returned from the procedure.

The mode of parameters and their data types are defined when a procedure is registered with the CREATE PROCEDURE statement.

**Related concepts:**

- "Calling stored procedures from REXX" on page 156
- "Calling stored procedures in C and C++ embedded SQL applications" on page 156

## Calling stored procedures in C and C++ embedded SQL applications

**Calling stored procedures in C and C++ embedded SQL applications:**

DB2 supports the use of input, output, and input and output parameters in SQL procedures. The keywords IN, OUT, and INOUT in the CREATE PROCEDURE statement indicate the mode or intended use of the parameter. IN and OUT parameters are passed by value, and INOUT parameters are passed by reference.

When working with C and C++ applications, a stored procedure, INOUT_PARAM, can be called using the following statement:

```
   EXEC SQL CALL INOUT_PARAM(:inout_median:medianind, :out_sqlcode:codeind,
                             :out_buffer:bufferind);
```

where inout_median, out_sqlcode, and out_buffer are host variables and medianind, codeind, and bufferind are null indicator variables.

**Note:** Stored procedures can also be called dynamically by preparing a CALL statement.

**Related concepts:**

- "Routines: Procedures" in *Developing SQL and External Routines*

**Related reference:**

- "SQL procedure samples" in *Samples Topics*

## Calling stored procedures from REXX

**Calling stored procedures from REXX:** The stored procedure can be written in any language supported on that server, except for REXX on AIX systems. (Client applications may be written in REXX on AIX systems, but, as with other languages, they cannot call a stored procedure written in REXX on AIX.)

**Related concepts:**

- "Stored procedure calls in REXX" on page 156

**Stored procedure calls in REXX:** The CALL statement allows a client application to pass data to, and receive data from, a server stored procedure. The interface for both input and output data is a list of host variables. Because REXX generally determines the type and size of host variables based on their content, any output-only variables passed to CALL should be initialized with *dummy* data similar in type and size to the expected output.

Data can also be passed to stored procedures through SQLDA REXX variables, using the USING DESCRIPTOR syntax of the CALL statement. The following list shows how the SQLDA is set up. In the list, ':value' is the stem of a REXX host variable that contains the values needed for the application. For the DESCRIPTOR, 'n' is a numeric value indicating a specific *sqlvar* element of the SQLDA. The numbers on the right refer to the notes following the list.

**Client-side REXX SQLDA for Stored Procedures using the CALL Statement:**

USING DESCRIPTOR

- `:value.SQLD` 1
- `:value.n.SQLTYPE` 1
- `:value.n.SQLLEN` 1
- `:value.n.SQLDATA` 1 2
- `:value.n.SQLDIND` 1 2

**Notes:**

1. Before invoking the stored procedure, the client application must initialize the REXX variable with appropriate data.

   When the SQL CALL statement is executed, the database manager allocates storage and retrieves the value of the REXX variable from the REXX variable pool. For an SQLDA used in a CALL statement, the database manager allocates storage for the SQLDATA and SQLIND fields based on the SQLTYPE and SQLLEN values.

   In the case of a REXX stored procedure (that is, the procedure being called is itself written in Windows-based REXX), the data passed by the client from either type of CALL statement or the DARI API is placed in the REXX variable pool at the database server using the following predefined names:

   **SQLRIDA**
   Predefined name for the REXX input SQLDA variable

   **SQLRODA**
   Predefined name for the REXX output SQLDA variable
2. When the stored procedure terminates, the database manager also retrieves the value of the variables from the stored procedure. The values are returned to the client application and placed in the client's REXX variable pool.

**Related concepts:**

- "Client considerations for calling stored procedures in REXX" on page 159
- "Retrieval of precision and SCALE values from SQLDA decimal fields" on page 159
- "Server considerations for calling stored procedures in REXX" on page 159

**Related reference:**

- "CALL statement" in *SQL Reference, Volume 2*

**API Syntax for REXX:** Use the SQLDBS routine to call DB2 APIs with the following syntax:

```
CALL SQLDBS 'command string'
```

If a DB2® API you want to use cannot be called using the SQLDBS routine, you may still call the API by calling the DB2 command line processor (CLP) from within the REXX application. However, because the DB2 CLP directs output either to the standard output device or to a specified file, your REXX application cannot directly access the output from the called DB2 API, nor can it easily make a determination as to whether the called API is successful or not. The SQLDB2 API provides an interface to the DB2 CLP that provides direct feedback to your REXX application on the success or failure of each called API by setting the compound REXX variable, SQLCA, after each call.

You can use the SQLDB2 routine to call DB2 APIs using the following syntax:

```
   CALL SQLDB2 'command string'
```

where `'command string'` is a string that can be processed by the command-line processor (CLP).

Calling a DB2 API using SQLDB2 is equivalent to calling the CLP directly, except for the following:

- The call to the CLP executable is replaced by the call to SQLDB2 (all other CLP options and parameters are specified the same way).
- The REXX compound variable SQLCA is set after calling the SQLDB2 but is not set after calling the CLP executable.
- The default display output of the CLP is set to off when you call SQLDB2, whereas the display is set to on output when you call the CLP executable. Note that you can turn the display output of the CLP to on by passing the `+o` or the `-o-` option to the SQLDB2.

Because the only REXX variable that is set after you call SQLDB2 is the SQLCA, you only use this routine to call DB2 APIs that do not return any data other than the SQLCA and that are not currently implemented through the SQLDBS interface. Thus, only the following DB2 APIs are supported by SQLDB2:

- Activate Database
- Add Node
- Bind for DB2 Version 1(1) (2)
- Bind for DB2 Version 2 or 5(1)
- Create Database at Node
- Drop Database at Node
- Drop Node Verify
- Deactivate Database
- Deregister
- Load(3)
- Load Query
- Precompile Program(1)
- Rebind Package(1)
- Redistribute Database Partition Group
- Register
- Start Database Manager
- Stop Database Manager

**Notes on DB2 APIs Supported by SQLDB2:**

1. These commands require a CONNECT statement through the SQLDB2 interface. Connections using the SQLDB2 interface are not accessible to the SQLEXEC interface and connections using the SQLEXEC interface are not accessible to the SQLDB2 interface.
2. Is supported on Windows®-based platforms through the SQLDB2 interface.

3. The optional output parameter, pLoadInfoOut for the Load API is not returned to the application in REXX.

**Note:** Although the SQLDB2 routine is intended to be used only for the DB2 APIs listed above, it can also be used for other DB2 APIs that are not supported through the SQLDBS routine. Alternatively, the DB2 APIs can be accessed through the CLP from within the REXX application.

**Related concepts:**
- "Embedded SQL statements in REXX applications" on page 9

**Client considerations for calling stored procedures in REXX:** When using host variables in the CALL statement, initialize each host variable to a value that is type compatible with any data that is returned to the host variable from the server procedure. You should perform this initialization even if the corresponding indicator is negative.

When using descriptors, SQLDATA must be initialized and contain data that is type compatible with any data that is returned from the server procedure. You should perform this initialization even if the SQLIND field contains a negative value.

**Related reference:**
- "Supported SQL data types in REXX embedded SQL applications" on page 64

**Server considerations for calling stored procedures in REXX:** Ensure that all the SQLDATA fields and SQLIND (if it is a nullable type) of the predefined output sqlda SQLRODA are initialized. For example, if SQLRODA.SQLD is 2, the following fields must contain some data (even if the corresponding indicators are negative and the data is not passed back to the client):
- SQLRODA.1.SQLDATA
- SQLRODA.2.SQLDATA

**Related concepts:**
- "Stored procedure calls in REXX" on page 156

**Retrieval of precision and SCALE values from SQLDA decimal fields:** To retrieve the precision and scale values for decimal fields from the SQLDA structure returned by the database manager, use the sqllen.scale and sqllen.precision values when you initialize the SQLDA output in your REXX program. For example:

```
.
.
.
/* INITIALIZE ONE ELEMENT OF OUTPUT SQLDA */
io_sqlda.sqld = 1
io_sqlda.1.sqltype = 485          /* DECIMAL DATA TYPE */
io_sqlda.1.sqllen.scale  = 2      /* DIGITS RIGHT OF DECIMAL POINT */
io_sqlda.1.sqllen.precision  = 7  /* WIDTH OF DECIMAL  */
io_sqlda.1.sqldata = 00000.00     /* HELPS DEFINE DATA FORMAT */
io_sqlda.1.sqlind = -1            /* NO INPUT DATA */
.
.
.
```

**Related concepts:**

- "Retrieving host variable information from the SQLDA structure in embedded SQL applications" on page 138
- "Stored procedure calls in REXX" on page 156

## Reading and scrolling through result sets in embedded SQL applications

### Reading and scrolling through result sets in embedded SQL applications

One of the most common tasks of an embedded SQL application program is to retrieve data. This task is done using the *select-statement*, which is a form of query that searches for rows of tables in the database that meet specified search conditions. If such rows exist, the data is retrieved and put into specified variables in the host program, where it can be used for whatever it was designed to do.

**Note:** Embedded SQL applications can call stored procedures with any of the supported stored procedure implementations and can retrieve output and input-output parameter values, however embedded SQL applications cannot read and scroll through result sets returned by stored procedures.

After you have written a select-statement, you code the SQL statements that define how information will be passed to your application.

You can think of the result of a select-statement as being a table having rows and columns, much like a table in the database. If only one row is returned, you can deliver the results directly into host variables specified by the SELECT INTO statement.

If more than one row is returned, you must use a *cursor* to fetch them one at a time. A cursor is a named control structure used by an application program to point to a specific row within an ordered set of rows.

**Related concepts:**
- "Example of a cursor in a statically executed SQL application" on page 169
- "Host Variables in embedded SQL applications" on page 66

**Related tasks:**
- "Declaring and using cursors in statically executed SQL applications" on page 167
- "Declaring host variables in embedded SQL applications" on page 68
- "Identifying null SQL values with null indicator variables" on page 73
- "Referencing host variables in embedded SQL applications" on page 76
- "Selecting multiple rows using a cursor in embedded SQL applications" on page 164

**Related samples:**
- "spclient.sqc -- Call various stored procedures (C)"

### Scrolling through previously retrieved data in embedded SQL applications

When an application retrieves data from the database, the FETCH statement allows it to scroll forward through the data, however, there is no SQL statement that allows scrolling backwards through the result set, (equivalent to a backward FETCH). DB2 CLI and the DB2 Universal JDBC Driver, however, do support a backward FETCH through read-only scrollable cursors.

**Procedure:**

For embedded SQL applications, you can use the following techniques to scroll through data that has been retrieved:

- Keep a copy of the data that has been fetched in the application memory and scroll through it by some programming technique.
- Use SQL to retrieve the data again, typically by using a second SELECT statement.

**Related tasks:**

- "Keeping a copy of fetched data in embedded SQL applications" on page 161
- "Retrieving fetched data a second time in embedded SQL applications" on page 162

**Related reference:**

- "Cursor positioning rules for SQLFetchScroll() (CLI)" in *Call Level Interface Guide and Reference, Volume 2*
- "SQLFetchScroll function (CLI) - Fetch rowset and return data for all bound columns" in *Call Level Interface Guide and Reference, Volume 2*

## Keeping a copy of fetched data in embedded SQL applications

In some situations, it may be useful to maintain a copy of data that is fetched by the application.

**Procedure:**

To keep a copy of the data, your application can do the following:

- Save the fetched data in virtual storage.
- Write the data to a temporary file (if the data does not fit in virtual storage). One effect of this approach is that a user, scrolling backward, always sees exactly the same data that was fetched, even if the data in the database was changed in the interim by a transaction.
- Using an isolation level of repeatable read, the data you retrieve from a transaction can be retrieved again by closing and opening a cursor. Other applications are prevented from updating the data in your result set. Isolation levels and locking can affect how users update data.

**Related concepts:**

- "Row order differences in result tables" on page 163
- "Isolation levels and performance" in *Performance Guide*

**Related tasks:**

- "Retrieving fetched data a second time in embedded SQL applications" on page 162
- "Specifying the isolation level" in *Performance Guide*

## Retrieving fetched data a second time in embedded SQL applications

The technique that you use to retrieve data a second time depends on the order in which you want to see the data again.

**Procedure:**

You can retrieve data a second time by using any of the following methods:

- Retrieve data from the beginning

  To retrieve the data again from the beginning of the result table, close the active cursor and reopen it. This action positions the cursor at the beginning of the result table. But, unless the application holds locks on the table, others may have changed it, so what had been the first row of the result table may no longer be.
- Retrieve data from the middle

  To retrieve data a second time from somewhere in the middle of the result table, execute a second SELECT statement and declare a second cursor on the statement. For example, suppose the first SELECT statement was:

  ```
  SELECT * FROM DEPARTMENT
    WHERE LOCATION = 'CALIFORNIA'
    ORDER BY DEPTNO
  ```

  Now, suppose that you want to return to the rows that start with `DEPTNO = 'M95'` and fetch sequentially from that point. Code the following:

  ```
  SELECT * FROM DEPARTMENT
    WHERE LOCATION = 'CALIFORNIA'
    AND DEPTNO >= 'M95'
    ORDER BY DEPTNO
  ```

  This statement positions the cursor where you want it.
- Retrieve data in reverse order

  Ascending ordering of rows is the default. If there is only one row for each value of `DEPTNO`, then the following statement specifies a unique ascending ordering of rows:

  ```
  SELECT * FROM DEPARTMENT
    WHERE LOCATION = 'CALIFORNIA'
    ORDER BY DEPTNO
  ```

  To retrieve the same rows in reverse order, specify that the order is descending, as in the following statement:

  ```
  SELECT * FROM DEPARTMENT
    WHERE LOCATION = 'CALIFORNIA'
    ORDER BY DEPTNO DESC
  ```

  A cursor on the second statement retrieves rows in exactly the opposite order from a cursor on the first statement. Order of retrieval is guaranteed only if the first statement specifies a unique ordering sequence.

  For retrieving rows in reverse order, it can be useful to have two indexes on the `DEPTNO` column, one in ascending order, and the other in descending order.

**Related concepts:**

- "Row order differences in result tables" on page 163
- "Cursor types and unit of work considerations in embedded SQL applications" on page 165

**Related tasks:**

- “Selecting multiple rows using a cursor in embedded SQL applications” on page 164
- “Positioning a cursor on a row with a particular column value” on page 167

## Row order differences in result tables

The rows of multiple result tables for the same SELECT statement might not be displayed in the same order. The database manager does not consider the order of rows as significant unless the SELECT statement uses ORDER BY. Thus, if there are several rows with the same `DEPTNO` value, the second SELECT statement can retrieve them in a different order from the first. The only guarantee is that they will all be in order by department number, as demanded by the clause ORDER BY DEPTNO.

The difference in ordering could occur even if you were to execute the same SQL statement, with the same host variables, a second time. For example, the statistics in the catalog could be updated between executions, or indexes could be created or dropped. You could then execute the SELECT statement again.

The ordering is more likely to change if the second SELECT has a predicate that the first did not have; the database manager could choose to use an index on the new predicate. For example, it could choose an index on `LOCATION` for the first statement in the example, and an index on `DEPTNO` for the second. Because rows are fetched in order by the index key, the second order need not be the same as the first.

Again, executing two similar SELECT statements can produce a different ordering of rows, even if no statistics change and no indexes are created or dropped. In the example, if there are many different values of `LOCATION`, the database manager could choose an index on `LOCATION` for both statements. Yet changing the value of `DEPTNO` in the second statement to the following, could cause the database manager to choose an index on `DEPTNO`:

```
SELECT * FROM DEPARTMENT
  WHERE LOCATION = 'CALIFORNIA'
  AND DEPTNO >= 'Z98'
  ORDER BY DEPTNO
```

Because of the subtle relationships between the form of an SQL statement and the values in this statement, never assume that two different SQL statements will return rows in the same order unless the order is uniquely determined by an ORDER BY clause.

**Related concepts:**

- “Reading and scrolling through result sets in embedded SQL applications” on page 160

**Related tasks:**

- “Retrieving fetched data a second time in embedded SQL applications” on page 162

## Updating previously retrieved data in embedded SQL applications

To scroll backward and update data that was retrieved previously, you can use a combination of the techniques that are used to scroll through previously retrieved data and to update retrieved data.

**Procedure:**

To update previously retrieved data, you can do one of two things:

- If you have a second cursor on the data to be updated and the SELECT statement uses none of the restricted elements, you can use a cursor-controlled UPDATE statement. Name the second cursor in the WHERE CURRENT OF clause.
- In other cases, use UPDATE with a WHERE clause that names all the values in the row or specifies the primary key of the table. You can execute one statement many times with different values of the variables.

**Related concepts:**

- "Example of a cursor in a statically executed SQL application" on page 169
- "Example of a cursor in a dynamically executed SQL application" on page 170

**Related tasks:**

- "Scrolling through previously retrieved data in embedded SQL applications" on page 160
- "Updating and deleting retrieved data in statically executed SQL application" on page 171
- "Declaring and using cursors in statically executed SQL applications" on page 167
- "Declaring and using cursors in a dynamically executed SQL application" on page 168

## Selecting multiple rows using a cursor

**Selecting multiple rows using a cursor in embedded SQL applications:** To allow an application to retrieve a set of rows, SQL uses a mechanism called a *cursor*.

To help understand the concept of a cursor, assume that the database manager builds a *result table* to hold all the rows retrieved by executing a SELECT statement. A cursor makes rows from the result table available to an application by identifying or pointing to a *current row* of this table. When a cursor is used, an application can retrieve each row sequentially from the result table until an end of data condition, that is, the NOT FOUND condition, SQLCODE +100 (SQLSTATE 02000) is reached. The set of rows obtained as a result of executing the SELECT statement can consist of zero, one, or more rows, depending on the number of rows that satisfy the search condition.

**Procedure:**

The steps involved in processing a cursor are as follows:

1. Specify the cursor using a DECLARE CURSOR statement.
2. Perform the query and build the result table using the OPEN statement.
3. Retrieve rows one at a time using the FETCH statement.
4. Process rows with the DELETE or UPDATE statements (if required).
5. Terminate the cursor using the CLOSE statement.

An application can use several cursors concurrently. Each cursor requires its own set of DECLARE CURSOR, OPEN, CLOSE, and FETCH statements.

**Related concepts:**

- “Example of a cursor in a statically executed SQL application” on page 169

**Related tasks:**
- “Declaring and using cursors in statically executed SQL applications” on page 167
- “Declaring and using cursors in a dynamically executed SQL application” on page 168

**Cursor types and unit of work considerations in embedded SQL applications:** Cursors fall into three categories:

**Read only**
The rows in the cursor can only be read, not updated. Read-only cursors are used when an application will only read data, not modify it. A cursor is considered read only if it is based on a read-only select-statement. See the description of how to update and retrieve data for the rules for select-statements that define non-updatable result tables.

There can be performance advantages for read-only cursors. If a cursor is determined to be read only and uses a repeatable read isolation level, repeatable read locks are still gathered and maintained on system tables needed by the unit of work. Therefore, it is important for applications to periodically issue COMMIT statements, even for read only cursors.

**Updatable**
The rows in the cursor can be updated. Updatable cursors are used when an application modifies data as the rows in the cursor are fetched. The specified query can only refer to one table or view. The query must also include the FOR UPDATE clause, naming each column that will be updated (unless the LANGLEVEL MIA precompile option is used).

**Ambiguous**
The cursor cannot be determined to be updatable or read only from its definition or context. This situation can happen when a dynamic SQL statement is encountered that could be used to change a cursor that would otherwise be considered read-only.

An ambiguous cursor is treated as read only if the BLOCKING ALL option is specified when precompiling or binding. Otherwise, the cursor is considered updatable.

**Note:** Cursors processed dynamically are always ambiguous.

Depending on how the cursors are declared, the actions of a COMMIT or ROLLBACK operation vary for cursors based on following declarations:

**WITH HOLD option**

If an application completes a unit of work by issuing a COMMIT statement, *all open cursors*, except those declared using the WITH HOLD option, are automatically closed by the database manager.

A cursor that is declared WITH HOLD maintains the resources it accesses across multiple units of work. The exact effect of declaring a cursor WITH HOLD depends on how the unit of work ends:
- If the unit of work ends with a COMMIT statement, open cursors defined WITH HOLD remain OPEN. The cursor is positioned before the next logical row of the result table. In addition, prepared statements referencing OPEN cursors defined WITH HOLD are retained. Only

FETCH and CLOSE requests associated with a particular cursor are valid immediately following the COMMIT. UPDATE WHERE CURRENT OF and DELETE WHERE CURRENT OF statements are valid only for rows fetched within the same unit of work.

**Note:** If a package is rebound during a unit of work, all held cursors are closed.

- If the unit of work ends with a ROLLBACK statement, all open cursors are closed, all locks acquired during the unit of work are released, and all prepared statements that are dependent on work done in that unit are dropped.

For example, suppose that the TEMPL table contains 1 000 entries. You want to update the salary column for all employees, and you expect to issue a COMMIT statement every time you update 100 rows.

1. Declare the cursor using the WITH HOLD option:
```
EXEC SQL DECLARE EMPLUPDT CURSOR WITH HOLD FOR
  SELECT EMPNO, LASTNAME, PHONENO, JOBCODE, SALARY
  FROM TEMPL FOR UPDATE OF SALARY
```
2. Open the cursor and fetch data from the result table one row at a time:
```
EXEC SQL OPEN EMPLUPDT
  .
  .
  .

EXEC SQL FETCH EMPLUPDT
  INTO :upd_emp, :upd_lname, :upd_tele, :upd_jobcd, :upd_wage,
```
3. When you want to update or delete a row, use an UPDATE or DELETE statement using the WHERE CURRENT OF option. For example, to update the current row, your program can issue:
```
EXEC SQL UPDATE TEMPL SET SALARY = :newsalary
  WHERE CURRENT OF EMPLUPDT
```
4. After a COMMIT is issued, you must issue a FETCH before you can update another row.

You should include code in your application to detect and handle an SQLCODE -501 (SQLSTATE 24501), which can be returned on a FETCH or CLOSE statement if your application either:

- Uses cursors declared WITH HOLD
- Executes more than one unit of work and leaves a WITH HOLD cursor open across the unit of work boundary (COMMIT WORK).

If an application invalidates its package by dropping a table on which it is dependent, the package gets rebound dynamically. If this is the case, an SQLCODE -501 (SQLSTATE 24501) is returned for a FETCH or CLOSE statement because the database manager closes the cursor. The way to handle an SQLCODE -501 (SQLSTATE 24501) in this situation depends on whether you want to fetch rows from the cursor:

- If you want to fetch rows from the cursor, open the cursor, then run the FETCH statement. Note, however, that the OPEN statement repositions the cursor to the start. The previous position held at the COMMIT WORK statement is lost.
- If you do not want to fetch rows from the cursor, do not issue any more SQL requests against the cursor.

**WITH RELEASE option**

When an application closes a cursor using the WITH RELEASE option, DB2 attempts to release all READ locks that the cursor still holds. The cursor will only continue to hold WRITE locks. If the application closes the cursor without using the RELEASE option, the READ and WRITE locks will be released when the unit of work completes.

**Related tasks:**
- "Declaring and using cursors in statically executed SQL applications" on page 167
- "Selecting multiple rows using a cursor in embedded SQL applications" on page 164

**Positioning a cursor on a row with a particular column value:** If you need to position the cursor at the end of a table, you can use an SQL statement to position it.

**Procedure:**

Use either of the following examples as a method for positioning a cursor through sorting:
- The database manager does not guarantee an order to data stored in a table; therefore, the end of a table is not defined. However, order is defined on the result of an SQL statement:
```
SELECT * FROM DEPARTMENT
  ORDER BY DEPTNO DESC
```
- The following statement positions the cursor at the row with the highest `DEPTNO` value:
```
SELECT * FROM DEPARTMENT
  WHERE DEPTNO =
  (SELECT MAX(DEPTNO) FROM DEPARTMENT)
```

Note, however, that if several rows have the same value, the cursor is positioned on the first of them.

**Related tasks:**
- "Processing the cursor in a dynamically executed SQL program" on page 144
- "Retrieving fetched data a second time in embedded SQL applications" on page 162

**Declaring and using cursors in statically executed SQL applications:** The DECLARE CURSOR statement must be used to declare a cursor, specify a name for the cursor, and the query that defines the result set associated with the cursor. The name specified is referred to in subsequent OPEN, FETCH, and CLOSE statements. The query is any valid select statement.

**Restrictions:**

The declaration of a cursor can take place anywhere within and embedded SQL application, however it must appear before any references to it in other SQL statements.

**Procedure:**

Use the DECLARE statement to define the cursor. The following table provides examples for the supported host languages:

*Table 18. Cursor Declarations by Host Language*

| Language | Example Source Code |
|---|---|
| C and C++ | `EXEC SQL DECLARE C1 CURSOR FOR`<br>`  SELECT PNAME, DEPT FROM STAFF`<br>`  WHERE JOB=:`*`host_var`*`;` |
| COBOL | `EXEC SQL DECLARE C1 CURSOR FOR`<br>`  SELECT NAME, DEPT FROM STAFF`<br>`    WHERE JOB=:`*`host-var`*` END-EXEC.` |
| FORTRAN | ` EXEC SQL DECLARE C1 CURSOR FOR`<br>`+  SELECT NAME, DEPT FROM STAFF`<br>`+  WHERE JOB=:`*`host_var`* |

**Related concepts:**

- "Cursor types and unit of work considerations in embedded SQL applications" on page 165

**Related tasks:**

- "Selecting multiple rows using a cursor in embedded SQL applications" on page 164

**Related reference:**

- "DECLARE CURSOR statement" in *SQL Reference, Volume 2*
- "FETCH statement" in *SQL Reference, Volume 2*
- "OPEN statement" in *SQL Reference, Volume 2*

**Declaring and using cursors in a dynamically executed SQL application:**
Processing a cursor dynamically is nearly identical to processing it using static SQL. When a cursor is declared, it is associated with a query. By using the FETCH statement the cursor is positioned on the next row of the result table and assigns the values of that row to host variables.

In static SQL, the query is a SELECT statement in text form, while in dynamic SQL, the query is associated with a statement name assigned in a PREPARE statement. Any referenced host variables are represented by parameter markers.

The main difference between a static and a dynamic cursor is that a static cursor is prepared at precompile time, and a dynamic cursor is prepared at run time. Additionally, host variables referenced in the query are represented by parameter markers, which are replaced by run-time host variables when the cursor is opened.

**Procedure:**

Use the examples shown in the following table when coding cursors for a dynamic SQL program:

*Table 19. Declare Statement Associated with a Dynamic SELECT*

| Language | Example Source Code |
|---|---|
| **C/C++** | `strcpy( prep_string, "SELECT tabname FROM syscat.tables"`<br>`                      "WHERE tabschema = ?" );`<br>`EXEC SQL PREPARE s1 FROM :prep_string;`<br>`EXEC SQL DECLARE c1 CURSOR FOR s1;`<br>`EXEC SQL OPEN c1 USING :`*`host_var`*`;` |

*Table 19. Declare Statement Associated with a Dynamic SELECT (continued)*

| Language | Example Source Code |
|---|---|
| **COBOL** | `MOVE "SELECT TABNAME FROM SYSCAT.TABLES WHERE TABSCHEMA = ?"`<br>`     TO PREP-STRING.`<br>`EXEC SQL PREPARE S1 FROM :PREP-STRING END-EXEC.`<br>`EXEC SQL DECLARE C1 CURSOR FOR S1 END-EXEC.`<br>`EXEC SQL OPEN C1 USING :`*`host-var`*` END-EXEC.` |
| **FORTRAN** | `prep_string = 'SELECT tabname FROM syscat.tables WHERE tabschema = ?'`<br>`EXEC SQL PREPARE s1 FROM :prep_string`<br>`EXEC SQL DECLARE c1 CURSOR FOR s1`<br>`EXEC SQL OPEN c1 USING :`*`host_var`* |
| **REXX** | `prep_string = "SELECT TABNAME FROM SYSCAT.TABLES WHERE TABSCHEMA = ?"`<br>`CALL SQLEXEC 'PREPARE S1 FROM :prep_string';`<br>`CALL SQLEXEC 'DECLARE C1 CURSOR FOR S1';`<br>`CALL SQLEXEC 'OPEN C1 USING :schema_name';` |

**Related concepts:**
- "Example of a cursor in a dynamically executed SQL application" on page 170

**Related tasks:**
- "Selecting multiple rows using a cursor in embedded SQL applications" on page 164

**Related reference:**
- "DECLARE CURSOR statement" in *SQL Reference, Volume 2*
- "FETCH statement" in *SQL Reference, Volume 2*
- "OPEN statement" in *SQL Reference, Volume 2*
- "PREPARE statement" in *SQL Reference, Volume 2*

**Example of a cursor in a statically executed SQL application:** The samples **tut_read.sqc** in C, **tut_read.sqC/sqx** in C++, and **cursor.sqb** in COBOL show how to declare a cursor, open the cursor, fetch rows from the table, and close the cursor.

Because REXX does not support static SQL, a sample is not provided.

- C and C++

  The sample tut_read shows a basic select from a table using a cursor. For example:

  ```
   /* declare cursor */
    EXEC SQL DECLARE c1 CURSOR FOR
      SELECT deptnumb, deptname FROM org WHERE deptnumb < 40;

  /* open cursor */
    EXEC SQL OPEN c1;

  /* fetch cursor */
    EXEC SQL FETCH c1 INTO :deptnumb, :deptname;

  while (sqlca.sqlcode != 100)
    {
      printf("    %8d %-14s\n", deptnumb, deptname);
      EXEC SQL FETCH c1 INTO :deptnumb, :deptname;
    }

    /* close cursor */
    EXEC SQL CLOSE c1;
  ```
- COBOL

The sample **cursor** shows an example on how to retrieve table data using a cursor with Static SQL statement. For example:

```
* Declare a cursor
     EXEC SQL DECLARE c1 CURSOR FOR
              SELECT name, dept FROM staff
              WHERE job='Mgr' END-EXEC.

* Open the cursor
     EXEC SQL OPEN c1 END-EXEC.

* Fetch rows from the 'staff' table
     perform Fetch-Loop thru End-Fetch-Loop
        until SQLCODE not equal 0.

* Close the cursor
     EXEC SQL CLOSE c1 END-EXEC.
     move "CLOSE CURSOR" to errloc.
```

**Related concepts:**

- "Cursor types and unit of work considerations in embedded SQL applications" on page 165
- "Error message retrieval in embedded SQL applications" on page 174

**Related tasks:**

- "Declaring and using cursors in statically executed SQL applications" on page 167
- "Selecting multiple rows using a cursor in embedded SQL applications" on page 164

**Related samples:**

- "cursor.sqb -- How to update table data with cursor statically (IBM COBOL)"

**Example of a cursor in a dynamically executed SQL application:** A dynamic SQL statement can be prepared for execution with the PREPARE statement and executed with the EXECUTE statement or the DECLARE CURSOR statement.

**PREPARE with EXECUTE**

The following example shows how a dynamic SQL statement can be prepared for execution with the PREPARE statement and executed with the EXECUTE statement:

- C and C++ (**dbuse.sqc/dbuse.sqC**):

  The following example is from the sample **dbuse**:

```
EXEC SQL BEGIN DECLARE SECTION;
  char hostVarStmt[50];
EXEC SQL END DECLARE SECTION;

strcpy(hostVarStmt, "DELETE FROM org WHERE deptnumb = 15");
EXEC SQL PREPARE Stmt FROM :hostVarStmt;
EXEC SQL EXECUTE Stmt;
```

**PREPARE with DECLARE CURSOR**

The following examples show how a dynamic SQL statement can be prepared for execution with the PREPARE statement, and executed with the DECLARE CURSOR statement:

- C

```
EXEC SQL BEGIN DECLARE SECTION;
  char  st[80];
  char  parm_var[19};
EXEC SQL END DECLARE SECTION;

strcpy( st, "SELECT tabname FROM syscat.tables" );
strcat( st, " WHERE tabname <> ? ORDER BY 1" );
EXEC SQL PREPARE s1 FROM :st;
EXEC SQL DECLARE c1 CURSOR FOR s1;
strcpy( parm_var, "STAFF" );
EXEC SQL OPEN c1 USING :parm_var;
```

- COBOL **(dynamic.sqb**)

  The following example is from the **dynamic.sqb** sample:

```
EXEC SQL BEGIN DECLARE SECTION END-EXEC.
     01 st             pic x(80).
     01 parm-var       pic x(18).
EXEC SQL END DECLARE SECTION END-EXEC.

move "SELECT TABNAME FROM SYSCAT.TABLES ORDER BY 1 WHERE TABNAME <> ?" to st.
EXEC SQL PREPARE s1 FROM :st END-EXEC.

EXEC SQL DECLARE c1 CURSOR FOR s1 END-EXEC.

move "STAFF" to parm-var.
EXEC SQL OPEN c1 USING :parm-var END-EXEC.
```

**EXECUTE IMMEDIATE**

You can also prepare and execute a dynamic SQL statement with the EXECUTE IMMEDIATE statement (except for SELECT statements that return more than one row).

- C and C++ (**dbuse.sqc/dbuse.sqC**)

  The following example is from the function `DynamicStmtEXECUTE_IMMEDIATE()` in the sample **dbuse**:

```
EXEC SQL BEGIN DECLARE SECTION;
  char stmt1[50];
EXEC SQL END DECLARE SECTION;

strcpy(stmt1, "CREATE TABLE table1(col1 INTEGER)");
EXEC SQL EXECUTE IMMEDIATE :stmt1;
```

**Related concepts:**

- "Error message retrieval in embedded SQL applications" on page 174

**Related samples:**

- "dbuse.sqc -- How to use a database (C)"
- "dbuse.out -- HOW TO USE A DATABASE (C)"
- "dbuse.out -- HOW TO USE A DATABASE (C++)"
- "dbuse.sqC -- How to use a database (C++)"

## Updating and deleting retrieved data in statically executed SQL application

It is possible to update and delete the row referenced by a cursor. For a row to be updatable, the query corresponding to the cursor must not be read-only.

**Procedure:**

To update with a cursor, use the WHERE CURRENT OF clause in an UPDATE statement. Use the FOR UPDATE clause to tell the system that you want to update some columns of the result table. You can specify a column in the FOR UPDATE without it being in the fullselect; therefore, you can update columns that are not explicitly retrieved by the cursor. If the FOR UPDATE clause is specified without column names, all columns of the table or view identified in the first FROM clause of the outer fullselect are considered to be updatable. Do not name more columns than you need in the FOR UPDATE clause. In some cases, naming extra columns in the FOR UPDATE clause can cause DB2 to be less efficient in accessing the data.

Deletion with a cursor is done using the WHERE CURRENT OF clause in a DELETE statement. In general, the FOR UPDATE clause is not required for deletion of the current row of a cursor. The only exception occurs when using dynamic SQL for either the SELECT statement or the DELETE statement in an application that has been precompiled with LANGLEVEL set to SAA1 and bound with BLOCKING ALL. In this case, a FOR UPDATE clause is necessary in the SELECT statement.

The DELETE statement causes the row being referenced by the cursor to be deleted. The deletion leaves the cursor positioned before the *next* row, and a FETCH statement must be issued before additional WHERE CURRENT OF operations can be performed against the cursor.

**Related tasks:**

- "Selecting multiple rows using a cursor in embedded SQL applications" on page 164

**Related reference:**

- "PRECOMPILE command" in *Command Reference*
- "SQL queries" in *SQL Reference, Volume 1*
- "UPDATE statement" in *SQL Reference, Volume 2*
- "DELETE statement" in *SQL Reference, Volume 2*

## Example of a fetch in a statically executed SQL program

The following sample selects from a table using a cursor, opens the cursor, and fetches rows from the table. For each row fetched, the program decides, based on simple criteria, whether the row should be deleted or updated.

The REXX language does not support static SQL, so a sample is not provided.

- C and C++ (**tut_mod.sqc/tut_mod.sqC**)

  The following example is from the sample **tut_mod**. This example selects from a table using a cursor, opens the cursor, fetches, updates, or delete rows from the table, then closes the cursor.

```
EXEC SQL DECLARE c1 CURSOR FOR SELECT * FROM staff WHERE id >= 310;
  EXEC SQL OPEN c1;
  EXEC SQL FETCH c1 INTO :id, :name, :dept, :job:jobInd, :years:yearsInd, :salary,
      :comm:commInd;
```

  The sample **tbmod** is a longer version of the **tut_mod** sample, and shows almost all possible cases of table data modification.

- COBOL (**openftch.sqb**)

  The following example is from the sample **openftch**. This example selects from a table using a cursor, opens the cursor, and fetches rows from the table.

```
   EXEC SQL DECLARE c1 CURSOR FOR
     SELECT name, dept FROM staff
     WHERE job='Mgr'
     FOR UPDATE OF job END-EXEC.

   EXEC SQL OPEN c1 END-EXEC


 * call the FETCH and UPDATE/DELETE loop.
    perform Fetch-Loop thru End-Fetch-Loop
    until SQLCODE not equal 0.

   EXEC SQL CLOSE c1 END-EXEC.
```

**Related concepts:**
- "Error message retrieval in embedded SQL applications" on page 174
- "Example of a cursor in a statically executed SQL application" on page 169

**Related tasks:**
- "Selecting multiple rows using a cursor in embedded SQL applications" on page 164
- "Declaring and using cursors in statically executed SQL applications" on page 167

**Related samples:**
- "openftch.sqb -- How to modify table data using cursor statically (IBM COBOL)"
- "tbmod.sqc -- How to modify table data (C)"
- "tbmod.sqC -- How to modify table data (C++)"

## Error message retrieval in embedded SQL applications

### Error information in the SQLCODE, SQLSTATE, and SQLWARN fields

Error information is returned in the SQLCODE and SQLSTATE fields of the SQLCA structure, which is updated after every executable SQL statement and most database manager API calls.

A source file containing executable SQL statements can provide at least one SQLCA structure with the name `sqlca`. The SQLCA structure is defined in the SQLCA include file. Source files without embedded SQL statements, but calling database manager APIs, can also provide one or more SQLCA structures, but their names are arbitrary.

If your application is compliant with the FIPS 127-2 standard, you can declare the SQLSTATE and SQLCODE as host variables for C, C++, COBOL, and FORTRAN applications, instead of using the SQLCA structure.

An SQLCODE value of 0 means successful execution (with possible SQLWARN warning conditions). A positive value means that the statement was successfully executed but with a warning, as with truncation of a host variable. A negative value means that an error condition occurred.

An additional field, SQLSTATE, contains a standardized error code consistent across other IBM database products and across SQL92–conformant database managers. Practically speaking, you should use SQLSTATE values when you are concerned about portability since SQLSTATE values are common across many database managers.

The SQLWARN field contains an array of warning indicators, even if SQLCODE is zero. The first element of the SQLWARN array, SQLWARN0, contains a blank if all other elements are blank. SQLWARN0 contains a `W` if at least one other element contains a warning character.

**Note:** If you want to develop applications that access various IBM RDBMS servers you should:

- Where possible, have your applications check the SQLSTATE rather than the SQLCODE.
- If your applications will use DB2 Connect, consider using the mapping facility provided by DB2 Connect to map SQLCODE conversions between unlike databases.

**Related concepts:**

- "Error message retrieval in embedded SQL applications" on page 174
- "Error-checking utilities" on page 202
- "SQLSTATE and SQLCODE variables in FORTRAN embedded SQL application" on page 122
- "SQLSTATE and SQLCODE Variables in COBOL embedded SQL application" on page 109
- "Include files and definitions required for embedded SQL applications" on page 42
- "SQLSTATE and SQLCODE variables in C and C++ embedded SQL application" on page 82

**Related reference:**

- "SQLCA data structure" in *Administrative API Reference*

## Error message retrieval in embedded SQL applications

Depending on the language in which your application is written, you use a different method to retrieve error information:

- C, C++, and COBOL applications can use the GET ERROR MESSAGE API to obtain the corresponding information related to the SQLCA passed in.

```
C Example: The SqlInfoPrint procedure from UTILAPI.C
/******************************************************************************
** 1.1 - SqlInfoPrint - prints diagnostic information to the screen.
**
******************************************************************************/
int SqlInfoPrint( char * appMsg,
    struct sqlca * pSqlca,
    int line,
    char * file )
{   int  rc = 0;
    char sqlInfo[1024];
    char sqlInfoToken[1024];
    char sqlstateMsg[1024];
    char errorMsg[1024];
    if (pSqlca->sqlcode != 0 && pSqlca->sqlcode != 100)
    {  strcpy(sqlInfo, "");
       if( pSqlca->sqlcode < 0)
       {   sprintf( sqlInfoToken, "\n---- error report ----\n");
```

```
            strcat( sqlInfo, sqlInfoToken);
        }
        else
        {   sprintf( sqlInfoToken, "\n---- warning report ----\n");
            strcat( sqlInfo, sqlInfoToken);
        } /* endif */

        sprintf( sqlInfoToken, " app. message = %s\n", appMsg);
        strcat( sqlInfo, sqlInfoToken);
        sprintf( sqlInfoToken, " line = %d\n", line);
        strcat( sqlInfo, sqlInfoToken);
        sprintf( sqlInfoToken, " file = %s\n", file);
        strcat( sqlInfo, sqlInfoToken);
        sprintf( sqlInfoToken, " SQLCODE = %ld\n", pSqlca->sqlcode);
        strcat( sqlInfo, sqlInfoToken);

        /* get error message */
        rc = sqlaintp( errorMsg, 1024, 80, pSqlca);
        /* return code is the length of the errorMsg string */
        if( rc > 0)
        { sprintf( sqlInfoToken, "%s\n", errorMsg);
          strcat( sqlInfo, sqlInfoToken);
        }

        /* get SQLSTATE message */
        rc = sqlogstt( sqlstateMsg, 1024, 80, pSqlca->sqlstate);
        if (rc == 0)
        { sprintf( sqlInfoToken, "%s\n", sqlstateMsg);
          strcat( sqlInfo, sqlInfoToken);
        }

        if( pSqlca->sqlcode < 0)
        { sprintf( sqlInfoToken, "--- end error report ---\n");
          strcat( sqlInfo, sqlInfoToken);
          printf("%s", sqlInfo);
          return 1;
        }
        else
        { sprintf( sqlInfoToken, "--- end warning report ---\n");
          strcat( sqlInfo, sqlInfoToken);

          printf("%s", sqlInfo);
          return 0;
        } /* endif */
    } /* endif */
    return 0;
}
```

```
COBOL Example: From CHECKERR.CBL
    ********************************
    * GET ERROR MESSAGE API called *
    ********************************
      call "sqlgintp" using
           by value buffer-size
           by value line-width
           by reference sqlca
           by reference error-buffer
        returning error-rc.
    ************************
    * GET SQLSTATE MESSAGE *
    ************************
      call "sqlggstt" using
           by value buffer-size
           by value line-width
           by reference sqlstate
           by reference state-buffer
        returning state-rc.
    if error-rc is greater than 0
      display error-buffer.

    if state-rc is greater than 0
      display state-buffer.

    if state-rc is less than 0
      display "return code from GET SQLSTATE =" state-rc.

    if SQLCODE is less than 0
      display "--- end error report ---"
      go to End-Prog.

    display "--- end error report ---"
    display "CONTINUING PROGRAM WITH WARNINGS!".
```

- REXX applications use the CHECKERR procedure.

```
/******   CHECKERR - Check SQLCODE *****/
CHECKERR:
  arg errloc

  if  ( SQLCA.SQLCODE = 0 ) then
    return 0
  else do
    say '--- error report ---'
    say 'ERROR occurred :' errloc
    say 'SQLCODE :' SQLCA.SQLCODE

   /*********************\
   * GET ERROR MESSAGE  *
   \*********************/
    call SQLDBS 'GET MESSAGE INTO :errmsg LINEWIDTH 80'
    say errmsg
    say '--- end error report ---'

    if (SQLCA.SQLCODE < 0 ) then
      exit
    else do
      say 'WARNING - CONTINUING PROGRAM WITH ERRORS'
      return 0
    end
  end
return 0
```

**Related concepts:**

- "SQLSTATE and SQLCODE variables in C and C++ embedded SQL application" on page 82
- "SQLSTATE and SQLCODE Variables in COBOL embedded SQL application" on page 109
- "SQLSTATE and SQLCODE variables in FORTRAN embedded SQL application" on page 122

**Related reference:**

- "sqlaintp API - Get error message" in *Administrative API Reference*

## Exit list routine considerations

Do not use SQL or DB2 API calls in exit list routines. Note that you cannot disconnect from a database in an exit routine.

**Related concepts:**

- "Disconnecting from embedded SQL applications" on page 178

**Related reference:**

- “DB2 APIs” in *Administrative API Reference*

## Exception, signal, and interrupt handler considerations

An exception, signal, or interrupt handler is a routine that gets control when an exception, signal, or interrupt occurs. The type of handler applicable is determined by your operating environment, as shown in the following:

**Windows operating systems**
: Pressing `Ctrl-C` or `Ctrl-Break` generates an interrupt.

**UNIX operating systems**
: Usually, pressing `Ctrl-C` generates the SIGINT interrupt signal. Note that keyboards can easily be redefined so SIGINT may be generated by a different key sequence on your machine.

Do not put SQL statements (other than COMMIT or ROLLBACK) in exception, signal, and interrupt handlers. With these kinds of error conditions, you normally want to do a ROLLBACK to avoid the risk of inconsistent data.

Note that you should exercise caution when coding a COMMIT and ROLLBACK in exception/signal/interrupt handlers. If you call either of these statements by themselves, the COMMIT or ROLLBACK is not executed until the current SQL statement is complete, if one is running. This is not the behavior desired from a `Ctrl-C` handler.

The solution is to call the INTERRUPT API (`sqleintr/sqlgintr`) before issuing a ROLLBACK. This API interrupts the current SQL query (if the application is executing one) and lets the ROLLBACK begin immediately. If you are going to perform a COMMIT rather than a ROLLBACK, you do not want to interrupt the current command.

When using APPC to access a remote database server (DB2 for AIX or host database system using DB2 Connect), the application may receive a SIGUSR1 signal. This signal is generated by SNA Services/6000 when an unrecoverable error occurs and the SNA connection is stopped. You may want to install a signal handler in your application to handle SIGUSR1.

Refer to your platform documentation for specific details on the various handler considerations.

**Related concepts:**

- “Error Handling Using the WHENEVER Statement” on page 51
- “Error message retrieval in embedded SQL applications” on page 174
- “Error-checking utilities” on page 202

**Related tasks:**

- “Declaring the SQLCA for Error Handling” on page 50

**Related reference:**

- “COMMIT statement” in *SQL Reference, Volume 2*
- “ROLLBACK statement” in *SQL Reference, Volume 2*

## Disconnecting from embedded SQL applications

The disconnect statement is the final step in working with a database. This topic will provide examples of the disconnect statement in the supported host languages.

**Disconnecting from DB2 databases in C and C++ Embedded SQL applications:**

When working with C and C++ applications, a database connection is closed by issuing the following statement:

```
EXEC SQL CONNECT RESET;
```

**Disconnecting from DB2 databases in COBOL Embedded SQL applications:**

When working with COBOL applications, a database connection is closed by issuing the following statement:

```
EXEC SQL CONNECT RESET END-EXEC.
```

**Disconnecting from DB2 databases in REXX Embedded SQL applications:**

When working with REXX applications, a database connection is closed by issuing the following statement:

```
CALL SQLEXEC 'CONNECT RESET'
```

When working with FORTRAN applications, a database connection is closed by issuing the following statement:

```
EXEC SQL CONNECT RESET
```

**Related concepts:**

- "Embedding SQL statements in a host language" on page 4
- "Executing SQL statements in embedded SQL applications" on page 136

**Related reference:**

- "DISCONNECT statement" in *SQL Reference, Volume 2*

# Chapter 4. Building embedded SQL applications

## Building embedded SQL applications

Once you have created the source code for your embedded SQL application, you must follow additional steps to build it. You should consider building 64-bit executables when developing new embedded SQL database applications. Along with compiling and linking your program, you must *precompile* and *bind* it.

The precompilation process converts embedded SQL statements into DB2 run-time API calls that a host language compiler can process. By default, a package is created at precompile time. Optionally, a bind file can be created at precompile time. The bind file contains information about the SQL statements in the application program. The bind file can be used later with the `BIND` command to create a package for the application.

Binding is the process of creating a *package* from a bind file and storing it in a database. The bind file must be bound to each database that needs to be accessed by the application. If your application accesses more than one database, you must create a package for each database.

To run applications written in compiled host languages, you must create the packages needed by the database manager at execution time. The following figure shows the order of these steps, along with the various modules of a typical compiled DB2 application.:

1. Create source files that contain programs with embedded SQL statements.
2. Connect to a database, then precompile each source file to convert embedded SQL source statements into a form the database manager can use.

   Since the SQL statements placed in an application are not specific to the host language, the database manager provides a way to convert the SQL syntax for processing by the host language. For C, C++, COBOL, or FORTRAN languages, this conversion is handled by the DB2 precompiler that is invoked using the `PRECOMPILE` (or `PREP`) command. The precompiler converts embedded SQL statements directly into DB2 run-time services API calls. When the precompiler processes a source file, it specifically looks for SQL statements and avoids the non-SQL host language.
3. Compile the modified source files (and other files without SQL statements) using the host language compiler.
4. Link the object files with the DB2 and host language libraries to produce an executable program.

   Compiling and linking (steps 3 and 4) create the required object modules
5. Bind the bind file to create the package if this was not already done at precompile time, or if a different database is going to be accessed. Binding creates the package to be used by the database manager when the program is run.

6. Run the application. The application accesses the database using the access plans.

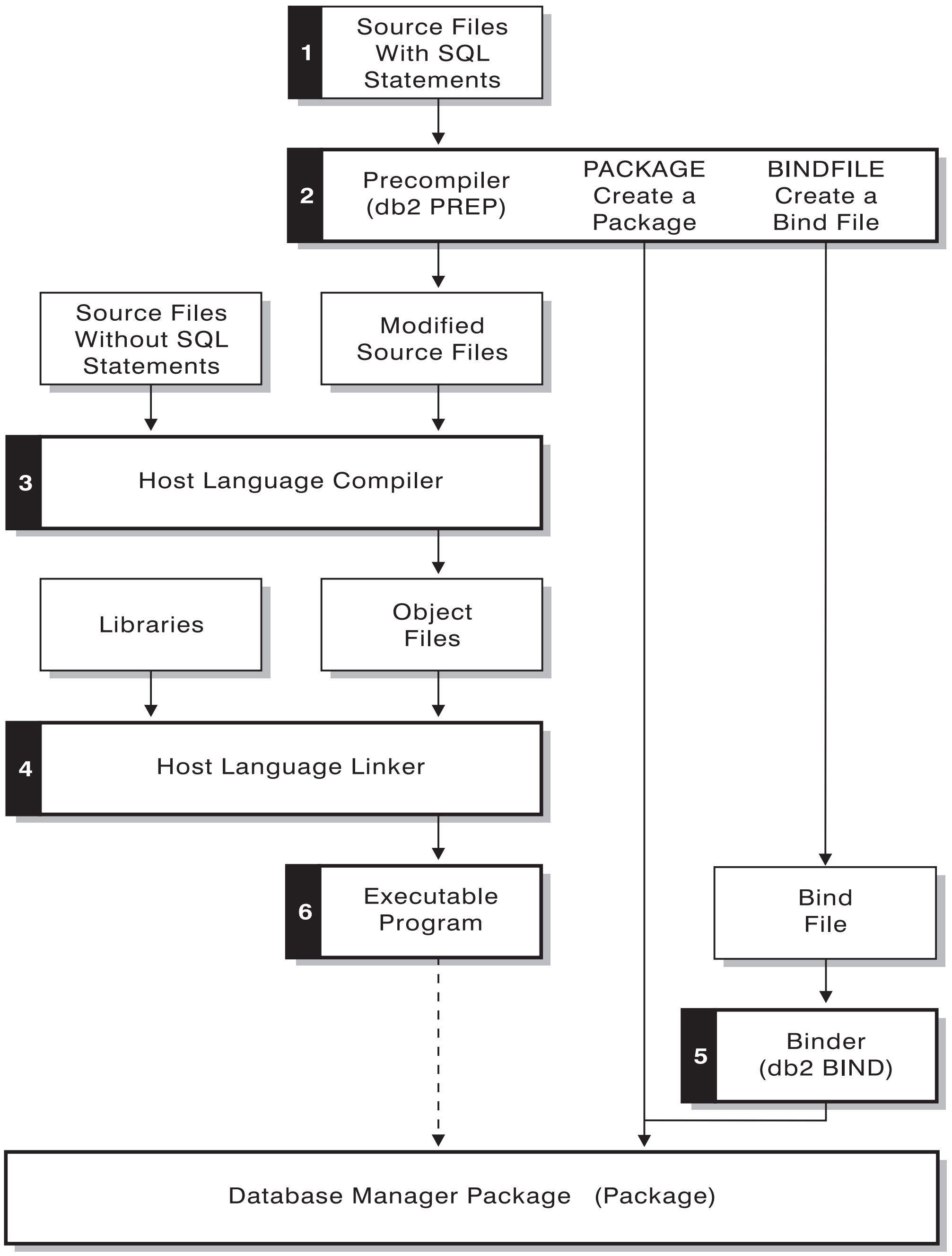


*Figure 4. Preparing Programs Written in Compiled Host Languages*

**Related concepts:**

- "Host Variables in embedded SQL applications" on page 66
- "Precompilation of embedded SQL applications with the PRECOMPILE command" on page 182
- "Binding embedded SQL packages to a database" on page 190
- "Embedded SQL application packages and access plans" on page 184

**Related tasks:**
- "Setting up the embedded SQL development environment" on page 11
- "Compiling and linking source files containing embedded SQL" on page 189

**Related reference:**
- "BIND command" in *Command Reference*
- "PRECOMPILE command" in *Command Reference*

## Precompilation of embedded SQL applications with the PRECOMPILE (or PREP) command

### Precompilation of embedded SQL applications with the PRECOMPILE command

Once you have created the embedded SQL application's source files, you must precompile each host language file containing SQL statements with the `PREP` command, using the options specific to the host language. The precompiler converts SQL statements contained in the source file to comments, and generates the DB2 run-time API calls for those statements.

You must always precompile a source file against a specific database, even if eventually you do not use the database with the application. In practice, you can use a test database for development, and after you fully test the application, you can bind its bind file to one or more production databases. This practice is known as *deferred binding*.

If your application uses a code page that is not the same as your database code page, you need to consider which code page to use when precompiling.

If your application uses user-defined functions (UDFs) or user-defined distinct types (UDTs), you may need to use the FUNCPATH option when you precompile your application. This option specifies the function path that is used to resolve UDFs and UDTs for applications containing static SQL. If FUNCPATH is not specified, the default function path is *SYSIBM*, *SYSFUN*, *USER*, where *USER* refers to the current user ID.

Before precompiling an application you must connect to a server, either implicitly or explicitly. Although you precompile application programs at the client workstation and the precompiler generates modified source and messages on the client, the precompiler uses the server connection to perform some of the validation.

The precompiler also creates the information the database manager needs to process the SQL statements against a database. This information is stored in a package, in a bind file, or in both, depending on the precompiler options selected.

A typical example of using the precompiler follows. To precompile a C embedded SQL source file called *filename.sqc*, you can issue the following command to create a C source file with the default name `filename.c` and a bind file with the default name `filename.bnd`:

```
DB2 PREP filename.sqc BINDFILE
```

The precompiler generates up to four types of output:

**Modified Source**

This file is the new version of the original source file after the precompiler converts the SQL statements into DB2 run-time API calls. It is given the appropriate host language extension.

**Package**
If you use the PACKAGE option (the default), or do not specify any of the BINDFILE, SYNTAX, or SQLFLAG options, the package is stored in the connected database. The package contains all the information required to execute the static SQL statements of a particular source file against this database only. Unless you specify a different name with the PACKAGE USING option, the precompiler forms the package name from the first 8 characters of the source file name.

If you use the PACKAGE option without SQLERROR CONTINUE, the database used during the precompile process must contain all of the database objects referenced by the static SQL statements in the source file. For example, you cannot precompile a SELECT statement unless the table it references exists in the database.

With the VERSION option, the bindfile (if the BINDFILE option is used) and the package (either if bound at PREP time or if bound separately) will be designated with a particular version identifier. Many versions of packages with the same name and creator can exist at once.

**Bind File**
If you use the BINDFILE option, the precompiler creates a bind file (with extension `.bnd`) that contains the data required to create a package. This file can be used later with the `BIND` command to bind the application to one or more databases. If you specify BINDFILE and do not specify the PACKAGE option, binding is deferred until you invoke the `BIND` command. Note that for the command line processor (CLP), the default for `PREP` does not specify the BINDFILE option. Thus, if you are using the CLP and want the binding to be deferred, you need to specify the BINDFILE option.

Specifying SQLERROR CONTINUE creates a package, even if errors occur when binding SQL statements. Those statements that fail to bind for authorization or existence reasons can be incrementally bound at execution time if VALIDATE RUN is also specified. Any attempt to execute them at run time generates an error.

**Message File**
If you use the MESSAGES option, the precompiler redirects messages to the indicated file. These messages include warning and error messages that describe problems encountered during precompilation. If the source file does not precompile successfully, use the warning and error messages to determine the problem, correct the source file, and then attempt to precompile the source file again. If you do not use the MESSAGES option, precompilation messages are written to the standard output.

**Related concepts:**

- “Advantages of deferred binding” on page 196
- “Character conversion between different code pages” in *Developing SQL and External Routines*

- "Character substitutions during code page conversions" in *Developing SQL and External Routines*
- "Code page conversion expansion factor" in *Developing SQL and External Routines*
- "Supported code page conversions" in *Developing SQL and External Routines*
- "When code page conversion occurs" in *Developing SQL and External Routines*
- "Host Variables in embedded SQL applications" on page 66
- "Comments in embedded SQL applications" on page 136
- "Binding embedded SQL packages to a database" on page 190
- "Connecting to DB2 databases in embedded SQL applications" on page 52

**Related tasks:**
- "Declaring host variables in embedded SQL applications" on page 68
- "Referencing host variables in embedded SQL applications" on page 76

**Related reference:**
- "BIND command" in *Command Reference*
- "PRECOMPILE command" in *Command Reference*

## Precompilation of embedded SQL applications that access more than one database server

To precompile an application program that accesses more than one server, you can do one of the following:
- Split the SQL statements for each database into separate source files. Do not mix SQL statements for different databases in the same file. Each source file can be precompiled against the appropriate database. This is the recommended method.
- Code your application using dynamic SQL statements only, and bind against each database your program will access.
- If all the databases look the same, that is, they have the same definition, you can group the SQL statements together into one source file.

The same procedures apply if your application will access a host application server through DB2 Connect. Precompile it against the server to which it will be connecting, using the `PREP` options available for that server.

**Related concepts:**
- "Static and dynamic SQL statement execution in embedded SQL applications" on page 17
- "Connecting to DB2 databases in embedded SQL applications" on page 52

## Embedded SQL application packages and access plans

The precompiler produces a package in the database and, optionally, a bind file, if you specify that you want one created.

The package contains access plans selected by the DB2 optimizer for the static SQL statements in your application. The access plans contain the information required by the database manager to execute the static SQL statements in the most efficient manner as determined by the optimizer. For dynamic SQL statements, the optimizer creates access plans when you run your application.

Packages stored in the database include information needed to execute specific SQL statements in a single source file. A database application uses one package for every precompiled source file used to build the application. Each package is a separate entity, and has no relationship to any other packages used by the same or other applications. Packages are created by running the precompiler against a source file with binding enabled, or by running the binder at a later time with one or more bind files.

The bind file contains the SQL statements and other data required to create a package. You can use the bind file to re-bind your application later without having to precompile it first. The re-binding creates packages that are optimized for current database conditions. You need to re-bind your application if it will access a different database from the one against which it was precompiled.

**Related concepts:**
- "Performance of embedded SQL applications" on page 22
- "Binding embedded SQL packages to a database" on page 190
- "Rebinding existing packages with the REBIND command" on page 194

## Package schema qualification using CURRENT PACKAGE PATH special register

Package schemas provide a method for logically grouping packages. Different approaches exist for grouping packages into schemas. Some implementations use one schema per environment (for example, a production and a test schema). Other implementations use one schema per business area (for example, `stocktrd` and `onlinebnk` schemas), or one schema per application (for example, `stocktrdAddUser` and `onlinebnkAddUser`). You can also group packages for general administration purposes, or to provide variations in the packages (for example, maintaining backup variations of applications, or testing new variations of applications).

When multiple schemas are used for packages, the database manager must determine in which schema to look for a package. To accomplish this task, the database manager uses the value of the CURRENT PACKAGESET special register. You can set this special register to a single schema name to indicate that any package to be invoked belongs to that schema. If an application uses packages in different schemas, a SET CURRENT PACKAGESET statement might have to be issued before each package is invoked if the schema for the package is different from that of the previous package.

**Note:** Only DB2 Version 9.1 for z/OS® (DB2 for z/OS) has a CURRENT PACKAGESET special register, which allows you to explicitly set the value (a single schema name) with the corresponding SET CURRENT PACKAGESET statement. Although DB2 Database for Linux, UNIX, and Windows has a SET CURRENT PACKAGESET statement, it does not have a CURRENT PACKAGESET special register. This means that CURRENT PACKAGESET cannot be referenced in other contexts (such as in a SELECT statement) with DB2 Database for Linux, UNIX, and Windows. DB2 UDB for iSeries does not provide support for CURRENT PACKAGESET.

The DB2 database server has more flexibility when it can consider a list of schemas during package resolution. The list of schemas is similar to the SQL path that is provided by the CURRENT PATH special register. The schema list is used for user-defined functions, procedures, methods, and distinct types.

**Note:** The SQL path is a list of schema names that DB2 should consider when trying to determine the schema for an unqualified function, procedure, method, or distinct type name.

If you need to associate multiple variations of a package (that is, multiple sets of BIND options for a package) with a single compiled program, consider isolating the path of schemas that are used for SQL objects from the path of schemas that are used for packages.

The CURRENT PACKAGE PATH special register allows you to specify a list of package schemas. Other DB2 family products provide similar capability with special registers such as CURRENT PATH and CURRENT PACKAGESET, which are pushed and popped for nested procedures and user-defined functions without corrupting the runtime environment of the invoking application. The CURRENT PACKAGE PATH special register provides this capability for package schema resolution.

Many installations use more than one schema for packages. If you do not specify a list of package schemas, you must issue the SET CURRENT PACKAGESET statement (which can contain at most one schema name) each time you require a package from a different schema. If, however, you issue a SET CURRENT PACKAGE PATH statement at the beginning of the application to specify a list of schema names, you do not need to issue a SET CURRENT PACKAGESET statement each time a package in a different schema is needed.

For example, assume that the following packages exist, and, using the following list, that you want to invoke the first one that exists on the server: SCHEMA1.PKG1, SCHEMA2.PKG2, SCHEMA3.PKG3, SCHEMA.PKG, and SCHEMA5.PKG5. Assuming the current support for a SET CURRENT PACKAGESET statement in DB2 Database for Linux, UNIX, and Windows (that is, accepting a single schema name), a SET CURRENT PACKAGESET statement would have to be issued before trying to invoke each package to specify the specific schema. For this example, five SET CURRENT PACKAGESET statements would need to be issued. However, using the CURRENT PACKAGE PATH special register, a single SET statement is sufficient. For example:

```
SET CURRENT PACKAGE PATH = SCHEMA1, SCHEMA2, SCHEMA3, SCHEMA, SCHEMA5;
```

**Note:** In DB2 Database for Linux, UNIX, and Windows, you can set the CURRENT PACKAGE PATH special register in the db2cli.ini file, by using the SQLSetConnectAttr API, in the SQLE-CLIENT-INFO structure, and by including the SET CURRENT PACKAGE PATH statement in embedded SQL programs. Only DB2 Universal Database for OS/390® and z/OS, Version 8 or later, supports the SET CURRENT PACKAGE PATH statement. If you issue this statement against a DB2 Database for Linux, UNIX, and Windows server or against DB2 Universal Database for AS/00, -30005 is returned.

You can use multiple schemas to maintain several variations of a package. These variations can be a very useful in helping to control changes made in production environments. You can also use different variations of a package to keep a backup version of a package, or a test version of a package (for example, to evaluate the impact of a new index). A previous version of a package is used in the same way as a backup application (load module or executable), specifically, to provide the ability to revert to a previous version.

For example, assume the PROD schema includes the current packages used by the production applications, and the BACKUP schema stores a backup copy of those packages. A new version of the application (and thus the packages) are promoted to production by binding them using the PROD schema. The backup copies of the packages are created by binding the current version of the applications using the backup schema (BACKUP). Then, at runtime, you can use the SET CURRENT PACKAGE PATH statement to specify the order in which the schemas should be checked for the packages. Assume that a backup copy of the application MYAPPL has been bound using the BACKUP schema, and the version of the application currently in production has been bound to the PROD schema creating a package PROD.MYAPPL. To specify that the variation of the package in the PROD schema should be used if it is available (otherwise the variation in the BACKUP schema is used), issue the following SET statement for the special register:

```
SET CURRENT PACKAGE PATH = PROD, BACKUP;
```

If you need to revert to the previous version of the package, the production version of the application can be dropped with the DROP PACKAGE statement, which causes the old version of the application (load module or executable) that was bound using the BACKUP schema to be invoked instead (application path techniques could be used here, specific to each operating system platform).

**Note:** This example assumes that the only difference between the versions of the package are in the BIND options that were used to create the packages (that is, there are no differences in the executable code).

The application does not use the SET CURRENT PACKAGESET statement to select the schema it wants. Instead, it allows DB2 to pick up the package by checking for it in the schemas listed in the CURRENT PACKAGE PATH special register.

**Note:** The DB2 Universal Database for OS/390 and z/OS precompile process stores a consistency token in the DBRM (which can be set using the LEVEL option), and during package resolution a check is made to ensure that the consistency token in the program matches the package. Similarly, the DB2 Database for Linux, UNIX, and Windows bind process stores a timestamp in the bind file. DB2 Database for Linux, UNIX, and Windows also supports a LEVEL option.

Another reason for creating several versions of a package in different schemas could be to cause different BIND options to be in affect. For example, you can use different qualifiers for unqualified name references in the package.

Applications are often written with unqualified table names. This supports multiple tables that have identical table names and structures, but different qualifiers to distinguish different instances. For example, a test system and a production system might have the same objects created in each, but they might have different qualifiers (for example, PROD and TEST). Another example is an application that distributes data into tables across different DB2 systems, with each table having a different qualifier (for example, EAST, WEST, NORTH, SOUTH; COMPANYA, COMPANYB; Y1999, Y2000, Y2001). With DB2 Universal Database for OS/390 and z/OS, you specify the table qualifier using the QUALIFIER option of the BIND command. When you use the QUALIFIER option, users do not have to maintain multiple programs, each of which specifies the fully qualified names that are required to access unqualified tables. Instead, the correct package can be accessed at runtime by issuing the SET CURRENT PACKAGESET statement from the application, and specifying a single schema name. However, if you use SET CURRENT PACKAGESET, multiple applications will still need to be kept and modified: each one with its own SET CURRENT PACKAGESET statement to access the required package. If you issue a SET CURRENT PACKAGE PATH statement instead, all of the schemas could be listed. At execution time, DB2 could choose the correct package.

**Note:** DB2 Database for Linux, UNIX, and Windows also supports a QUALIFIER bind option. However, the QUALIFIER bind option only affects static SQL or packages that use the DYNAMICRULES option of the BIND command.

**Related reference:**
- "SET CURRENT PACKAGESET statement" in *SQL Reference, Volume 2*
- "CURRENT PATH special register" in *SQL Reference, Volume 1*

## Precompiler generated timestamps

When an application is precompiled with binding enabled, the package and modified source file are generated with matching timestamps. These timestamps are individually known as a consistency token. If multiple versions of a package exist (by using the PRECOMPILE VERSION option), each version will have an associated timestamp. When the application is run, the package name, creator and timestamp are sent to the database manager, which checks for a package whose name, creator and timestamp match that sent by the application. If such a match does not exist, one of the two following SQL error codes is returned to the application:
- SQL0818N (timestamp conflict). This error is returned if a single package is found that matches the name and creator (but not the consistency token), and the package has a version of "" (an empty string)
- SQL0805N (package not found). This error is returned in all other situations.

Remember that when you bind an application to a database, the first eight characters of the application name are used as the package name *unless you override the default by using the PACKAGE USING option on the PREP command*. As well, the version ID will be "" (an empty string) unless it is specified by the VERSION option of the PREP command. This means that if you precompile and bind two programs using the same name without changing the version ID, the second package will replace the package of the first. When you run the first program, you will get a timestamp or a package not found error because the timestamp for the modified source file no longer matches that of the package in the database. The package not found error can also result from the use of the ACTION REPLACE REPLVER precompile or bind option as in the following example:
1. Precompile and bind the package SCHEMA1.PKG specifying VERSION VER1. Then generate the associated application A1.
2. Precompile and bind the package SCHEMA1.PKG, specifying VERSION VER2 ACTION REPLACE REPLVER VER1. Then generate the associated application A2.

   The second precompile and bind generates a package SCHEMA1.PKG that has a VERSION of VER2, and the specification of ACTION REPLACE REPLVER VER1 removes the SCHEMA1.PKG package that had a VERSION of VER1.

   An attempt to run the first application will result in a package mismatch and will fail.

A similar symptom will occur in the following example:
1. Precompile and bind the package SCHEMA1.PKG, specifying VERSION VER1. Then generate the associated application A1

2. Precompile and bind the package SCHEMA1.PKG, specifying VERSION VER2. Then generate the associated application A2

   At this point it is possible to run both applications A1 and A2, which will execute from packages SCHEMA1.PKG versions VER1 and VER2 respectively. If, for example, the first package is dropped (using the DROP PACKAGE SCHEMA1.PKG VERSION VER1 SQL statement), an attempt to run the application A1 will fail with a package not found error.

When a source file is precompiled but a package is not created, a bind file and modified source file are generated with matching timestamps. To run the application, the bind file is bound in a separate BIND step to create a package and the modified source file is compiled and linked. For an application that requires multiple source modules, the binding process must be done for each bind file.

In this deferred binding scenario, the application and package timestamps match because the bind file contains the same timestamp as the one that was stored in the modified source file during precompilation.

**Related concepts:**
- "Package recreation using the BIND command and an existing bind file" on page 194
- "Embedded SQL application packages and access plans" on page 184
- "Binding embedded SQL packages to a database" on page 190
- "Package versioning" on page 198

### Errors and warnings from precompilation of embedded SQL applications

Embedded SQL errors at precompile time are detected by the embedded SQL precompiler. The embedded SQL precompiler detects syntax errors such as missing semicolons and undeclared host variables in SQL statements. For each of these errors, an appropriate error message is generated.

**Related concepts:**
- "Programming embedded SQL applications" on page 37

## Compiling and linking source files containing embedded SQL

When precompiling embedded SQL source files, the `PRECOMPILE` command generates modified source files with a file extension applicable to the programming language.

Compile the modified source files (and any additional source files that do not contain SQL statements) using the appropriate host language compiler. The language compiler converts each modified source file into an *object module*.

Refer to the programming documentation for your operating platform for any exceptions to the default compiler options. Refer to your compiler's documentation for a complete description of available compiler options.

The host language linker creates an executable application. For example:
- On Windows operating systems, the application can be an executable file or a dynamic link library (DLL).

- On UNIX and Linux based operating systems, the application can be an executable load module or a shared library.

**Note:** Although applications can be DLLs on Windows operating systems, the DLLs are loaded directly by the application and not by the DB2 database manager. On Windows operating systems, the database manager loads embedded SQL stored procedures and user-defined functions as DLLs.

To create the executable file, link the following:

- User object modules, generated by the language compiler from the modified source files and other files not containing SQL statements.
- Host language library APIs, supplied with the language compiler.
- The database manager library containing the database manager APIs for your operating environment. Refer to the appropriate programming documentation for your operating platform for the specific name of the database manager library you need for your database manager APIs.

**Related tasks:**

- "Building and running embedded SQL applications written in REXX" on page 239
- "Building applications in C or C++ using the sample build script (UNIX)" on page 203
- "Building IBM COBOL applications on AIX" on page 222
- "Building UNIX Micro Focus COBOL applications" on page 223

## Binding embedded SQL packages to a database with the BIND command

### Binding embedded SQL packages to a database

Binding is the process of creating a package from a bind file and storing it in a database.

**Application, Bind File, and Package Relationships:**

Database applications use packages for some of the same reasons that applications are compiled: improved performance and compactness. By precompiling an SQL statement, the statement is compiled into the package when the application is built, instead of at run time. Each statement is parsed, and a more efficiently interpreted operand string is stored in the package. At run time, the code generated by the precompiler calls run-time services database manager APIs with any variable information required for input or output data, and the information stored in the package is executed.

The advantages of precompilation apply only to static SQL statements. SQL statements that are executed dynamically (using PREPARE and EXECUTE or EXECUTE IMMEDIATE) are not precompiled; therefore, they must go through the entire set of processing steps at run time.

With the DB2 bind file description (`db2bfd`) utility, you can easily display the contents of a bind file to examine and verify the SQL statements within it, as well as display the precompile options used to create the bind file. This may be useful in problem determination related to your application's bind file.

**Related concepts:**

- "Performance of embedded SQL applications" on page 22
- "Performance improvements when using REOPT option of the BIND command" on page 197

**Related reference:**

- "db2bfd - Bind file description tool command" in *Command Reference*
- "EXECUTE statement" in *SQL Reference, Volume 2*
- "PREPARE statement" in *SQL Reference, Volume 2*

## Effect of DYNAMICRULES bind option on dynamic SQL

The PRECOMPILE command and BIND command option DYNAMICRULES determines what values apply at run-time for the following dynamic SQL attributes:

- The authorization ID that is used during authorization checking.
- The qualifier that is used for qualification of unqualified objects.
- Whether the package can be used to dynamically prepare the following statements: GRANT, REVOKE, ALTER, CREATE, DROP, COMMENT ON, RENAME, SET INTEGRITY and SET EVENT MONITOR STATE statements.

In addition to the DYNAMICRULES value, the run-time environment of a package controls how dynamic SQL statements behave at run-time. The two possible run-time environments are:

- The package runs as part of a stand-alone program
- The package runs within a routine context

The combination of the DYNAMICRULES value and the run-time environment determine the values for the dynamic SQL attributes. That set of attribute values is called the dynamic SQL statement behavior. The four behaviors are:

**Run behavior** DB2 uses the authorization ID of the user (the ID that initially connected to DB2) executing the package as the value to be used for authorization checking of dynamic SQL statements and for the initial value used for implicit qualification of unqualified object references within dynamic SQL statements.

**Bind behavior** At run-time, DB2 uses all the rules that apply to static SQL for authorization and qualification. That is, take the authorization ID of the package owner as the value to be used for authorization checking of dynamic SQL statements and the package default qualifier for implicit qualification of unqualified object references within dynamic SQL statements.

**Define behavior**
Define behavior applies only if the dynamic SQL statement is in a package that is run within a routine context, and the package was bound with DYNAMICRULES DEFINEBIND or DYNAMICRULES DEFINERUN. DB2 uses the authorization ID of the routine definer (not the routine's package binder) as the value to be used for authorization checking of dynamic SQL statements and for implicit qualification of unqualified object references within dynamic SQL statements within that routine.

**Invoke behavior**

Invoke behavior applies only if the dynamic SQL statement is in a package that is run within a routine context, and the package was bound with DYNAMICRULES INVOKEBIND or DYNAMICRULES INVOKERUN. DB2 uses the current statement authorization ID in effect when the routine is invoked as the value to be used for authorization checking of dynamic SQL and for implicit qualification of unqualified object references within dynamic SQL statements within that routine. This is summarized by the following table:

| Invoking Environment | ID Used |
|---|---|
| Any static SQL | Implicit or explicit value of the OWNER of the package the SQL invoking the routine came from. |
| Used in definition of view or trigger | Definer of the view or trigger. |
| Dynamic SQL from a run behavior package | ID used to make the initial connection to DB2. |
| Dynamic SQL from a define behavior package | Definer of the routine that uses the package that the SQL invoking the routine came from. |
| Dynamic SQL from an invoke behavior package | Current authorization ID invoking the routine. |

The following table shows the combination of the DYNAMICRULES value and the run-time environment that yields each dynamic SQL behavior.

*Table 20. How DYNAMICRULES and the Run-Time Environment Determine Dynamic SQL Statement Behavior*

| DYNAMICRULES Value | Behavior of Dynamic SQL Statements in a Standalone Program Environment | Behavior of Dynamic SQL Statements in a Routine Environment |
|---|---|---|
| BIND | Bind behavior | Bind behavior |
| RUN | Run behavior | Run behavior |
| DEFINEBIND | Bind behavior | Define behavior |
| DEFINERUN | Run behavior | Define behavior |
| INVOKEBIND | Bind behavior | Invoke behavior |
| INVOKERUN | Run behavior | Invoke behavior |

The following table shows the dynamic SQL attribute values for each type of dynamic SQL behavior.

*Table 21. Definitions of Dynamic SQL Statement Behaviors*

| Dynamic SQL Attribute | Setting for Dynamic SQL Attributes: Bind Behavior | Setting for Dynamic SQL Attributes: Run Behavior | Setting for Dynamic SQL Attributes: Define Behavior | Setting for Dynamic SQL Attributes: Invoke Behavior |
|---|---|---|---|---|
| Authorization ID | The implicit or explicit value of the OWNER BIND option | ID of User Executing Package | Routine definer (not the routine's package owner) | Current statement authorization ID when routine is invoked. |

Table 21. Definitions of Dynamic SQL Statement Behaviors (continued)

| Dynamic SQL Attribute | Setting for Dynamic SQL Attributes: Bind Behavior | Setting for Dynamic SQL Attributes: Run Behavior | Setting for Dynamic SQL Attributes: Define Behavior | Setting for Dynamic SQL Attributes: Invoke Behavior |
|---|---|---|---|---|
| Default qualifier for unqualified objects | The implicit or explicit value of the QUALIFIER BIND option | CURRENT SCHEMA Special Register | Routine definer (not the routine's package owner) | Current statement authorization ID when routine is invoked. |
| Can execute GRANT, REVOKE, ALTER, CREATE, DROP, COMMENT ON, RENAME, SET INTEGRITY and SET EVENT MONITOR STATE | No | Yes | No | No |

**Related concepts:**
- "Authorization considerations for dynamic SQL" in *Developing SQL and External Routines*
- "Authorizations and binding of routines that contain SQL" in *Developing SQL and External Routines*
- "Precompilation of embedded SQL applications with the PRECOMPILE command" on page 182

**Related tasks:**
- "Setting up the embedded SQL development environment" on page 11

## Using special registers to control the statement compilation environment

For dynamically prepared statements, the values of a number of special registers determine the statement compilation environment:
- The CURRENT QUERY OPTIMIZATION special register determines which optimization class is used.
- The CURRENT PATH special register determines the function path used for UDF and UDT resolution.
- The CURRENT EXPLAIN SNAPSHOT register determines whether explain snapshot information is captured.
- The CURRENT EXPLAIN MODE register determines whether explain table information is captured for any eligible dynamic SQL statement. The default values for these special registers are the same defaults used for the related bind options.

**Related tasks:**
- "Preparing a dynamically executed SQL statement using the minimum SQLDA structure" on page 140

**Related reference:**
- "CURRENT EXPLAIN MODE special register" in *SQL Reference, Volume 1*
- "CURRENT EXPLAIN SNAPSHOT special register" in *SQL Reference, Volume 1*

- “CURRENT PATH special register” in *SQL Reference, Volume 1*
- “CURRENT QUERY OPTIMIZATION special register” in *SQL Reference, Volume 1*

## Package recreation using the BIND command and an existing bind file

Binding is the process that creates the package the database manager needs to access the database when the application is executed. By default the PRECOMPILE command creates a package. Binding is done implicitly at precompile time unless the BINDFILE option is specified. The PACKAGE option allows you to specify a package name for the package created at precompile time.

A typical example of using the `BIND` command follows. To bind a bind file named *filename.bnd* to the database, you can issue the following command:

```
BIND filename.bnd
```

One package is created for each separately precompiled source code module. If an application has five source files, of which three require precompilation, three packages or bind files are created. By default, each package is given a name that is the same as the name of the source module from which the `.bnd` file originated, but truncated to 8 characters. To explicitly specify a different package name, you must use the PACKAGE USING option on the `PREP` command. The version of a package is given by the VERSION precompile option and defaults to the empty string. If the name and schema of this newly created package is the same as a package that currently exists in the target database, but the version identifier differs, a new package is created and the previous package still remains. However if a package exists that matches the name, schema and the version of the package being bound, then that package is dropped and replaced with the new package being bound (specifying ACTION ADD on the bind would prevent that and an error (SQL0719) would be returned instead).

**Related concepts:**

- “Precompilation of embedded SQL applications with the PRECOMPILE command” on page 182
- “Precompiler generated timestamps” on page 188

**Related reference:**

- “BIND command” in *Command Reference*
- “PRECOMPILE command” in *Command Reference*

## Rebinding existing packages with the REBIND command

*Rebinding* is the process of recreating a package for an application program that was previously bound. You must rebind packages if they have been marked invalid or inoperative or if the database statistics have changed since the last binding. In some situations, however, you may want to rebind packages that are valid. For example, you may want to take advantage of a newly created index, or make use of updated statistics after executing the RUNSTATS command.

Packages can be dependent on certain types of database objects such as tables, views, aliases, indexes, triggers, referential constraints and table check constraints. If a package is dependent on a database object (such as a table, view, trigger, and so on), and that object is dropped, the package is placed into an *invalid* state. If the object that is dropped is a UDF, the package is placed into an *inoperative* state.

Invalid packages are implicitly (or automatically) rebound by the database manager when they are executed. Inoperative packages must be explicitly rebound by executing either the `BIND` command or the `REBIND` command. Note that implicit rebinding can cause unexpected errors if the implicit rebind fails. That is, the implicit rebind error is returned on the statement being executed, which may not be the statement that is actually in error. If an attempt is made to execute an inoperative package, an error occurs. You may decide to explicitly rebind invalid packages rather than have the system automatically rebind them. This enables you to control when the rebinding occurs.

The choice of which command to use to explicitly rebind a package depends on the circumstances. You must use the `BIND` command to rebind a package for a program which has been modified to include more, fewer, or changed SQL statements. You must also use the `BIND` command if you need to change any bind options from the values with which the package was originally bound. In all other cases, use either the `BIND` or `REBIND` command. You should use `REBIND` whenever your situation does not specifically require the use of `BIND`, as the performance of `REBIND` is significantly better than that of `BIND`.

When multiple versions of the same package name coexist in the catalog, only one version can be rebound at a time.

**Related concepts:**
- "Statement dependencies when changing objects" in *Administration Guide: Implementation*
- "Binding embedded SQL packages to a database" on page 190

**Related reference:**
- "BIND command" in *Command Reference*
- "REBIND command" in *Command Reference*
- "RUNSTATS command" in *Command Reference*

## Bind considerations

If your application code page uses a different code page from your database code page, you may need to consider which code page to use when binding.

If your application issues calls to any of the database manager utility APIs, such as IMPORT or EXPORT, you must bind the supplied utility bind files to the database.

You can use bind options to control certain operations that occur during binding, as in the following examples:
- The QUERYOPT bind option takes advantage of a specific optimization class when binding.
- The EXPLSNAP bind option stores Explain Snapshot information for eligible SQL statements in the Explain tables.
- The FUNCPATH bind option properly resolves user-defined distinct types and user-defined functions in static SQL.

If the bind process starts but never returns, it may be that other applications connected to the database hold locks that you require. In this case, ensure that no applications are connected to the database. If they are, disconnect all applications on the server and the bind process will continue.

If your application will access a server using DB2 Connect, you can use the `BIND` options available for that server.

Bind files are not backward compatible with previous versions of DB2 UDB for Linux, UNIX, and Windows. In mixed-level environments, DB2 can only use the functions available to the lowest level of the database environment. For example, if a version 8 client connects to a version 7.2 server, the client will only be able to use version 7.2 functions. As bind files express the functionality of the database, they are subject to the mixed-level restriction.

If you need to rebind higher-level bind files on lower-level systems, you can:
- Use a lower-level DB2 Client to connect to the higher-level server and create bind files which can be shipped and bound to the lower-level DB2 UDB for Linux, UNIX, and Windows environment.
- Use a higher-level DB2 client in the lower-level production environment to bind the higher-level bind files that were created in the test environment. The higher-level client passes only the options that apply to the lower-level server.

**Related concepts:**
- "Character conversion between different code pages" in *Developing SQL and External Routines*
- "Character substitutions during code page conversions" in *Developing SQL and External Routines*
- "Code page conversion expansion factor" in *Developing SQL and External Routines*
- "Active code page for precompilation and binding" in *Developing SQL and External Routines*

**Related tasks:**
- "Binding utilities to the database" in *Administration Guide: Implementation*

**Related reference:**
- "BIND command" in *Command Reference*
- "SET CURRENT QUERY OPTIMIZATION statement" in *SQL Reference, Volume 2*

## Advantages of deferred binding

Precompiling with binding enabled allows an application to access only the database used during the precompile process. Precompiling with binding deferred, however, allows an application to access many databases, because you can bind the BIND file against each one. This method of application development is inherently more flexible in that applications are precompiled only once, but the application can be bound to a database at any time.

Using the BIND API during execution allows an application to bind itself, perhaps as part of an installation procedure or before an associated module is executed. For example, an application can perform several tasks, only one of which requires the use of SQL statements. You can design the application to bind itself to a database only when the application calls the task requiring SQL statements, and only if an associated package does not already exist.

Another advantage of the deferred binding method is that it lets you create packages without providing source code to end users. You can ship the associated bind files with the application.

**Related concepts:**
- "Connecting to DB2 databases in embedded SQL applications" on page 52

**Related reference:**
- "sqlabndx API - Bind application program to create a package" in *Administrative API Reference*

## Performance improvements when using REOPT option of the BIND command

The bind option REOPT can significantly improve the Embedded SQL application performance. The following are the descriptions for both Static and Dynamic SQL.

**Effects of REOPT on static SQL:**

The bind option REOPT can make static SQL statements containing host variables or special registers behave like incremental-bind statements. This means that these statements get compiled at the time of EXECUTE or OPEN instead of at bind time. During this compilation, the access plan is chosen, based on the real values of these variables.

With REOPT ONCE, the access plan is cached after the first OPEN or EXECUTE request and is used for subsequent execution of this statement. With REOPT ALWAYS, the access plan is regenerated for every OPEN and EXECUTE request, and the current set of host variable, parameter marker, and special register values is used to create this plan.

**Effects of REOPT on dynamic SQL:**

When you specify the option REOPT ALWAYS, DB2 postpones preparing any statement containing host variables, parameter markers, or special registers until it encounters an OPEN or EXECUTE statement; that is, when the values for these variables become known. At this time, the access plan is generated using these values. Subsequent OPEN or EXECUTE requests for the same statement will recompile the statement, reoptimize the query plan using the current set of values for the variables, and execute the newly generated query plan.

The option REOPT ONCE has a similar effect, with the exception that the plan is only optimized once using the values of the host variables, parameter markers and special registers. This plan is cached and will be used by subsequent requests.

**Related concepts:**
- "Performance of embedded SQL applications" on page 22
- "Determining when to execute SQL statements statically or dynamically in embedded SQL applications" on page 19
- "Using special registers to control the statement compilation environment" on page 193
- "Host Variables in embedded SQL applications" on page 66
- "Embedded SQL application packages and access plans" on page 184

## Package storage and maintenance

### Package storage and maintenance

Packages are created by precompiling/binding an application program. The package contains an optimized access plan which oversees the execution of all of the SQL statements found within the application. The three types of privileges that deal with packages are the `CONTROL`, `EXECUTE`, and `BIND` privilege and they are used to filter the level of access acceptable. Multiple versions of the same package can be created by specifying the `VERSION` option at compile time. This option helps prevent the mismatched timestamp error and allows for multiple versions of the application to run simultaneously.

**Related concepts:**

- "Package versioning" on page 198
- "Resolution of unqualified table names" on page 199

### Package versioning

If you need to create multiple versions of an application, you can use the VERSION option of the PRECOMPILE command. This option allows multiple versions of the same package name (that is, the package name and creator name) to coexist. For example, assume you have an application called `foo`, which is compiled from `foo.sqc`. You would precompile and bind the package `foo` to the database and deliver the application to the users. The users could then run the application. To make subsequent changes to the application, you would update `foo.sqc`, then repeat the process of recompiling, binding, and sending the application to the users. If the VERSION option was not specified for either the first or second precompilation of `foo.sqc`, the first package is replaced by the second package. Any user who attempts to run the old version of the application will receive the SQLCODE -818, indicating a mismatched timestamp error.

To avoid the mismatched timestamp error and in order to allow both versions of the application to run at the same time, use package versioning. As an example, when you build the first version of `foo`, precompile it using the VERSION option, as follows:

```
DB2 PREP FOO.SQC VERSION V1.1
```

This first version of the program may now be run. When you build the new version of `foo`, precompile it with the command:

```
DB2 PREP FOO.SQC VERSION V1.2
```

At this point this new version of the application will also run, even if there still are instances of the first application still executing. Because the package version for the first package is V1.1 and the package version for the second is V1.2, no naming conflict exists: both packages will exist in the database and both versions of the application can be used.

You can use the ACTION option of the PRECOMPILE or BIND commands in conjunction with the VERSION option of the PRECOMPILE command. You use the ACTION option to control the way in which different versions of packages can be added or replaced.

Package privileges do not have granularity at the version level. That is, a GRANT or a REVOKE of a package privilege applies to all versions of a package that share the name and creator. So, if package privileges on package `foo` were granted to a user or a group after version V1.1 was created, when version V1.2 is distributed the user or group has the same privileges on version V1.2. This behavior is usually required because typically the same users and groups have the same privileges on all versions of a package. If you do not want the same package privileges to apply to all versions of an application, you should not use the PRECOMPILE VERSION option to accomplish package versioning. Instead, you should use different package names (either by renaming the updated source file, or by using the PACKAGE USING option to explicitly rename the package).

**Related concepts:**
- "Precompiler generated timestamps" on page 188
- "Precompilation of embedded SQL applications with the PRECOMPILE command" on page 182

**Related tasks:**
- "Including SQLSTATE and SQLCODE host variables in embedded SQL applications" on page 75

**Related reference:**
- "BIND command" in *Command Reference*
- "PRECOMPILE command" in *Command Reference*

## Resolution of unqualified table names

You can handle unqualified table names in your application by using one of the following methods:

- Each user can bind their package with different COLLECTION parameters using different authorization identifiers by using the following commands:
  ```
  CONNECT TO db_name USER user_name
  BIND file_name COLLECTION schema_name
  ```
  In the above example, *db_name* is the name of the database, *user_name* is the name of the user, and *file_name* is the name of the application that will be bound. Note that *user_name* and *schema_name* are usually the same value. Then use the SET CURRENT PACKAGESET statement to specify which package to use, and therefore, which qualifiers will be used. If COLLECTION is not specified, then the default qualifier is the authorization identifier that is used when binding the package. If COLLECTION is specified, then the *schema_name* specified is the qualifier that will be used for unqualified objects.
- Create views for each user with the same name as the table so the unqualified table names resolve correctly.
- Create an alias for each user to point to the desired table.

**Related reference:**
- "BIND command" in *Command Reference*
- "SET CURRENT PACKAGESET statement" in *SQL Reference, Volume 2*
- "CREATE ALIAS statement" in *SQL Reference, Volume 2*
- "CREATE VIEW statement" in *SQL Reference, Volume 2*

## Building embedded SQL applications using the sample build script

### Building embedded SQL applications using the sample build script

The files used to demonstrate building sample programs are known as script files on UNIX and Linux, and batch files on Windows. We refer to them, generically, as build files. They contain the recommended compile and link commands for supported platform compilers.

Build files are provided by DB2 for host languages pertaining to supported platforms. The build files are available in the same directory to where the samples for that language are contained. The following table lists the different types of build files for building different types of programs. These build files, unless otherwise indicated, are for supported languages on all supported platforms. The build files have the `.bat` (batch) extension on Windows, which is not included in the table. There is no extension for UNIX platforms.

*Table 22. DB2 build files*

| Build file | Types of programs built |
|---|---|
| bldapp | Application programs |
| bldrtn | Routines (stored procedures and UDFs) |
| bldmc | C/C++ multi-connection applications |
| bldmt | C/C++ multi-threaded applications |
| bldcli | CLI client applications for SQL procedures in the sqlproc samples sub-directory. |

**Note:** By default the bldapp sample scripts for building executables from source code will build 64-bit executables.

The following table lists the build files by platform and programming language, and the directories where they are located. In the online documentation, the build file names are hot-linked to the source files in HTML. The user can also access the text files in the appropriate samples directories.

*Table 23. Build files by language and platform*

| Platform —><br>Language | AIX | HP-UX | Linux | Solaris | Windows |
|---|---|---|---|---|---|
| C<br>samples/c | `bldapp`<br>`bldrtn`<br>`bldmt`<br>`bldmc` | `bldapp`<br>`bldrtn`<br>`bldmt`<br>`bldmc` | `bldapp`<br>`bldrtn`<br>`bldmt`<br>`bldmc` | `bldapp`<br>`bldrtn`<br>`bldmt`<br>`bldmc` | `bldapp.bat`<br>`bldrtn.bat`<br>`bldmt.bat`<br>`bldmc.bat` |
| C++<br>samples/cpp | `bldapp`<br>`bldrtn`<br>`bldmt`<br>`bldmc` | `bldapp`<br>`bldrtn`<br>`bldmt`<br>`bldmc` | `bldapp`<br>`bldrtn`<br>`bldmt`<br>`bldmc` | `bldapp`<br>`bldrtn`<br>`bldmt`<br>`bldmc` | `bldapp.bat`<br>`bldrtn.bat`<br>`bldmt.bat`<br>`bldmc.bat` |
| IBM COBOL<br>samples/cobol | `bldapp`<br>`bldrtn` | `n/a` | `n/a` | `n/a` | `bldapp.bat`<br>`bldrtn.bat` |
| Micro Focus COBOL<br>samples/cobol_mf | `bldapp`<br>`bldrtn` | `bldapp`<br>`bldrtn` | `bldapp`<br>`bldrtn` | `bldapp`<br>`bldrtn` | `bldapp.bat`<br>`bldrtn.bat` |

The build files are used in the documentation for building applications and routines because they demonstrate very clearly the compile and link options that DB2 recommends for the supported compilers. There are generally many other compile and link options available, and users are free to experiment with them. See your compiler documentation for all the compile and link options provided. Besides building the sample programs, developers can also build their own programs with the build files. The sample programs can be used as templates that can be modified by users to assist in their application development.

Conveniently, the build files are designed to build a source file with any file name allowed by the compiler. This is unlike the makefiles, where the program names are hardcoded into the file. The makefiles access the build files for compiling and linking the programs they make. The build files use the `$1` variable on UNIX and Linux and the `%1` variable on Windows operating systems to substitute internally for the program name. Incremented numbers for these variable names substitute for other arguments that might be required.

The build files allow for quick and easy experimentation, as each one is suited to a specific kind of program-building, such as stand-alone applications, routines (stored procedures and UDFs) or more specialized program types such as multi-connection or multi-threaded programs. Each type of build file is provided wherever the specific kind of program it is designed for is supported by the compiler.

The object and executable files produced by a build file are automatically overwritten each time a program is built, even if the source file is not modified. This is not the case when using a makefile. It means a developer can rebuild an existing program without having to delete previous object and executable files, or modifying the source.

The build files contain a default setting for the sample database. If the user is accessing another database, they can simply supply another parameter to override the default. If they are using the other database consistently, they could hardcode this database name, replacing `sample`, within the build file itself.

For embedded SQL programs, except when using the IBM COBOL precompiler on Windows, the build files call another file, `embprep`, that contains the precompile and bind steps for embedded SQL programs. These steps might require the optional parameters for user ID and password, depending on where the embedded SQL program is being built.

Finally, the build files can be modified by the developer for his or her convenience. Besides changing the database name in the build file (explained above) the developer can easily hardcode other parameters within the file, change compile and link options, or change the default DB2 instance path. The simple, straightforward, and specific nature of the build files makes tailoring them to your needs an easy task.

**Related concepts:**
- "Sample files" in *Samples Topics*
- "The DB2 database application development environment" in *Getting Started with Database Application Development*
- "Building embedded SQL applications" on page 180
- "Error-checking utilities" on page 202
- "Host Variables in embedded SQL applications" on page 66

**Related tasks:**

- "Compiling and linking source files containing embedded SQL" on page 189

## Error-checking utilities

The DB2 Client provides several utility files. These files have functions for error-checking and printing out error information. Utility files are provided for each language in the samples directory. When used with an application program, the error-checking utility files provide helpful error information, and make debugging a DB2 program much easier. Most of the error-checking utilities use the DB2 APIs `GET SQLSTATE MESSAGE` (sqlogstt) and `GETERROR MESSAGE` (sqlaintp) to obtain pertinent SQLSTATE and SQLCA information related to problems encountered in program execution. The DB2 CLI utility file, `utilcli.c`, does not use these DB2 APIs; instead it uses equivalent DB2 CLI statements. With all the error-checking utilities, descriptive error messages are printed out to allow the developer to quickly understand the problem. Some DB2 programs, such as routines (stored procedures and user-defined functions), do not need to use the utilities.

Here are the error-checking utility files used by DB2-supported compilers for the different programming languages:

*Table 24. Error-checking utility files by language*

| Language | Non-embedded SQL source file | Non-embedded SQL header file | Embedded SQL source file | Embedded SQL header file |
|---|---|---|---|---|
| C<br>samples/c | `utilapi.c` | `utilapi.h` | `utilemb.sqc` | `utilemb.h` |
| C++<br>samples/cpp | `utilapi.C` | `utilapi.h` | `utilemb.sqC` | `utilemb.h` |
| IBM COBOL<br>samples/cobol | `checkerr.cbl` | `n/a` | `n/a` | `n/a` |
| Micro Focus COBOL<br>samples/cobol_mf | `checkerr.cbl` | `n/a` | `n/a` | `n/a` |

In order to use the utility functions, the utility file must first be compiled, and then its object file linked in during the creation of the target program's executable. Both the `makefile` and build files in the `samples` directories do this for the programs that require the error-checking utilities.

The following example demonstrates how the error-checking utilities are used in DB2 programs. The `utilemb.h` header file defines the `EMB_SQL_CHECK` macro for the functions `SqlInfoPrint()` and `TransRollback()`:

```
/* macro for embedded SQL checking */
#define EMB_SQL_CHECK(MSG_STR)                          \
SqlInfoPrint(MSG_STR, &sqlca, __LINE__, __FILE__);      \
if (sqlca.sqlcode < 0)                                  \
{                                                       \
  TransRollback();                                      \
  return 1;                                             \
}
```

`SqlInfoPrint()` checks the SQLCODE and prints out any available information related to the specific error encountered. It also points to where the error occurred in the source code. `TransRollback()` allows the utility file to safely rollback a transaction where an error has occurred. It uses the embedded SQL statement `EXEC`

SQL ROLLBACK. The following is an example of how the C program dbuse calls the utility functions by using the macro, supplying the value "Delete with host variables -- Execute" for the MSG_STR parameter of the SqlInfoPrint() function:

```
EXEC SQL DELETE FROM org
  WHERE deptnumb = :hostVar1 AND
        division = :hostVar2;
EMB_SQL_CHECK("Delete with host variables -- Execute");
```

The EMB_SQL_CHECK macro ensures that if the DELETE statement fails, the transaction will be safely rolled back, and an appropriate error message printed out.

Developers are encouraged to use and expand upon these error-checking utilities when creating their own DB2 programs.

**Related concepts:**

- "Building embedded SQL applications using the sample build script" on page 200
- "Sample files" in *Samples Topics*
- "Error information in the SQLCODE, SQLSTATE, and SQLWARN fields" on page 173

**Related tasks:**

- "Including SQLSTATE and SQLCODE host variables in embedded SQL applications" on page 75
- "Declaring the SQLCA for Error Handling" on page 50

## Building applications and routines written in C and C++

### Building applications and routines written in C and C++

Build scripts for various operating system platforms are provided with the product to allow for building of embedded SQL applications in C and C++. Aside from build scripts used to build applications there is a specific bldrtn script provided used to build routines (stored procedures and user defined functions). For applications and routines written in VisualAge®, configuration files are used to build the applications. The C application samples provided vary from tutorials to client level or instance level examples, they can be found in the sqllib/samples/c directory for UNIX and sqllib\samples\c directory for Windows.

**Related tasks:**

- "Building applications in C or C++ using the sample build script (UNIX)" on page 203
- "Building C and C++ routine code" on page 328
- "Building embedded SQL applications written in VisualAge C++ with configuration files" on page 207

### Building applications in C or C++ using the sample build script (UNIX)

DB2 provides build scripts for compiling and linking embedded SQL and DB2 administrative API programs in C or C++. These are located in the sqllib/samples/c directory for applications in C and sqllib/samples/cpp directory for applications in C++, along with sample programs that can be built with these files.

The build file, `bldapp`, contains the commands to build a DB2 application program.

The first parameter, `$1`, specifies the name of your source file. This is the only required parameter, and the only one needed for DB2 administrative API programs that do not contain embedded SQL. Building embedded SQL programs requires a connection to the database so three optional parameters are also provided: the second parameter, `$2`, specifies the name of the database to which you want to connect; the third parameter, `$3`, specifies the user ID for the database, and `$4` specifies the password.

For an embedded SQL program, `bldapp` passes the parameters to the precompile and bind script, `embprep`. If no database name is supplied, the default `sample` database is used. The user ID and password parameters are only needed if the instance where the program is built is different from the instance where the database is located.

**Procedure:**

The following examples show you how to build and run DB2 administrative API and embedded SQL applications.

**Building and running DB2 administrative API applications**

To build the DB2 administrative API sample program, `cli_info`, from the source file `cli_info.c` for C and `cli_info.C` for C++, enter:

```
bldapp cli_info
```

The result is an executable file, `cli_info`.

To run the executable file, enter the executable name:

```
cli_info
```

**Building and running embedded SQL applications**

There are three ways to build the embedded SQL application, `tbmod`, from the source file `tbmod.sqc` for C and `tbmod.sqC` for C++,:

1. If connecting to the sample database on the same instance, enter:
   ```
   bldapp tbmod
   ```
2. If connecting to another database on the same instance, also enter the database name:
   ```
   bldapp tbmod database
   ```
3. If connecting to a database on another instance, also enter the user ID and password of the database instance:
   ```
   bldapp tbmod database userid password
   ```

The result is an executable file, `tbmod`.

There are three ways to run this embedded SQL application:

1. If accessing the `sample` database on the same instance, simply enter the executable name:
   ```
   tbmod
   ```
2. If accessing another database on the same instance, enter the executable name and the database name:
   ```
   tbmod database
   ```

3. If accessing a database on another instance, enter the executable name, database name, and user ID and password of the database instance:

```
tbmod database userid password
```

**Related concepts:**

- "Building embedded SQL applications using the sample build script" on page 200

**Related tasks:**

- "Building routines in C or C++ using the sample build script (UNIX)" on page 330

**Related reference:**

- "AIX C embedded SQL and DB2 API applications compile and link options" on page 212
- "HP-UX C application compile and link options" on page 213
- "Linux C application compile and link options" on page 216
- "Solaris C application compile and link options" on page 218

**Related samples:**

- "bldapp -- Builds AIX C application programs (C)"
- "bldapp -- Builds HP-UX C applications (C)"
- "bldapp -- Builds Linux C applications (C)"
- "bldapp -- Builds Solaris C applications (C)"
- "cli_info.c -- Set and get information at the client level (C)"
- "embprep -- To prep and bind C/C++ and Micro Focus COBOL embedded SQL programs (C)"
- "tbmod.sqc -- How to modify table data (C)"

## Building C/C++ applications on Windows

DB2 provides build scripts for compiling and linking DB2 API and embedded SQL C/C++ programs. These are located in the `sqllib\samples\c` and `sqllib\samples\cpp` directories, along with sample programs that can be built with these files.

The batch file, `bldapp.bat`, contains the commands to build DB2 API and embedded SQL programs. It takes up to four parameters, represented inside the batch file by the variables `%1`, `%2`, `%3`, and `%4`.

The first parameter, `%1`, specifies the name of your source file. This is the only required parameter for programs that do not contain embedded SQL. Building embedded SQL programs requires a connection to the database so three additional parameters are also provided: the second parameter, `%2`, specifies the name of the database to which you want to connect; the third parameter, `%3`, specifies the user ID for the database, and `%4` specifies the password.

For an embedded SQL program, `bldapp` passes the parameters to the precompile and bind file, `embprep.bat`. If no database name is supplied, the default `sample` database is used. The user ID and password parameters are only needed if the instance where the program is built is different from the instance where the database is located.

**Procedure:**

The following examples show you how to build and run DB2 API and embedded SQL applications.

To build the DB2 API non-embedded SQL sample program, `cli_info`, from either the source file `cli_info.c`, in `sqllib\samples\c`, or from the source file `cli_info.cxx`, in `sqllib\samples\cpp`, enter:

```
bldapp cli_info
```

The result is an executable file, `cli_info.exe`. You can run the executable file by entering the executable name (without the extension) on the command line:

```
cli_info
```

**Building and running embedded SQL applications**

There are three ways to build the embedded SQL application, `tbmod`, from the C source file `tbmod.sqc` in `sqllib\samples\c`, or from the C++ source file `tbmod.sqx` in `sqllib\samples\cpp`:

1. If connecting to the sample database on the same instance, enter:
   ```
   bldapp tbmod
   ```
2. If connecting to another database on the same instance, also enter the database name:
   ```
   bldapp tbmod database
   ```
3. If connecting to a database on another instance, also enter the user ID and password of the database instance:
   ```
   bldapp tbmod database userid password
   ```

The result is an executable file `tbmod.exe`.

There are three ways to run this embedded SQL application:

1. If accessing the `sample` database on the same instance, simply enter the executable name:
   ```
   tbmod
   ```
2. If accessing another database on the same instance, enter the executable name and the database name:
   ```
   tbmod database
   ```
3. If accessing a database on another instance, enter the executable name, database name, and user ID and password of the database instance:
   ```
   tbmod database userid password
   ```

**Building and running multi-threaded applications**

C/C++ multi-threaded applications on Windows need to be compiled with either the `-MT` or `-MD` options. The `-MT` option will link using the static library `LIBCMT.LIB`, and `-MD` will link using the dynamic library `MSVCRT.LIB`. The binary linked with `-MD` will be smaller but dependent on `MSVCRT.DLL`, while the binary linked with `-MT` will be larger but will be self-contained with respect to the runtime.

The batch file `bldmt.bat` uses the `-MT` option to build a multi-threaded program. All other compile and link options are the same as those used by the batch file `bldapp.bat` to build regular standalone applications.

To build the multi-threaded sample program, `dbthrds`, from either the `samples\c\dbthrds.sqc` or `samples\cpp\dbthrds.sqx` source file, enter:

```
bldmt dbthrds
```

The result is an executable file, `dbthrds.exe`.

There are three ways to run this multi-threaded application:

1. If accessing the `sample` database on the same instance, simply enter the executable name (without the extension):
   ```
   dbthrds
   ```
2. If accessing another database on the same instance, enter the executable name and the database name:
   ```
   dbthrds database
   ```
3. If accessing a database on another instance, enter the executable name, database name, and user ID and password of the database instance:
   ```
   dbthrds database userid password
   ```

**Related reference:**

- "Windows C and C++ application compile and link options" on page 220

**Related samples:**

- "cli_info.C -- Set and get information at the client level (C++)"
- "dbthrds.sqC -- How to use multiple context APIs on Windows (C++)"
- "tbmod.sqC -- How to modify table data (C++)"
- "bldapp.bat -- Builds C++ applications on Windows"
- "bldmt.bat -- Builds C++ multi-threaded applications on Windows"
- "dbthrds.sqc -- How to use multiple context APIs on Windows (C)"
- "embprep.bat -- Prep and binds a C/C++ or Micro Focus COBOL embedded SQL program on Windows"
- "tbmod.sqc -- How to modify table data (C)"
- "bldapp.bat -- Builds C applications on Windows"
- "bldmt.bat -- Builds C multi-threaded applications on Windows"
- "cli_info.c -- Set and get information at the client level (C)"

## Building embedded SQL applications written in VisualAge C++ with configuration files

VisualAge C++ has both an incremental compiler and a batch mode compiler. While the batch mode compiler uses makefiles and build files, the incremental compiler uses configuration files instead. See the documentation that comes with VisualAge C++ Version 5.0 to learn more about this.

DB2 provides configuration files for the different types of DB2 programs you can build with the VisualAge C++ compiler.

**Procedure:**

To use a DB2 configuration file, you first set an environment variable to the program name you want to compile. Then you compile the program with a command supplied by VisualAge C++. Here are the topics describing how you can use the configuration files provided by DB2 to compile different types of programs:

- Building embedded SQL and DB2 API applications in C or C++ with configuration files (AIX)
- Building embedded SQL stored procedures in C or C++ with configuration files
- Building user-defined functions in C or C++ with configuration files (AIX)

**Related tasks:**
- "Building embedded SQL and DB2 API applications in C or C++ with configuration files (AIX)" on page 208
- "Building embedded SQL stored procedures in C or C++ with configuration files" on page 348
- "Building user-defined functions in C or C++ with configuration files (AIX)" on page 350
- "Building applications in C or C++ using the sample build script (UNIX)" on page 203
- "Building routines in C or C++ using the sample build script (UNIX)" on page 330

## Building embedded SQL and DB2 API applications in C or C++ with configuration files (AIX)

Configuration files are essentially build scripts used to build applications written in VisualAge. The script does many things which includes error checking, connecting to a database, precomiling the code, setting the path where DB2 can be accessed and finally producing the executable file.

**Procedure:**

The following examples show you how to build and run DB2 administrative API and embedded SQL applications with configuration files.

**Building C and C++ embedded SQL applications with configuration files**

The configuration file, `emb.icc`, in `sqllib/samples/c` and `sqllib/samples/cpp`, allows you to build DB2 embedded SQL applications in C and C++ on AIX.

To use the configuration file to build the embedded SQL application `tbmod` from the source file `tbmod.sqc`, do the following:

1. Set the EMB environment variable to the program name by entering:
   - For bash or Korn shell:
     ```
     export EMB=tbmod
     ```
   - For C shell:
     ```
     setenv EMB tbmod
     ```
2. If you have an `emb.ics` file in your working directory, produced by building a different program with the `emb.icc` file, delete the `emb.ics` file with this command:
   ```
   rm emb.ics
   ```

   An existing `emb.ics` file produced for the same program you are going to build again does not have to be deleted.
3. Compile the sample program by entering:
   ```
   vacbld emb.icc
   ```

   **Note:** The `vacbld` command is provided by VisualAge C++.

The result is an executable file, tbmod. You can run the program by entering the executable name:

```
tbmod
```

**Building C and C++ DB2 API applications with configuration files**

The configuration file, api.icc, in sqllib/samples/c and in sqllib/samples/cpp, allows you to build DB2 administrative API programs in C or C++ on AIX.

To use the configuration file to build the DB2 administrative API sample program cli_info from the source file cli_info.c, do the following:

1. Set the API environment variable to the program name by entering:
   - For bash or Korn shell:
     ```
     export API=cli_info
     ```
   - For C shell:
     ```
     setenv API cli_info
     ```
2. If you have an api.ics file in your working directory, produced by building a different program with the api.icc file, delete the api.ics file with this command:
   ```
   rm api.ics
   ```

   An existing api.ics file produced for the same program you are going to build again does not have to be deleted.
3. Compile the sample program by entering:
   ```
   vacbld api.icc
   ```

   **Note:** The vacbld command is provided by VisualAge C++.

The result is an executable file, cli_info. You can run the program by entering the executable name:

```
cli_info
```

**Related tasks:**

- "Building embedded SQL stored procedures in C or C++ with configuration files" on page 348
- "Building user-defined functions in C or C++ with configuration files (AIX)" on page 350
- "Setting up the embedded SQL development environment" on page 11

## Building C/C++ multi-connection applications on Windows

DB2 provides build scripts for compiling and linking C and C++ embedded SQL and DB2 API programs. These are located in the sqllib\samples\c and sqllib\samples\cpp directories, along with sample programs that can be built with these files.

The batch file, bldmc.bat, contains the commands to build a DB2 multi-connection program, requiring two databases. The compile and link options are the same as those used in the bldapp.bat file.

The first parameter, %1, specifies the name of your source file. The second parameter, %2, specifies the name of the first database to which you want to connect. The third parameter, `%3`, specifies the second database to which you want to connect. These are all required parameters.

**Note:** The build script hardcodes default values of "sample" and "sample2" for the database names (`%2` and `%3`, respectively) so if you are using the build script, and accept these defaults, you only have to specify the program name (the `%1` parameter). If you are using the `bldmc.bat` script, you must specify all three parameters.

Optional parameters are not required for a local connection, but are required for connecting to a server from a remote client. These are: `%4` and `%5` to specify the user ID and password, respectively, for the first database; and `%6` and `%7` to specify the user ID and password, respectively, for the second database.

**Procedure:**

For the multi-connection sample program, `dbmcon.exe`, you require two databases. If the `sample` database is not yet created, you can create it by entering `db2sampl` on the command line of a DB2 command window. The second database, here called `sample2`, can be created with one of the following commands:

If creating the database locally:

```
db2 create db sample2
```

If creating the database remotely:

```
db2 attach to <node_name>
db2 create db sample2
db2 detach
db2 catalog db sample2 as sample2 at node <node_name>
```

where `<node_name>` is the node where the database resides.

Multi-connection also requires that the TCP/IP listener is running. To ensure it is, do the following:

1. Set the environment variable DB2COMM to TCP/IP as follows:
   ```
   db2set DB2COMM=TCPIP
   ```
2. Update the database manager configuration file with the TCP/IP service name as specified in the services file:
   ```
   db2 update dbm cfg using SVCENAME <TCP/IP service name>
   ```
   Each instance has a TCP/IP service name listed in the services file. Ask your system administrator if you cannot locate it or do not have the file permission to change the services file.
3. Stop and restart the database manager in order for these changes to take effect:
   ```
   db2stop
   db2start
   ```

The `dbmcon.exe` program is created from five files in either the `samples\c` or `samples\cpp` directories:

**`dbmcon.sqc` or `dbmcon.sqx`**
Main source file for connecting to both databases.

**`dbmcon1.sqc` or `dbmcon1.sqx`**
Source file for creating a package bound to the first database.

**`dbmcon1.h`**
: Header file for `dbmcon1.sqc` or `dbmcon1.sqx` included in the main source file, `dbmcon.sqc` or `dbmcon.sqx`, for accessing the SQL statements for creating and dropping a table bound to the first database.

**`dbmcon2.sqc` or `dbmcon2.sqx`**
: Source file for creating a package bound to the second database.

**`dbmcon2.h`**
: Header file for `dbmcon2.sqc` or `dbmcon2.sqx` included in the main source file, `dbmcon.sqc` or `dbmcon.sqx`, for accessing the SQL statements for creating and dropping a table bound to the second database.

To build the multi-connection sample program, `dbmcon.exe`, enter:

```
bldmc dbmcon sample sample2
```

The result is an executable file, `dbmcon.exe`.

To run the executable file, enter the executable name, without the extension:

```
dbmcon
```

The program demonstrates a one-phase commit to two databases.

**Related concepts:**
- "Building embedded SQL applications using the sample build script" on page 200

**Related reference:**
- "Windows C and C++ application compile and link options" on page 220
- "svcename - TCP/IP service name configuration parameter" in *Performance Guide*

**Related samples:**
- "bldmc.bat -- Builds C multi-connection application on Windows"
- "dbmcon.sqc -- How to use multiple databases (C)"
- "dbmcon1.h -- Function declarations for the source file, dbmcon1.sqc (C)"
- "dbmcon1.sqc -- Functions used in the multiple databases program dbmcon.sqc (C)"
- "dbmcon2.h -- Function declarations for the source file, dbmcon2.sqc (C)"
- "dbmcon2.sqc -- Functions used in the multiple databases program dbmcon.sqc (C)"
- "bldmc.bat -- Builds C++ multi-connection application on Windows"
- "dbmcon.sqC -- How to use multiple databases (C++)"
- "dbmcon1.h -- Class declaration for the source file, dbmcon1.sqC (C++)"
- "dbmcon1.sqC -- Functions used in the multiple databases program dbmcon.sqC (C++)"
- "dbmcon2.h -- Class declaration for the source file, dbmcon2.sqC (C++)"
- "dbmcon2.sqC -- Functions used in the multiple databases program dbmcon.sqC (C++)"

## AIX C embedded SQL and DB2 API applications compile and link options

The following are the compile and link options recommended by DB2 for building C embedded SQL and DB2 API applications with the AIX IBM C compiler, as demonstrated in the `bldapp` build script.

**Compile and link options for bldapp:**

| | |
|---|---|
| Compile Options: | |
| **`xlc`** | The IBM XL C/C++ compiler. |
| **`$EXTRA_CFLAG`** | Contains ″-q64″ for an instance where 64-bit support is enabled; otherwise, it contains no value. |
| **`-I$DB2PATH/include`** | Specify the location of the DB2 include files. For example: `$HOME/sqllib/include`. |
| **`-c`** | Perform compile only; no link. Compile and link are separate steps. |
| Link Options: | |
| **`xlc`** | Use the compiler as a front end for the linker. |
| **`$EXTRA_CFLAG`** | Contains ″-q64″ for an instance where 64-bit support is enabled; otherwise, it contains no value. |
| **`-o $1`** | Specify the executable program. |
| **`$1.o`** | Specify the program object file. |
| **`utilemb.o`** | If an embedded SQL program, include the embedded SQL utility object file for error checking. |
| **`utilapi.o`** | If not an embedded SQL program, include the DB2 API utility object file for error checking. |
| **`-ldb2`** | Link to the DB2 library. |
| **`-L$DB2PATH/$LIB`** | Specify the location of the DB2 runtime shared libraries. For example: `$HOME/sqllib/$LIB`. If you do not specify the `-L` option, the compiler assumes the following path: `/usr/lib:/lib`. |
| Refer to your compiler documentation for additional compiler options. | |

**Related tasks:**

- “Building applications in C or C++ using the sample build script (UNIX)” on page 203

**Related reference:**

- “AIX C routine compile and link options” on page 338

**Related samples:**

- “bldapp -- Builds AIX C application programs (C)”

## AIX C++ embedded SQL and DB2 administrative API applications compile and link options

The following are the compile and link options recommended by DB2 for building C++ embedded SQL and DB2 administrative API applications with the AIX IBM XL C/C++ compiler, as demonstrated in the `bldapp` build script.

**Compile and link options for bldapp:**

Compile options:

**`xlC`** The IBM XL C/C++ compiler.

**`EXTRA_CFLAG`**
Contains ″-q64″ for an instance where 64-bit support is enabled; otherwise, it contains no value.

**`-I$DB2PATH/include`**
Specify the location of the DB2 include files. For example: `$HOME/sqllib/include`.

**`-c`** Perform compile only; no link. Compile and link are separate steps.

Link options:

**`xlC`** Use the compiler as a front end for the linker.

**`EXTRA_CFLAG`**
Contains ″-q64″ for an instance where 64-bit support is enabled; otherwise, it contains no value.

**`-o $1`** Specify the executable program.

**`$1.o`** Specify the program object file.

**`utilapi.o`**
Include the API utility object file for non-embedded SQL programs.

**`utilemb.o`**
Include the embedded SQL utility object file for embedded SQL programs.

**`-ldb2`** Link with the DB2 library.

**`-L$DB2PATH/$LIB`**
Specify the location of the DB2 runtime shared libraries. For example: `$HOME/sqllib/$LIB`. If you do not specify the `-L` option, the compiler assumes the following path `/usr/lib:/lib`.

Refer to your compiler documentation for additional compiler options.

**Related tasks:**

- "Building embedded SQL and DB2 API applications in C or C++ with configuration files (AIX)" on page 208

**Related reference:**

- "AIX C++ routine compile and link options" on page 339

**Related samples:**

- "bldapp -- Builds AIX C++ applications (C++)"

## HP-UX C application compile and link options

The following are the compile and link options recommended by DB2 for building C embedded SQL and DB2 API applications with the HP-UX C compiler, as demonstrated in the `bldapp` build script.

**Compile and link options for bldapp:**

Compile options:

**cc** The C compiler.

**$EXTRA_CFLAG**
: If the HP-UX platform is IA64 and 64-bit support is enabled, this flag contains the value **+DD64**; if 32-bit support is enabled, it contains the value **+DD32**. If the HP-UX platform is PA-RISC and 64-bit support is enabled, it contains the value **+DA2.0W**. For 32-bit support on a PA-RISC platform, this flag contains the value **+DA2.0N**.
  - **+DD64** Must be used to generate 64-bit code for HP-UX on IA64.
  - **+DD32** Must be used to generate 32-bit code for HP-UX on IA64.
  - **+DA2.0W** Must be used to generate 64-bit code for HP-UX on PA-RISC.
  - **+DA2.0N** Must be used to generate 32-bit code for HP-UX on PA-RISC.

**-Ae** Enables HP ANSI extended mode.

**-I$DB2PATH/include**
: Specifies the location of the DB2 include files.

**-c** Perform compile only; no link. Compile and link are separate steps.

Link options:

**cc** Use the compiler as a front end to the linker.

**$EXTRA_CFLAG**
: If the HP-UX platform is IA64 and 64-bit support is enabled, this flag contains the value **+DD64**; if 32-bit support is enabled, it contains the value **+DD32**. If the HP-UX platform is PA-RISC and 64-bit support is enabled, it contains the value **+DA2.0W**. For 32-bit support on a PA-RISC platform, this flag contains the value **+DA2.0N**.
  - **+DD64** Must be used to generate 64-bit code for HP-UX on IA64.
  - **+DD32** Must be used to generate 32-bit code for HP-UX on IA64.
  - **+DA2.0W** Must be used to generate 64-bit code for HP-UX on PA-RISC.
  - **+DA2.0N** Must be used to generate 32-bit code for HP-UX on PA-RISC.

**-o $1** Specify the executable.

**$1.o** Specify the program object file.

**utilemb.o**
: If an embedded SQL program, include the embedded SQL utility object file for error checking.

**utilapi.o**
: If a non-embedded SQL program, include the DB2 API utility object file for error checking.

**$EXTRA_LFLAG**
: Specify the runtime path. If set, for 32-bit it contains the value `-Wl,+b$HOME/sqllib/lib32`, and for 64-bit: `-Wl,+b$HOME/sqllib/lib64`. If not set, it contains no value.

**-L$DB2PATH/$LIB**
: Specify the location of the DB2 runtime shared libraries. For 32-bit: `$HOME/sqllib/lib32`; for 64-bit: `$HOME/sqllib/lib64`.

**-ldb2** Link with the DB2 library.

Refer to your compiler documentation for additional compiler options.

**Related tasks:**

- "Building applications in C or C++ using the sample build script (UNIX)" on page 203

**Related samples:**

- "bldapp -- Builds HP-UX C applications (C)"

## HP-UX C++ application compile and link options

The following are the compile and link options recommended by DB2 for building C++ embedded SQL and DB2 API applications with the HP-UX C++ compiler, as demonstrated in the `bldapp` build script.

**Compile and link options for bldapp:**

Compile options:

**`aCC`** The HP aC++ compiler.

**`$EXTRA_CFLAG`**
If the HP-UX platform is IA64 and 64-bit support is enabled, this flag contains the value **`+DD64`**; if 32-bit support is enabled, it contains the value **`+DD32`**. If the HP-UX platform is PA-RISC and 64-bit support is enabled, it contains the value **`+DA2.0W`**. For 32-bit support on a PA-RISC platform, this flag contains the value **`+DA2.0N`**.

- **`+DD64`** Must be used to generate 64-bit code for HP-UX on IA64.
- **`+DD32`** Must be used to generate 32-bit code for HP-UX on IA64.
- **`+DA2.0W`**
  Must be used to generate 64-bit code for HP-UX on PA-RISC.
- **`+DA2.0N`**
  Must be used to generate 32-bit code for HP-UX on PA-RISC.

**`-ext`** Allows various C++ extensions including ″long long″ support.

**`-I$DB2PATH/include`**
Specifies the location of the DB2 include files. For example: `$HOME/sqllib/include`

**`-c`** Perform compile only; no link. Compile and link are separate steps.

Link options:

**`aCC`** Use the HP aC++ compiler as a front end for the linker.

**`$EXTRA_CFLAG`**
If the HP-UX platform is IA64 and 64-bit support is enabled, this flag contains the value **`+DD64`**; if 32-bit support is enabled, it contains the value **`+DD32`**. If the HP-UX platform is PA-RISC and 64-bit support is enabled, it contains the value **`+DA2.0W`**. For 32-bit support on a PA-RISC platform, this flag contains the value **`+DA2.0N`**.

- **`+DD64`** Must be used to generate 64-bit code for HP-UX on IA64.
- **`+DD32`** Must be used to generate 32-bit code for HP-UX on IA64.
- **`+DA2.0W`**
  Must be used to generate 64-bit code for HP-UX on PA-RISC.
- **`+DA2.0N`**
  Must be used to generate 64-bit code for HP-UX on PA-RISC.

**`-o $1`** Specify the executable.

**`$1.o`** Specify the program object file.

**`utilemb.o`**
If an embedded SQL program, include the embedded SQL utility object file for error checking.

**`utilapi.o`**
If a non-embedded SQL program, include the DB2 API utility object file for error checking.

**`$EXTRA_LFLAG`**
Specify the runtime path. If set, for 32-bit it contains the value ″`-Wl,+b$HOME/sqllib/lib32`″, and for 64-bit: ″`-Wl,+b$HOME/sqllib/lib64`″. If not set, it contains no value.

**`-L$DB2PATH/$LIB`**
Specify the location of the DB2 runtime shared libraries. For 32-bit: `$HOME/sqllib/lib32`; for 64-bit: `$HOME/sqllib/lib64`.

**`-ldb2`** Link with the DB2 library.

Refer to your compiler documentation for additional compiler options.

**Related concepts:**

- “Building applications and routines written in C and C++” on page 203

**Related samples:**

- "bldapp -- Builds HP-UX C++ applications (C++)"

## Linux C application compile and link options

The following are the compile and link options recommended by DB2 for building C embedded SQL and DB2 API applications with the Linux C compiler, as demonstrated in the `bldapp` build script.

**Compile and link options for bldapp:**

Compile options:

**`$CC`** The gcc or xlc_r compiler.

**`$EXTRA_C_FLAGS`**
Contains one of the following:
- `-m31` on Linux for zSeries only, to build a 32-bit library;
- `-m32` on Linux for x86, x86_64 and POWER, to build a 32-bit library;
- `-m64` on Linux for zSeries, POWER, x86_64, to build a 64-bit library; or
- No value on Linux for IA64, to build a 64-bit library.

**`-I$DB2PATH/include`**
Specify the location of the DB2 include files.

**`-c`** Perform compile only; no link. This script file has separate compile and link steps.

Link options:

**`$CC`** The gcc or xlc_r compiler; use the compiler as a front end for the linker.

**`$EXTRA_C_FLAGS`**
Contains one of the following:
- `-m31` on Linux for zSeries only, to build a 32-bit library;
- `-m32` on Linux for x86, x86_64 and POWER, to build a 32-bit library;
- `-m64` on Linux for zSeries, POWER, x86_64, to build a 64-bit library; or
- No value on Linux for IA64, to build a 64-bit library.

**`-o $1`** Specify the executable.

**`$1.o`** Specify the object file.

**`utilemb.o`**
If an embedded SQL program, include the embedded SQL utility object file for error checking.

**`utilapi.o`**
If a non-embedded SQL program, include the DB2 API utility object file for error checking.

**`$EXTRA_LFLAG`**
For 32-bit it contains the value ″-Wl,-rpath,$DB2PATH/lib32″, and for 64-bit it contains the value ″-Wl,-rpath,$DB2PATH/lib64″.

**`-L$DB2PATH/$LIB`**
Specify the location of the DB2 static and shared libraries at link-time. For example, for 32-bit: `$HOME/sqllib/lib32`, and for 64-bit: `$HOME/sqllib/lib64`.

**`-ldb2`** Link with the DB2 library.

Refer to your compiler documentation for additional compiler options.

**Related tasks:**

- "Building applications in C or C++ using the sample build script (UNIX)" on page 203

**Related samples:**

- "bldapp -- Builds Linux C applications (C)"

## Linux C++ application compile and link options

The following are the compile and link options recommended by DB2 for building C++ embedded SQL and DB2 API applications with the Linux C++ compiler, as demonstrated in the `bldapp` build script.

**Compile and link options for bldapp:**

| | |
|---|---|
| Compile options: | |
| **`g++`** | The GNU/Linux C++ compiler. |
| **`$EXTRA_C_FLAGS`** | Contains one of the following:<br>• `-m31` on Linux for zSeries only, to build a 32-bit library;<br>• `-m32` on Linux for x86, x86_64 and POWER, to build a 32-bit library;<br>• `-m64` on Linux for zSeries, POWER, x86_64, to build a 64-bit library; or<br>• No value on Linux for IA64, to build a 64-bit library. |
| **`-I$DB2PATH/include`** | Specify the location of the DB2 include files. |
| **`-c`** | Perform compile only; no link. This script file has separate compile and link steps. |
| Link options: | |
| **`g++`** | Use the compiler as a front end for the linker. |
| **`$EXTRA_C_FLAGS`** | Contains one of the following:<br>• `-m31` on Linux for zSeries only, to build a 32-bit library;<br>• `-m32` on Linux for x86, x86_64 and POWER, to build a 32-bit library;<br>• `-m64` on Linux for zSeries, POWER, x86_64, to build a 64-bit library; or<br>• No value on Linux for IA64, to build a 64-bit library. |
| **`-o $1`** | Specify the executable. |
| **`$1.o`** | Include the program object file. |
| **`utilemb.o`** | If an embedded SQL program, include the embedded SQL utility object file for error checking. |
| **`utilapi.o`** | If a non-embedded SQL program, include the DB2 API utility object file for error checking. |
| **`$EXTRA_LFLAG`** | For 32-bit it contains the value ″-Wl,-rpath,$DB2PATH/lib32″, and for 64-bit it contains the value ″-Wl,-rpath,$DB2PATH/lib64″. |
| **`-L$DB2PATH/$LIB`** | Specify the location of the DB2 static and shared libraries at link-time. For example, for 32-bit: `$HOME/sqllib/lib32`, and for 64-bit: `$HOME/sqllib/lib64`. |
| **`-ldb2`** | Link with the DB2 library. |
| Refer to your compiler documentation for additional compiler options. | |

**Related concepts:**

- "Building applications and routines written in C and C++" on page 203

**Related samples:**

- "bldapp -- Builds Linux C++ applications (C++)"

## Solaris C application compile and link options

These are the compile and link options recommended by DB2 for building C embedded SQL and DB2 API applications with the Forte C compiler, as demonstrated in the `bldapp` build script.

**Compile and link options for bldapp:**

|  |  |
|---|---|
| Compile options: | |
| **`cc`** | The C compiler. |
| **`-xarch=$CFLAG_ARCH`** | This option ensures that the compiler will produce valid executables when linking with `libdb2.so`. The value for $CFLAG_ARCH is set to either ″v8plusa″ for 32-bit, or ″v9″ for 64-bit. |
| **`-I$DB2PATH/include`** | Specify the location of the DB2 include files. For example: `$HOME/sqllib/include` |
| **`-c`** | Perform compile only; no link. This script has separate compile and link steps. |

Link options:

**cc** Use the compiler as a front end for the linker.

**-xarch=$CFLAG_ARCH**
This option ensures that the compiler will produce valid executables when linking with `libdb2.so`. The value for $CFLAG_ARCH is set to either ″v8plusa″ for 32-bit, or ″v9″ for 64-bit.

**-mt** Link in multi-thread support. Needed for linking with libdb2.
**Note:** If POSIX threads are used, DB2 applications also have to link with `-lpthread`, whether or not they are threaded.

**-o $1** Specify the executable.

**$1.o** Include the program object file.

**utilemb.o**
If an embedded SQL program, include the embedded SQL utility object file for error checking.

**utilapi.o**
If not an embedded SQL program, include the DB2 API utility object file for error checking.

**-L$DB2PATH/$LIB**
Specify the location of the DB2 static and shared libraries at link-time. For example, for 32-bit: `$HOME/sqllib/lib32`, and for 64-bit: `$HOME/sqllib/lib64`.

**$EXTRA_LFLAG**
Specify the location of the DB2 shared libraries at run-time. For 32-bit it contains the value ″-R$DB2PATH/lib32″, and for 64-bit it contains the value ″-R$DB2PATH/lib64″.

**-ldb2** Link with the DB2 library.

Refer to your compiler documentation for additional compiler options.

**Related tasks:**

- "Building applications in C or C++ using the sample build script (UNIX)" on page 203

**Related samples:**

- "bldapp -- Builds Solaris C applications (C)"

## Solaris C++ application compile and link options

These are the compile and link options recommended by DB2 for building C++ embedded SQL and DB2 API applications with the Forte C++ compiler, as demonstrated in the `bldapp` build script.

**Compile and link options for bldapp:**

Compile options:

**`CC`** The C++ compiler.

**`-xarch=$CFLAG_ARCH`**
This option ensures that the compiler will produce valid executables when linking with `libdb2.so`. The value for $CFLAG_ARCH is set to either ″v8plusa″ for 32-bit, or ″v9″ for 64-bit.

**`-I$DB2PATH/include`**
Specify the location of the DB2 include files. For example: `$HOME/sqllib/include`

**`-c`** Perform compile only; no link. This script has separate compile and link steps.

Link options:

**`CC`** Use the compiler as a front end for the linker.

**`-xarch=$CFLAG_ARCH`**
This option ensures that the compiler will produce valid executables when linking with `libdb2.so`. The value for $CFLAG_ARCH is set to either ″v8plusa″ for 32-bit, or ″v9″ for 64-bit.

**`-mt`** Link in multi-thread support. Needed for linking with libdb2.
**Note:** If POSIX threads are used, DB2 applications also have to link with `-lpthread`, whether or not they are threaded.

**`-o $1`** Specify the executable.

**`$1.o`** Include the program object file.

**`utilemb.o`**
If an embedded SQL program, include the embedded SQL utility object file for error checking.

**`utilapi.o`**
If a non-embedded SQL program, include the DB2 API utility object file for error checking.

**`-L$DB2PATH/$LIB`**
Specify the location of the DB2 static and shared libraries at link-time. For example, for 32-bit: `$HOME/sqllib/lib32`, and for 64-bit: `$HOME/sqllib/lib64`.

**`$EXTRA_LFLAG`**
Specify the location of the DB2 shared libraries at run-time. For 32-bit it contains the value ″-R$DB2PATH/lib32″, and for 64-bit it contains the value ″-R$DB2PATH/lib64″.

**`-ldb2`** Link with the DB2 library.

Refer to your compiler documentation for additional compiler options.

**Related tasks:**

- "Compiling and linking source files containing embedded SQL" on page 189

**Related samples:**

- "bldapp -- Builds Solaris C++ applications (C++)"

## Windows C and C++ application compile and link options

The following are the compile and link options recommended by DB2 for building C and C++ embedded SQL and DB2 API applications on Windows with the Microsoft Visual C++ compiler, as demonstrated in the `bldapp.bat` batch file.

**Compile and link options for bldapp:**

Compile options:

**`%BLDCOMP%`**
Variable for the compiler. The default is `cl`, the Microsoft Visual C++ compiler. It can be also set to `icl`, the Intel® C++ Compiler for 32-bit and 64-bit applications, or `ecl`, the Intel C++ Compiler for Itanium® 64-bit applications.

**`-Zi`** Enable debugging information

**`-Od`** Disable optimizations. It is easier to use a debugger with optimization off.

**`-c`** Perform compile only; no link. The batch file has separate compile and link steps.

**`-W2`** Output warning, error, and severe and unrecoverable error messages.

**`-DWIN32`**
Compiler option necessary for Windows operating systems.

Link options:

**`link`** Use the linker to link.

**`-debug`** Include debugging information.

**`-out:%1.exe`**
Specify a filename

**`%1.obj`** Include the object file

**`utilemb.obj`**
If an embedded SQL program, include the embedded SQL utility object file for error checking.

**`utilapi.obj`**
If not an embedded SQL program, include the DB2 API utility object file for error checking.

**`db2api.lib`**
Link with the DB2 library.

Refer to your compiler documentation for additional compiler options.

**Related tasks:**
- "Building C/C++ applications on Windows" on page 205

**Related samples:**
- "bldapp.bat -- Builds C++ applications on Windows"
- "bldapp.bat -- Builds C applications on Windows"

## Building applications and routines written in COBOL

### Building applications and routines written in COBOL

Build scripts for various operating system platforms are provided with the product to allow for building of embedded SQL applications in COBOL. Aside from build scripts used to build applications there is a specific `bldrtn` script provided used to build routines (stored procedures and user defined functions). When working with applications written in the Micro Focus COBOL language on Linux, be sure to configure the compiler to be able to access certain COBOL shared libraries. IBM COBOL samples are provided and can be found in the `sqllib/samples/cobol` directory for UNIX and `sqllib\samples\cobol` directory for Windows, for the Micro Focus COBOL samples directories replace the ’cobol’ at the end of the path with ’cobol_mf’.

**Related tasks:**
- “Building UNIX Micro Focus COBOL applications” on page 223
- “Building IBM COBOL applications on AIX” on page 222
- “Building Micro Focus COBOL applications on Windows” on page 226

**Related reference:**
- “COBOL samples” on page 373

## Building IBM COBOL applications on AIX

DB2 provides build scripts for compiling and linking IBM COBOL embedded SQL and DB2 administrative API programs. These are located in the `sqllib/samples/cobol` directory, along with sample programs that can be built with these files.

The build file, `bldapp` contains the commands to build a DB2 application program.

The first parameter, `$1`, specifies the name of your source file. This is the only required parameter for programs that do not contain embedded SQL. Building embedded SQL programs requires a connection to the database so three optional parameters are also provided: the second parameter, `$2`, specifies the name of the database to which you want to connect; the third parameter, `$3`, specifies the user ID for the database, and `$4` specifies the password.

For an embedded SQL program, `bldapp` passes the parameters to the precompile and bind script, `embprep`. If no database name is supplied, the default `sample` database is used. The user ID and password parameters are only needed if the instance where the program is built is different from the instance where the database is located.

**Procedure:**

To build the non-embedded SQL sample program `client` from the source file `client.cbl`, enter:

```
bldapp client
```

The result is an executable file `client`. You can run the executable file against the `sample` database by entering:

```
client
```

**Building and running embedded SQL applications**

There are three ways to build the embedded SQL application, `updat`, from the source file `updat.sqb`:

1. If connecting to the sample database on the same instance, enter:
   ```
   bldapp updat
   ```
2. If connecting to another database on the same instance, also enter the database name:
   ```
   bldapp updat database
   ```

3. If connecting to a database on another instance, also enter the user ID and password of the database instance:
   ```
   bldapp updat database userid password
   ```

The result is an executable file, `updat`.

There are three ways to run this embedded SQL application:
1. If accessing the `sample` database on the same instance, simply enter the executable name:
   ```
   updat
   ```
2. If accessing another database on the same instance, enter the executable name and the database name:
   ```
   updat database
   ```
3. If accessing a database on another instance, enter the executable name, database name, and user ID and password of the database instance:
   ```
   updat database userid password
   ```

**Related concepts:**
- "Building embedded SQL applications using the sample build script" on page 200

**Related tasks:**
- "Building IBM COBOL routines on AIX" on page 360

**Related reference:**
- "AIX IBM COBOL application compile and link options" on page 234
- "COBOL samples" on page 373

**Related samples:**
- "client.cbl -- How to set and query a client (IBM COBOL)"
- "updat.sqb -- How to update, delete and insert table data (IBM COBOL)"
- "bldapp -- Builds AIX COBOL applications"
- "embprep -- To prep and bind a COBOL embedded SQL sample on AIX"

## Building UNIX Micro Focus COBOL applications

DB2 provides build scripts for compiling and linking Micro Focus COBOL embedded SQL and DB2 administrative API programs. These are located in the `sqllib/samples/cobol_mf` directory, along with sample programs that can be built with these files.

The build file, `bldapp` contains the commands to build a DB2 application program.

The first parameter, `$1`, specifies the name of your source file. This is the only required parameter for programs that do not contain embedded SQL. Building embedded SQL programs requires a connection to the database so three optional parameters are also provided: the second parameter, `$2`, specifies the name of the database to which you want to connect; the third parameter, `$3`, specifies the user ID for the database, and `$4` specifies the password.

For an embedded SQL program, `bldapp` passes the parameters to the precompile and bind script, `embprep`. If no database name is supplied, the default `sample` database is used. The user ID and password parameters are only needed if the instance where the program is built is different from the instance where the database is located.

**Procedure:**

To build the non-embedded SQL sample program, `client`, from the source file `client.cbl`, enter:

```
bldapp client
```

The result is an executable file `client`. You can run the executable file against the `sample` database by entering:

```
client
```

**Building and Running Embedded SQL Applications**

There are three ways to build the embedded SQL application, `updat`, from the source file `updat.sqb`:

1. If connecting to the sample database on the same instance, enter:
   ```
   bldapp updat
   ```
2. If connecting to another database on the same instance, also enter the database name:
   ```
   bldapp updat database
   ```
3. If connecting to a database on another instance, also enter the user ID and password of the database instance:
   ```
   bldapp updat database userid password
   ```

The result is an executable file, `updat`.

There are three ways to run this embedded SQL application:

1. If accessing the `sample` database on the same instance, simply enter the executable name:
   ```
   updat
   ```
2. If accessing another database on the same instance, enter the executable name and the database name:
   ```
   updat database
   ```
3. If accessing a database on another instance, enter the executable name, database name, and user ID and password of the database instance:
   ```
   updat database userid password
   ```

**Related tasks:**

- “Building UNIX Micro Focus COBOL routines” on page 361

**Related reference:**

- “AIX Micro Focus COBOL application compile and link options” on page 234
- “HP-UX Micro Focus COBOL application compile and link options” on page 235
- “Linux Micro Focus COBOL application compile and link options” on page 237
- “Solaris Micro Focus COBOL application compile and link options” on page 236

**Related samples:**

- “bldapp -- Builds Linux Micro Focus COBOL applications”

- "bldapp -- Builds Solaris Micro Focus COBOL applications"
- "client.cbl -- How to set and query a client (MF COBOL)"
- "updat.sqb -- How to update, delete and insert table data (MF COBOL)"
- "bldapp -- Builds AIX Micro Focus COBOL applications"
- "bldapp -- Builds HP-UX Micro Focus COBOL applications"
- "embprep -- To prep and bind C/C++ and Micro Focus COBOL embedded SQL programs (C)"

## Building IBM COBOL applications on Windows

DB2 provides build scripts for compiling and linking DB2 API and embedded SQL programs. These are located in the `sqllib\samples\cobol` directory, along with sample programs that can be built with these files.

DB2 supports two precompilers for building IBM COBOL applications on Windows, the DB2 precompiler and the IBM COBOL precompiler. The default is the DB2 precompiler. The IBM COBOL precompiler can be selected by uncommenting the appropriate line in the batch file you are using. Precompilation with IBM COBOL is done by the compiler itself, using specific precompile options.

The batch file, `bldapp.bat`, contains the commands to build a DB2 application program. It takes up to four parameters, represented inside the batch file by the variables `%1`, `%2`, `%3`, and `%4`.

The first parameter, `%1`, specifies the name of your source file. This is the only required parameter for programs that do not contain embedded SQL. Building embedded SQL programs requires a connection to the database so three optional parameters are also provided: the second parameter, `%2`, specifies the name of the database to which you want to connect; the third parameter, `%3`, specifies the user ID for the database, and `%4` specifies the password.

For an embedded SQL program using the default DB2 precompiler, `bldapp.bat` passes the parameters to the precompile and bind file, `embprep.bat`.

For an embedded SQL program using the IBM COBOL precompiler, `bldapp.bat` copies the `.sqb` source file to a `.cbl` source file. The compiler performs the precompile on the `.cbl` source file with specific precompile options.

For either precompiler, if no database name is supplied, the default `sample` database is used. The user ID and password parameters are only needed if the instance where the program is built is different from the instance where the database is located.

**Procedure:**

The following examples show you how to build and run DB2 API and embedded SQL applications.

To build the non-embedded SQL sample program `client` from the source file `client.cbl`, enter:

```
bldapp client
```

The result is an executable file `client.exe`. You can run the executable file against the `sample` database by entering the executable name (without the extension):

```
client
```

**Building and running embedded SQL applications**

There are three ways to build the embedded SQL application, `updat`, from the source file `updat.sqb`:

1. If connecting to the sample database on the same instance, enter:
   ```
   bldapp updat
   ```
2. If connecting to another database on the same instance, also enter the database name:
   ```
   bldapp updat database
   ```
3. If connecting to a database on another instance, also enter the user ID and password of the database instance:
   ```
   bldapp updat database userid password
   ```

The result is an executable file, `updat`.

There are three ways to run this embedded SQL application:

1. If accessing the `sample` database on the same instance, simply enter the executable name:
   ```
   updat
   ```
2. If accessing another database on the same instance, enter the executable name and the database name:
   ```
   updat database
   ```
3. If accessing a database on another instance, enter the executable name, database name, and user ID and password of the database instance:
   ```
   updat database userid password
   ```

**Related concepts:**

- "Building embedded SQL applications using the sample build script" on page 200

**Related reference:**

- "COBOL samples" on page 373
- "Windows IBM COBOL application compile and link options" on page 237

**Related samples:**

- "bldapp.bat -- Builds Windows VisualAge COBOL applications"
- "client.cbl -- How to set and query a client (IBM COBOL)"
- "embprep.bat -- To prep and bind a COBOL embedded SQL program on Windows"
- "updat.sqb -- How to update, delete and insert table data (IBM COBOL)"

## Building Micro Focus COBOL applications on Windows

DB2 provides build scripts for compiling and linking DB2 API and embedded SQL programs. These are located in the `sqllib\samples\cobol_mf` directory, along with sample programs that can be built with these files.

The batch file `bldapp.bat` contains the commands to build a DB2 application program. It takes up to four parameters, represented inside the batch file by the variables `%1`, `%2`, `%3`, and `%4`.

The first parameter, `%1`, specifies the name of your source file. This is the only required parameter for programs that do not contain embedded SQL. Building embedded SQL programs requires a connection to the database so three optional parameters are also provided: the second parameter, `%2`, specifies the name of the database to which you want to connect; the third parameter, `%3`, specifies the user ID for the database, and `%4` specifies the password.

For an embedded SQL program, `bldapp` passes the parameters to the precompile and bind batch file, `embprep.bat`. If no database name is supplied, the default `sample` database is used. The user ID and password parameters are only needed if the instance where the program is built is different from the instance where the database is located.

**Procedure:**

The following examples show you how to build and run DB2 API and embedded SQL applications.

To build the non-embedded SQL sample program, `client`, from the source file `client.cbl`, enter:

```
bldapp client
```

The result is an executable file `client.exe`. You can run the executable file against the `sample` database by entering the executable name (without the extension):

```
client
```

**Building and Running Embedded SQL Applications**

There are three ways to build the embedded SQL application, `updat`, from the source file `updat.sqb`:

1. If connecting to the sample database on the same instance, enter:
   ```
   bldapp updat
   ```
2. If connecting to another database on the same instance, also enter the database name:
   ```
   bldapp updat database
   ```
3. If connecting to a database on another instance, also enter the user ID and password of the database instance:
   ```
   bldapp updat database userid password
   ```

The result is an executable file, `updat.exe`.

There are three ways to run this embedded SQL application:

1. If accessing the `sample` database on the same instance, simply enter the executable name (without the extension):
   ```
   updat
   ```
2. If accessing another database on the same instance, enter the executable name and the database name:
   ```
   updat database
   ```
3. If accessing a database on another instance, enter the executable name, database name, and user ID and password of the database instance:
   ```
   updat database userid password
   ```

**Related concepts:**

- “Building embedded SQL applications using the sample build script” on page 200

**Related reference:**
- “COBOL samples” on page 373
- “Windows Micro Focus COBOL application compile and link options” on page 238

**Related samples:**
- “bldapp.bat -- Builds Windows Micro Focus Cobol applications”
- “client.cbl -- How to set and query a client (MF COBOL)”
- “updat.sqb -- How to update, delete and insert table data (MF COBOL)”
- “embprep.bat -- Prep and binds a C/C++ or Micro Focus COBOL embedded SQL program on Windows”

## Configuring the IBM COBOL compiler on Windows

If you develop applications that contain embedded SQL and DB2 API calls, and you are using the IBM VisualAge COBOL compiler, there are several points to keep in mind.

**Procedure:**
- When you precompile your application with the DB2 precompiler, and use the command line processor command `db2 prep`, use the `target ibmcob` option.
- Do not use tab characters in your source files.
- Use the `PROCESS` and `CBL` keywords in your source files to set compile options. Place the keywords in columns 8 to 72 only.
- If your application contains only embedded SQL, but no DB2 API calls, you do not need to use the `pgmname(mixed)` compile option. If you use DB2 API calls, you must use the `pgmname(mixed)` compile option.
- If you are using the ″System/390 host data type support″ feature of the IBM VisualAge COBOL compiler, the DB2 include files for your applications are in the following directory:
  ```
  %DB2PATH%\include\cobol_i
  ```
  If you are building DB2 sample programs using the batch files provided, the include file path specified in the batch files must be changed to point to the `cobol_i` directory and not the `cobol_a` directory.

  If you are NOT using the ″System/390 host data type support″ feature of the IBM VisualAge COBOL compiler, or you are using an earlier version of this compiler, then the DB2 include files for your applications are in the following directory:
  ```
  %DB2PATH%\include\cobol_a
  ```
  The `cobol_a` directory is the default.
- Specify COPY file names to include the `.cbl` extension as follows:
  ```
  COPY "sql.cbl".
  ```

**Related tasks:**
- “Building IBM COBOL applications on Windows” on page 225
- “Building IBM COBOL routines on Windows” on page 362

**Related reference:**

- “Windows IBM COBOL application compile and link options” on page 237
- “Windows IBM COBOL routine compile and link options” on page 358

## Configuring the Micro Focus COBOL compiler on Windows

If you develop applications that contain embedded SQL and DB2 API calls, and you are using the Micro Focus compiler, there are several points to keep in mind.

**Procedure:**

- When you precompile your application using the command line processor command `db2 prep`, use the `target mfcob` option.
- Ensure that the LIB environment variable points to `%DB2PATH%\lib` by using the following command:

  ```
  set LIB="%DB2PATH%\lib;%LIB%"
  ```
- The DB2 COPY files for Micro Focus COBOL reside in `%DB2PATH%\include\cobol_mf`. Set the `COBCPY` environment variable to include the directory as follows:

  ```
  set COBCPY="%DB2PATH%\include\cobol_mf;%COBCPY%"
  ```

  You must ensure that the above mentioned environment variables are permanently set in the `System` settings. This can be checked by going through the following steps:

  1. Open the **Control Panel**
  2. Select **System**
  3. Select the **Advanced** tab
  4. Click **Environment Variables**
  5. Check the **System variables** list for the required environment variables. If not present, add them to the **System variables** list

  Setting them in either the `User` settings, at a command prompt, or in a script is insufficient.

You must make calls to all DB2 application programming interfaces using calling convention 74. The DB2 COBOL precompiler automatically inserts a CALL-CONVENTION clause in a SPECIAL-NAMES paragraph. If the SPECIAL-NAMES paragraph does not exist, the DB2 COBOL precompiler creates it, as follows:

```
Identification Division
Program-ID. "static".
special-names.
    call-convention 74 is DB2API.
```

Also, the precompiler automatically places the symbol DB2API, which is used to identify the calling convention, after the ″call″ keyword whenever a DB2 API is called. This occurs, for instance, whenever the precompiler generates a DB2 API run-time call from an embedded SQL statement.

If calls to DB2 APIs are made in an application which is not precompiled, you should manually create a SPECIAL-NAMES paragraph in the application, similar to that given above. If you are calling a DB2 API directly, then you will need to manually add the DB2API symbol after the ″call″ keyword.

**Related tasks:**

- “Building Micro Focus COBOL applications on Windows” on page 226
- “Building Micro Focus COBOL routines on Windows” on page 364

**Related reference:**

- "Windows Micro Focus COBOL application compile and link options" on page 238
- "Windows Micro Focus COBOL routine compile and link options" on page 359

## Configuring the Micro Focus COBOL compiler on Linux

**Procedure:**

To run Micro Focus COBOL routines, the Linux run-time linker must be able to access certain COBOL shared libraries, and DB2 must be able to load these libraries. Since the program that does this loading runs with setuid privileges, it will only look for the dependent libraries in `/usr/lib`.

Create symbolic links to `/usr/lib` for the COBOL shared libraries. This must be done as root. The simplest way to do this is to link all COBOL library files from `$COBDIR/lib` to `/usr/lib`:

```
ln -s $COBDIR/lib/libcob* /usr/lib
```

where `$COBDIR` is where Micro Focus COBOL is installed, usually `/opt/lib/mfcobol`.

Here are the commands to link each individual file (assuming Micro Focus COBOL is installed in `/opt/lib/mfcobol`):

```
ln -s /opt/lib/mfcobol/lib/libcobrts.so /usr/lib
ln -s /opt/lib/mfcobol/lib/libcobrts_t.so /usr/lib
ln -s /opt/lib/mfcobol/lib/libcobrts.so.2 /usr/lib
ln -s /opt/lib/mfcobol/lib/libcobrts_t.so.2 /usr/lib
ln -s /opt/lib/mfcobol/lib/libcobcrtn.so /usr/lib
ln -s /opt/lib/mfcobol/lib/libcobcrtn.so.2 /usr/lib
ln -s /opt/lib/mfcobol/lib/libcobmisc.so /usr/lib
ln -s /opt/lib/mfcobol/lib/libcobmisc_t.so /usr/lib
ln -s /opt/lib/mfcobol/lib/libcobmisc.so.2 /usr/lib
ln -s /opt/lib/mfcobol/lib/libcobmisc_t.so.2 /usr/lib
ln -s /opt/lib/mfcobol/lib/libcobscreen.so /usr/lib
ln -s /opt/lib/mfcobol/lib/libcobscreen.so.2 /usr/lib
ln -s /opt/lib/mfcobol/lib/libcobtrace.so /usr/lib
ln -s /opt/lib/mfcobol/lib/libcobtrace_t.so /usr/lib
ln -s /opt/lib/mfcobol/lib/libcobtrace.so.2 /usr/lib
ln -s /opt/lib/mfcobol/lib/libcobtrace_t.so.2 /usr/lib
```

The following need to be done on each DB2 instance:

- When you precompile your application using the command line processor command `db2 prep`, use the `target mfcob` option.
- You must include the DB2 COBOL COPY file directory in the Micro Focus COBOL environment variable COBCPY. The COBCPY environment variable specifies the location of the COPY files. The DB2 COPY files for Micro Focus COBOL reside in `sqllib/include/cobol_mf` under the database instance directory.

  To include the directory, enter:
  - On bash or Korn shell:
    ```
    export COBCPY=$HOME/sqllib/include/cobol_mf:$COBDIR/cpylib
    ```
  - On C shell:
    ```
    setenv COBCPY $HOME/sqllib/include/cobol_mf:$COBDIR/cpylib
    ```
- Update the environment variable:
  - On bash or Korn shell:

```
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:$HOME/sqllib/lib:$COBDIR/lib
```

  – On C shell:

```
setenv LD_LIBRARY_PATH $LD_LIBRARY_PATH:$HOME/sqllib/lib:$COBDIR/lib
```

- Set the DB2 Environment List:

```
db2set DB2ENVLIST="COBDIR LD_LIBRARY_PATH"
```

**Note:** You might want to set COBCPY, COBDIR, and LD_LIBRARY_PATH in the `.bashrc`, `.kshrc` (depending on shell being used), `.bash_profile`, `.profile` (depending on shell being used), or in the `.login`. .

**Related tasks:**

- "Building UNIX Micro Focus COBOL applications" on page 223
- "Building UNIX Micro Focus COBOL routines" on page 361

**Related reference:**

- "Linux Micro Focus COBOL application compile and link options" on page 237
- "Linux Micro Focus COBOL routine compile and link options" on page 357

## Configuring the IBM COBOL compiler on AIX

The following are steps you need to take if you develop applications that contain embedded SQL and DB2 API calls, and you are using the IBM COBOL Set for AIX compiler.

**Procedure:**

- When you precompile your application using the command line processor command `db2 prep`, use the `target ibmcob` option.
- Do not use tab characters in your source files.
- You can use the `PROCESS` and `CBL` keywords in the first line of your source files to set compile options.
- If your application contains only embedded SQL, but no DB2 API calls, you do not need to use the `pgmname(mixed)` compile option. If you use DB2 API calls, you must use the `pgmname(mixed)` compile option.
- If you are using the "System/390 host data type support" feature of the IBM COBOL Set for AIX compiler, the DB2 include files for your applications are in the following directory:

  ```
  $HOME/sqllib/include/cobol_i
  ```

  If you are building DB2 sample programs using the script files provided, the include file path specified in the script files must be changed to point to the `cobol_i` directory and not the `cobol_a` directory.

  If you are NOT using the "System/390 host data type support" feature of the IBM COBOL Set for AIX compiler, or you are using an earlier version of this compiler, then the DB2 include files for your applications are in the following directory:

  ```
  $HOME/sqllib/include/cobol_a
  ```

  Specify COPY file names to include the `.cbl` extension as follows:

  ```
  COPY "sql.cbl".
  ```

**Related concepts:**

- “Considerations for installing COBOL on AIX” in *Developing SQL and External Routines*

**Related tasks:**
- “Building IBM COBOL applications on AIX” on page 222
- “Building IBM COBOL routines on AIX” on page 360

**Related reference:**
- “AIX IBM COBOL application compile and link options” on page 234
- “AIX IBM COBOL routine compile and link options” on page 354

## Configuring the Micro Focus COBOL compiler on AIX

Do the following if you develop applications that contain embedded SQL and DB2 API calls with the Micro Focus COBOL compiler.

**Procedure:**
- When you precompile your application using the command line processor command `db2 prep`, use the `target mfcob` option.
- You must include the DB2 COBOL COPY file directory in the Micro Focus COBOL environment variable COBCPY. The COBCPY environment variable specifies the location of the COPY files. The DB2 COPY files for Micro Focus COBOL reside in `sqllib/include/cobol_mf` under the database instance directory.

  To include the directory, enter:
  - On bash or Korn shell:
    ```
    export COBCPY=$COBCPY:$HOME/sqllib/include/cobol_mf
    ```
  - On C shell:
    ```
    setenv COBCPY $COBCPY:$HOME/sqllib/include/cobol_mf
    ```

  **Note:** You might want to set COBCPY in the `.profile` or `.login` file.

**Related concepts:**
- “Considerations for installing COBOL on AIX” in *Developing SQL and External Routines*

**Related tasks:**
- “Building UNIX Micro Focus COBOL applications” on page 223
- “Building UNIX Micro Focus COBOL routines” on page 361

**Related reference:**
- “AIX Micro Focus COBOL application compile and link options” on page 234
- “AIX Micro Focus COBOL routine compile and link options” on page 355

## Configuring the Micro Focus COBOL compiler on HP-UX

If you develop applications that contain embedded SQL and DB2 API calls, and you are using the Micro Focus COBOL compiler, there are several points to keep in mind.

**Procedure:**

- When you precompile your application using the command line processor command `db2 prep`, use the `target mfcob` option.
- You must include the DB2 COBOL COPY file directory in the Micro Focus COBOL environment variable COBCPY. The COBCPY environment variable specifies the location of COPY files. The DB2 COPY files for Micro Focus COBOL reside in `sqllib/include/cobol_mf` under the database instance directory.

  To include the directory,
  - on bash or Korn shell, enter:
    ```
    export COBCPY=$COBCPY:$HOME/sqllib/include/cobol_mf
    ```
  - on C shell, enter:
    ```
    setenv COBCPY ${COBCPY}:${HOME}/sqllib/include/cobol_mf
    ```

  **Note:** You might want to set COBCPY in the `.profile` or `.login` file.

**Related tasks:**
- “Building UNIX Micro Focus COBOL applications” on page 223
- “Building UNIX Micro Focus COBOL routines” on page 361

**Related reference:**
- “HP-UX Micro Focus COBOL application compile and link options” on page 235
- “HP-UX Micro Focus COBOL routine compile and link options” on page 356

## Configuring the Micro Focus COBOL compiler on Solaris

If you develop applications that contain embedded SQL and DB2 API calls, and you are using the Micro Focus COBOL compiler, these are points you have to keep in mind.

**Procedure:**
- When you precompile your application using the command line processor command `db2 prep`, use the `target mfcob` option.
- You must include the DB2 COBOL COPY file directory in the Micro Focus COBOL environment variable COBCPY. The COBCPY environment variable specifies the location of COPY files. The DB2 COPY files for Micro Focus COBOL reside in `sqllib/include/cobol_mf` under the database instance directory.

  To include the directory, enter:
  - On bash or Korn shells:
    ```
    export COBCPY=$COBCPY:$HOME/sqllib/include/cobol_mf
    ```
  - On C shell:
    ```
    setenv COBCPY $COBCPY:$HOME/sqllib/include/cobol_mf
    ```

  **Note:** You might want to set COBCPY in the `.profile` file.

**Related tasks:**
- “Building UNIX Micro Focus COBOL applications” on page 223
- “Building UNIX Micro Focus COBOL routines” on page 361

**Related reference:**
- “Solaris Micro Focus COBOL application compile and link options” on page 236

- “Solaris Micro Focus COBOL routine compile and link options” on page 357

## AIX IBM COBOL application compile and link options

The following are the compile and link options recommended by DB2 for building COBOL embedded SQL and DB2 API applications with the IBM COBOL for AIX compiler, as demonstrated in the `bldapp` build script.

**Compile and link options for bldapp:**

Compile options:

**`cob2`** The IBM COBOL for AIX compiler.

**`-qpgmname\(mixed\)`**
Instructs the compiler to permit CALLs to library entry points with mixed-case names.

**`-qlib`** Instructs the compiler to process COPY statements.

**`-I$DB2PATH/include/cobol_a`**
Specify the location of the DB2 include files. For example: `$HOME/sqllib/include/cobol_a`.

**`-c`** Perform compile only; no link. Compile and link are separate steps.

Link options:

**`cob2`** Use the compiler as a front end for the linker.

**`-o $1`** Specify the executable program.

**`$1.o`** Specify the program object file.

**`checkerr.o`**
Include the utility object file for error-checking.

**`-L$DB2PATH/$LIB`**
Specify the location of the DB2 runtime shared libraries. For example: `$HOME/sqllib/lib32`.

**`-ldb2`** Link with the database manager library.

Refer to your compiler documentation for additional compiler options.

**Related tasks:**
- “Building IBM COBOL applications on AIX” on page 222
- “Configuring the IBM COBOL compiler on AIX” on page 231

**Related reference:**
- “AIX IBM COBOL routine compile and link options” on page 354

**Related samples:**
- “bldapp -- Builds AIX COBOL applications”

## AIX Micro Focus COBOL application compile and link options

The following are the compile and link options recommended by DB2 for building COBOL embedded SQL and DB2 API applications with the Micro Focus COBOL compiler on AIX, as demonstrated in the `bldapp` build script. Note that the DB2 MicroFocus COBOL include files are found by setting up the COBCPY environment variable, so no -I flag is needed in the compile step. Refer to the bldapp script for an example.

**Compile and link options for bldapp:**

| | |
|---|---|
| Compile options: | |
| **cob** | The MicroFocus COBOL compiler. |
| **-c** | Perform compile only; no link. |
| **$EXTRA_COBOL_FLAG="-C MFSYNC"** | Enables 64-bit support. |
| **-x** | When used with -c, produces an object file. |
| Link Options: | |
| **cob** | Use the compiler as a front end for the linker. |
| **-x** | Produces an executable program. |
| **-o $1** | Specify the executable program. |
| **$1.o** | Specify the program object file. |
| **-L$DB2PATH/$LIB** | Specify the location of the DB2 runtime shared libraries. For example: `$HOME/sqllib/lib32`. |
| **-ldb2** | Link to the DB2 library. |
| **-ldb2gmf** | Link to the DB2 exception-handler library for Micro Focus COBOL. |
| Refer to your compiler documentation for additional compiler options. | |

**Related tasks:**

- "Building UNIX Micro Focus COBOL applications" on page 223
- "Configuring the Micro Focus COBOL compiler on AIX" on page 232

**Related reference:**

- "AIX Micro Focus COBOL routine compile and link options" on page 355

**Related samples:**

- "bldapp -- Builds AIX Micro Focus COBOL applications"

## HP-UX Micro Focus COBOL application compile and link options

The following are the compile and link options recommended by DB2 for building COBOL embedded SQL and DB2 API applications with the Micro Focus COBOL compiler on HP-UX, as demonstrated in the `bldapp` build script.

**Compile and link options for bldapp:**

| | |
|---|---|
| Compile options: | |
| **cob** | The Micro Focus COBOL compiler. |
| **-cx** | Compile to object module. |
| **$EXTRA_COBOL_FLAG** | Contains ″-C MFSYNC″ if the HP-UX platform is IA64 and 64-bit support is enabled. |

Link options:

**cob** Use the compiler as a front end for the linker.

**-x** Specify an executable program.

**$1.o** Include the program object file.

**checkerr.o**
Include the utility object file for error checking.

**-L$DB2PATH/$LIB**
Specify the location of the DB2 runtime shared libraries.

**-ldb2** Link to the DB2 library.

**-ldb2gmf**
Link to the DB2 exception-handler library for Micro Focus COBOL.

Refer to your compiler documentation for additional compiler options.

**Related tasks:**

- "Building UNIX Micro Focus COBOL applications" on page 223
- "Configuring the Micro Focus COBOL compiler on HP-UX" on page 232

**Related samples:**

- "bldapp -- Builds HP-UX Micro Focus COBOL applications"

## Solaris Micro Focus COBOL application compile and link options

The following are the compile and link options recommended by DB2 for building COBOL embedded SQL and DB2 API applications with the Micro Focus COBOL compiler on Solaris, as demonstrated in the `bldapp` build script.

**Compile and link options for bldapp:**

Compile options:

**cob** The Micro Focus COBOL compiler.

**$EXTRA_COBOL_FLAG**
For 64-bit support, contains the value "-C MFSYNC"; otherwise it contains no value.

**-cx** Compile to object module.

Link options:

**cob** Use the compiler as a front end for the linker.

**-x** Specify an executable program.

**$1.o** Include the program object file.

**checkerr.o**
Include the utility object file for error-checking.

**-L$DB2PATH/$LIB**
Specify the location of the DB2 static and shared libraries at link-time. For example: `$HOME/sqllib/lib64`.

**-ldb2** Link with the DB2 library.

**-ldb2gmf**
Link with the DB2 exception-handler library for Micro Focus COBOL.

Refer to your compiler documentation for additional compiler options.

**Related tasks:**

- “Building UNIX Micro Focus COBOL applications” on page 223
- “Configuring the Micro Focus COBOL compiler on Solaris” on page 233

**Related samples:**

- “bldapp -- Builds Solaris Micro Focus COBOL applications”

## Linux Micro Focus COBOL application compile and link options

The following are the compile and link options recommended by DB2 for building COBOL embedded SQL and DB2 API applications with the Micro Focus COBOL compiler on Linux, as demonstrated in the `bldapp` build script.

**Compile and link options for bldapp:**

| | |
|---|---|
| Compile options: | |
| **`cob`** | The Micro Focus COBOL compiler. |
| **`-cx`** | Compile to object module. |
| **`$EXTRA_COBOL_FLAG`** | For 64-bit support, contains the value ″-C MFSYNC″; otherwise it contains no value. |
| Link options: | |
| **`cob`** | Use the compiler as a front end for the linker. |
| **`-x`** | Specify an executable program. |
| **`-o $1`** | Include the executable. |
| **`$1.o`** | Include the program object file. |
| **`checkerr.o`** | Include the utility object file for error checking. |
| **`-L$DB2PATH/$LIB`** | Specify the location of the DB2 runtime shared libraries. |
| **`-ldb2`** | Link to the DB2 library. |
| **`-ldb2gmf`** | Link to the DB2 exception-handler library for Micro Focus COBOL. |
| Refer to your compiler documentation for additional compiler options. | |

**Related tasks:**

- “Configuring the Micro Focus COBOL compiler on Linux” on page 230
- “Building UNIX Micro Focus COBOL applications” on page 223

**Related samples:**

- “bldapp -- Builds Linux Micro Focus COBOL applications”
- “embprep -- To prep and bind C/C++ and Micro Focus COBOL embedded SQL programs (C)”

## Windows IBM COBOL application compile and link options

The following are the compile and link options recommended by DB2 for building COBOL embedded SQL and DB2 API applications on Windows with the IBM VisualAge COBOL compiler, as demonstrated in the `bldapp.bat` batch file.

**Compile and link options for bldapp:**

Compile options:

**`cob2`** The IBM VisualAge COBOL compiler.

**`-qpgmname(mixed)`**
: Instructs the compiler to permit CALLs to library entry points with mixed-case names.

**`-c`** Perform compile only; no link. Compile and link are separate steps.

**`-qlib`** Instructs the compiler to process COPY statements.

**`-I`***path* Specify the location of the DB2 include files. For example: `-I"%DB2PATH%\include\cobol_a"`.

**`%EXTRA_COMPFLAG%`**
: If "set `IBMCOB_PRECOMP=true"` is uncommented, the IBM COBOL precompiler is used to precompile the embedded SQL. It is invoked with one of the following formulations, depending on the input parameters:

  **`-q"SQL('database sample CALL_RESOLUTION DEFERRED')"`**
  : precompile using the default sample database, and defer call resolution.

  **`-q"SQL('database %2 CALL_RESOLUTION DEFERRED')"`**
  : precompile using a database specified by the user, and defer call resolution.

  **`-q"SQL('database %2 user %3 using %4 CALL_RESOLUTION DEFERRED')"`**
  : precompile using a database, user ID, and password specified by the user, and defer call resolution. This is the format for remote client access.

Link options:

**`cob2`** Use the compiler as a front-end for the linker

**`%1.obj`** Include the program object file.

**`checkerr.obj`**
: Include the error-checking utility object file.

**`db2api.lib`**
: Link with the DB2 library.

Refer to your compiler documentation for additional compiler options.

**Related tasks:**

- "Building IBM COBOL applications on Windows" on page 225
- "Configuring the IBM COBOL compiler on Windows" on page 228

**Related samples:**

- "bldapp.bat -- Builds Windows VisualAge COBOL applications"

## Windows Micro Focus COBOL application compile and link options

The following are the compile and link options recommended by DB2 for building COBOL embedded SQL and DB2 API applications on Windows with the Micro Focus COBOL compiler, as demonstrated in the `bldapp.bat` batch file.

**Compile and link options for bldapp:**

Compile option:

**`cobol`** The Micro Focus COBOL compiler.

Link options:

**cbllink**
: Use the linker to link edit.

**-l**
: Link with the lcobol library.

**checkerr.obj**
: Link with the error-checking utility object file.

**db2api.lib**
: Link with the DB2 API library.

Refer to your compiler documentation for additional compiler options.

**Related tasks:**
- "Building Micro Focus COBOL applications on Windows" on page 226
- "Configuring the Micro Focus COBOL compiler on Windows" on page 229

**Related samples:**
- "bldapp.bat -- Builds Windows Micro Focus Cobol applications"

## Building and running embedded SQL applications written in REXX

### Building and running embedded SQL applications written in REXX

REXX applications are not precompiled, compiled, or linked. The instructions below describe how to build and run REXX applications on Windows operating systems, and on the AIX operating system.

**Restrictions:**

On Windows-based platforms, your application file must have a .CMD extension. After creation, you can run your application directly from the operating system command prompt. On AIX, your application file can have any extension.

**Procedure:**

Build and run your REXX applications as follows:
- On Windows operating systems, your application file can have any name. After creation, you can run your application from the operating system command prompt by invoking the REXX interpreter as follows:
  ```
  REXX file_name
  ```
- On AIX, you can run your application using either of the following two methods:
  - At the shell command prompt, type `rexx name` where `name` is the name of your REXX program.
  - If the first line of your REXX program contains a "magic number" (#!) and identifies the directory where the REXX/6000 interpreter resides, you can run your REXX program by typing its name at the shell command prompt. For example, if the REXX/6000 interpreter file is in the /usr/bin directory, include the following as the very first line of your REXX program:
    ```
    #! /usr/bin/rexx
    ```

Then, make the program executable by typing the following command at the shell command prompt:

```
chmod +x name
```

Run your REXX program by typing its file name at the shell command prompt.

**Note:** On AIX, you should set the LIBPATH environment variable to include the directory where the REXX SQL library, db2rexx is located. For example:

```
export LIBPATH=/lib:/usr/lib:/$DB2PATH/lib
```

**Related concepts:**

- "Embedded SQL statements in REXX applications" on page 9

**Related reference:**

- "Bind files for REXX" on page 240
- "REXX samples" on page 377

## Bind files for REXX

Five bind files are provided to support REXX applications. The names of these files are included in the DB2UBIND.LST file. Each bind file was precompiled using a different isolation level; therefore, there are five different packages stored in the database.

The five bind files are:

**DB2ARXCS.BND**
Supports the cursor stability isolation level.

**DB2ARXRR.BND**
Supports the repeatable read isolation level.

**DB2ARXUR.BND**
Supports the uncommitted read isolation level.

**DB2ARXRS.BND**
Supports the read stability isolation level.

**DB2ARXNC.BND**
Supports the no commit isolation level. This isolation level is used when working with some host, AS/400, or iSeries database servers. On other databases, it behaves like the uncommitted read isolation level.

**Note:** In some cases, it can be necessary to explicitly bind these files to the database.

When you use the SQLEXEC routine, the package created with cursor stability is used as a default. If you require one of the other isolation levels, you can change isolation levels with the SQLDBS CHANGE SQL ISOLATION LEVEL API, before connecting to the database. This will cause subsequent calls to the SQLEXEC routine to be associated with the specified isolation level.

Windows-based REXX applications cannot assume that the default isolation level is in effect unless they know that no other REXX programs in the session have changed the setting. Before connecting to a database, a REXX application should explicitly set the isolation level.

**Related concepts:**

- "Cursor types and unit of work considerations in embedded SQL applications" on page 165

- “Connecting to DB2 databases in embedded SQL applications” on page 52

### Building Object REXX applications on Windows

Object REXX is an object-oriented version of the REXX language. Object-oriented extensions have been added to classic REXX, but its existing functions and instructions have not changed. The Object REXX interpreter is an enhanced version of its predecessor, with additional support for:

- Classes, objects, and methods
- Messaging and polymorphism
- Single and multiple inheritance

Object REXX is fully compatible with classic REXX. In this section, whenever REXX is used, all versions of REXX are inferred, including Object REXX.

You do not precompile or bind REXX programs.

On Windows, REXX programs are not required to start with a comment. However, for portability reasons you are recommended to start each REXX program with a comment that begins in the first column of the first line. This will allow the program to be distinguished from a batch command on other platforms:

```
/* Any comment will do. */
```

REXX sample programs can be found in the directory `sqllib\samples\rexx`.

**Procedure:**

To run the sample REXX program `updat`, enter:

```
   rexx updat.cmd
```

**Related concepts:**

- “Restrictions on using REXX to program embedded SQL applications” on page 26

**Related reference:**

- “REXX samples” on page 377

## Building embedded SQL applications from the command line

## Building embedded SQL applications from the command line

Building embedded SQL applications from the command line involves the following steps:

1. Precompile the application by issuing the PRECOMPILE command
2. If you created a bind file, bind this file to a database to create an application package by issuing the BIND command.
3. Compile the modified application source and the source files that do not contain embedded SQL to create an application object file (a .obj file).
4. Link the application object files with the DB2 and host language libraries to create an executable program using the link command.

**Related concepts:**

- “Embedding SQL statements in a host language” on page 4

- "Binding embedded SQL packages to a database" on page 190

## Building embedded SQL applications written in C or C++ (Windows)

After you have written the source file, you have to build your embedded SQL application. Some steps in the build process depend on the compiler that you use. The examples provided with each step of the procedure show how to build an application called myapp with a Microsoft Visual Studio 6.0 compiler, which is a C compiler. You can run each step in the procedure individually or run the steps together within a batch file from a DB2 Command Window prompt. For an example of a batch file that can be used to build the embedded SQL sample applications in the %DB2PATH%\SQLLIB\samples\c\ directory, refer to the %DB2PATH%\SQLLIB\samples\c\bldapp.bat file. This batch file calls another batch file, %DB2PATH%\SQLLIB\samples\c\embprep.bat, to precompile the application and bind the application to a database.

**Prerequisites:**

- An active database connection
- An application source code file with the extension .sqc in C or .sqx in C++ and containing embedded SQL
- A supported C or C++ compiler
- The authorities or privileges required to run the PRECOMPILE command and BIND command

**Procedure:**

1. Precompile the application by issuing the PRECOMPILE command. For example:
   ```
   C application: db2 PRECOMPILE myapp.sqc BINDFILE
   C++ application: db2 PRECOMPILE myapp.sqx BINDFILE
   ```
   The PRECOMPILE command generates a .c or .C file, that contains a modified form of the source code in a .sqc or .sqC file, and an application package. If you use the BINDFILE option, the PRECOMPILE command generates a bind file. In the example above, the bind file would be called myapp.bnd.
2. If you created a bind file, bind this file to a database to create an application package by issuing the BIND command. For example:
   ```
   db2 bind myapp.bnd
   ```
   The BIND command associates the application package with and stores the package within the database.
3. Compile the modified application source and the source files that do not contain embedded SQL to create an application object file (a .obj file). For example:
   ```
   C application: cl -Zi -Od -c -W2 -DWIN32 myapp.c
   C++ application: cl -Zi -Od -c -W2 -DWIN32 myapp.cxx
   ```
4. Link the application object files with the DB2 and host language libraries to create an executable program using the link command. For example:
   ```
   link -debug -out:myapp.exe myapp.obj
   ```

**Next Task:**

You can do any of the following tasks next:

- Debugging embedded SQL application compile time errors

- Tuning the performance of embedded SQL applications
- Deploying the executable application, myapp.exe
- Running embedded SQL applications

**Related concepts:**
- "Connecting to DB2 databases in embedded SQL applications" on page 52
- "Performance of embedded SQL applications" on page 22
- "Precompiler generated timestamps" on page 188

**Related reference:**
- "Windows C and C++ application compile and link options" on page 220

## Migrating embedded SQL applications

To ensure that your embedded SQL applications built for DB2 UDB Version 8 will work with DB2 Version 9, you need to migrate these applications.

**Prerequisites:**

Before migrating your embedded SQL applications, ensure the following:
- The DB2 client or server from where your application is running needs to be migrated to DB2 Version 9.
- Ensure that the development software you use for your embedded SQL applications is supported for DB2 Version 9.

**Procedure:**

To migrate your embedded SQL application to work with DB2 Version 9, complete the following steps:
1. Test your embedded SQL applications in a DB2 Version 9 testing environment. If testing is successful, you do not need to perform any additional steps.
2. Determine if there are any SQL statements in your embedded SQL applications that reference catalog views, SQL administrative routines, or SQL administrative views. If you do have SQL statements that reference these objects, test them from the DB2 Command Line Processor to ensure the view or routine still has the same signature. If necessary, see the reference documentation on catalog views or SQL administrative routines and views for updated signatures.
3. Rebuild any changed embedded SQL applications, using the appropriate DB2 build file, specifying the appropriate DB2 shared library path. For 32-bit applications, specify `$INSTHOME/sqllib/lib32` as the shared library path, and for 64-bit applications, specify `$INSTHOME/sqllib/lib64` as the shared library path, where `INSTHOME` is the instance home directory.
4. For embedded SQL applications on UNIX and Linux, run the **db2chglibpath** command to ensure that any libraries that have changed paths between DB2 version are found.
5. Optional: Explicitly validate packages associated with your embedded SQL applications: Rebinding packages in migrated databases.

   During database migration, the existing packages for your embedded SQL applications are invalidated. After the migration process, each package is automatically rebuilt when it is used for the first time by the DB2 database manager.

6. Test your database applications to ensure that they run as expected under DB2 Version 9.

**Related concepts:**

- "Administrative SQL routines and views" in *Administrative SQL Routines and Views*

**Related tasks:**

- "Rebinding packages in migrated databases" in *Migration Guide*
- "Migrating 32-bit database applications to run on 64-bit instances" in *Migration Guide*
- "Migrating database applications" in *Migration Guide*

**Related reference:**

- "Supported development software for embedded SQL applications" on page 11
- "System catalog views" in *SQL Reference, Volume 1*

## Restrictions on linking to libdb2.so

On some Linux distributions, the libc development rpm comes with the

```
/usr/lib/libdb2.so
```

or

```
/usr/lib64/libdb2.so
```

library. This library is used for Sleepycat Software's Berkeley DB implementation and is not associated with IBM DB2 database systems.

If you do not plan to use Berkeley DB, you can rename or delete these library files permanently on your systems.

If you do want to use Berkeley DB, you can rename the folder containing these library files and modify the environment variable to point to the new folder.

**Related concepts:**

- "Building embedded SQL applications" on page 180

# Part 2. Developing embedded routines

# Chapter 5. Developing external routines

## External routines

External routines are routines that have their logic implemented in a programming language application that resides outside of the database, in the file system of the database server. The association of the routine with the external code application is asserted by the specification of the EXTERNAL clause in the CREATE statement of the routine.

You can create external procedures, external functions, and external methods. Although they are all implemented in external programming languages, each routine functional type has different features. Before deciding to implement an external routine, it is important that you first understand what external routines are, and how they are implemented and used, by reading the topic, ″Overview of external routines″. With that knowledge you can then learn more about external routines from the topics targeted by the related links so that you can make informed decisions about when and how to use them in your database environment.

**Related concepts:**
- "OLE DB user-defined table functions" in *Developing SQL and External Routines*
- "Overview of external routines" on page 248
- "Overview of routines" in *Developing SQL and External Routines*
- "Support for external routine development in .NET CLR languages" in *Developing SQL and External Routines*
- "Support for external routine development in C" on page 285
- "Support for external routine development in C++" on page 285
- "Supported Java routine development software" in *Developing SQL and External Routines*
- ".NET common language runtime (CLR) routines" in *Developing SQL and External Routines*
- "COBOL procedures" on page 351
- "DB2GENERAL routines" in *Developing SQL and External Routines*
- "External routine features" on page 248
- "Java routines" in *Developing SQL and External Routines*
- "OLE automation routine design" in *Developing SQL and External Routines*

**Related tasks:**
- "Creating C and C++ routines" on page 326
- "Creating external routines" on page 282
- "Developing routines" in *Developing SQL and External Routines*

**Related reference:**
- "Support for external procedure development in COBOL" on page 354
- "CREATE FUNCTION (External Table) statement" in *SQL Reference, Volume 2*
- "CREATE PROCEDURE (External) statement" in *SQL Reference, Volume 2*

## Overview of external routines

### Overview of external routines

External routines are characterized primarily by the fact that their routine logic is implemented in programming language code and not in SQL.

Before deciding to implement an external routine, it is important that you understand what external routines are, how they are implemented, and how they can be used. The following concept topics will help you get an understanding of external routines so that you can make informed decisions about when and how to use them in your database environment:

- "External routine features"
- External routine creation
- External routine library or class management
- Supported programming languages for external routine development
- 32-bit and 64-bit support for external routines
- External routine parameter styles
- Restrictions on external routines

Once you have an understanding of external routine concepts you might want to:

- "Creating external routines" on page 282

**Related concepts:**

- "32-bit and 64-bit support for external routines" on page 276
- "External function and method features" on page 249
- "External routine creation" on page 268
- "External routine features" on page 248
- "External routine library and class management" on page 272
- "External routine parameter styles" on page 269
- "External routines" on page 247
- "Performance of routines with 32-bit libraries on 64-bit database servers" on page 277
- "Restrictions on external routines" on page 279
- "Supported APIs and programming languages for external routine development" on page 260
- "XML data type support in external routines" on page 278

**Related tasks:**

- "Creating external routines" on page 282

### External routine features

External routines provide support for most of the common routine features as well as support for additional features not supported by SQL routines. The following features are unique to external routines:

**Access to files, data, and applications residing outside of the database**

External routines can access and manipulate data or files that reside outside of the database itself. They can also invoke applications that reside outside of the database. The data, files, or applications might, for example, reside in the database server file system or within the available network.

**Variety of external routine parameter style options**
The implementation of external routines in a programming language can be done using a choice of parameter styles. Although there might be a preferred parameter style for a chosen programming language, there is sometimes choice. Some parameter styles provide support for the passing of additional database and routine property information to and from the routine in a structure named *dbinfo* structure that might be useful within the routine logic.

**Preservation of state between external function invocations with a scratchpad**
External user-defined functions provide support for state preservation between function invocations for a set of values. This is done with a structure called a *scratchpad*. This can be useful both for functions that return aggregated values and for functions that require initial setup logic such as initialization of buffers.

**Call-types identify individual external function invocations**
External user-defined functions are invoked multiple times for a set of values. Each invocation is identified with a call-type value that can be referenced within the function logic. For example there are special call-types for the first invocation of a function, for data fetching calls, and for the final invocation. Call-types are useful, because specific logic can be associated with a particular call-type.

**Related concepts:**
- "32-bit and 64-bit support for external routines" on page 276
- "External function and method features" on page 249
- "External routines" on page 247
- "Features of SQL procedures" in *Developing SQL and External Routines*
- "Overview of external routines" on page 248
- "Restrictions on external routines" on page 279
- "SQL in external routines" in *Developing SQL and External Routines*
- "XML data type support in external routines" on page 278

## External function and method features

### External function and method features

External functions and external methods provide support for functions that, for a given set of input data, might be invoked multiple times and produce a set of output values.

To learn more about the features of external functions and methods, see the following topics:
- "External scalar functions" on page 250
- "External scalar function and method processing model" on page 252
- "External table functions" on page 252
- "External table function processing model" on page 253
- "Table function execution model for Java" on page 255
- "Scratchpads for external functions and methods" on page 256
- "Scratchpads on 32-bit and 64-bit operating systems" on page 259

These features are unique to external functions and methods and do not apply to SQL functions and SQL methods.

**Related concepts:**
- "Restrictions on external routines" on page 279
- "Overview of external routines" on page 248
- "External scalar functions" on page 250
- "External routine features" on page 248
- "External routines" on page 247
- "External scalar function and method processing model" on page 252

## External scalar functions

External scalar functions are scalar functions that have their logic implemented in an external programming language.

These functions can be developed and used to extend the set of existing SQL functions and can be invoked in the same manner as DB2 built-in functions such as LENGTH and COUNT. That is, they can be referenced in SQL statements wherever an expression is valid.

The execution of external scalar function logic takes place on the DB2 database server, however unlike built-in or user-defined SQL scalar functions, the logic of external functions can access the database server filesystem, perform system calls or access a network.

External scalar functions can read SQL data, but cannot modify SQL data.

External scalar functions can be repeatedly invoked for a single reference of the function and can maintain state between these invocations by using a scratchpad, which is a memory buffer. This can be powerful if a function requires some initial, but expensive, setup logic. The setup logic can be done on a first invocation using the scratchpad to store some values that can be accessed or updated in subsequent invocations of the scalar function.

**Features of external scalar functions**
- Can be referenced as part of an SQL statement anywhere an expression is supported.
- The output of a scalar function can be used directly by the invoking SQL statement.
- For external scalar user-defined functions, state can be maintained between the iterative invocations of the function by using a scratchpad.
- Can provide a performance advantage when used in predicates, because they are executed at the server. If a function can be applied to a candidate row at the server, it can often eliminate the row from consideration before transmitting it to the client machine, reducing the amount of data that must be passed from server to client.

**Limitations**
- Cannot do transaction management within a scalar function. That is, you cannot issue a COMMIT or a ROLLBACK within a scalar function.
- Cannot return result sets.
- Scalar functions are intended to return a single scalar value per set of inputs.

- External scalar functions are not intended to be used for a single invocation. They are designed such that for a single reference to the function and a given set of inputs, that the function be invoked once per input, and return a single scalar value. On the first invocation, scalar functions can be designed to do some setup work, or store some information that can be accessed in subsequent invocations. SQL scalar functions are better suited to functionality that requires a single invocation.
- In a single partition database external scalar functions can contain SQL statements. These statements can read data from tables, but cannot modify data in tables. If the database has more than one partition then there must be no SQL statements in an external scalar function. SQL scalar functions can contain SQL statements that read or modify data.

**Common uses**

- Extend the set of DB2 built-in functions.
- Perform logic inside an SQL statement that SQL cannot natively perform.
- Encapsulate a scalar query that is commonly reused as a subquery in SQL statements. For example, given a postal code, search a table for the city where the postal code is found.

**Supported languages**

- C
- C++
- Java
- OLE
- .NET common language runtime languages

**Notes:**

1. There is a limited capability for creating aggregate functions. Also known as column functions, these functions receive a set of like values (a column of data) and return a single answer. A user-defined aggregate function can only be created if it is sourced upon a built-in aggregate function. For example, if a distinct type `SHOESIZE` exists that is defined with base type INTEGER, you could define a function, `AVG(SHOESIZE)`, as an aggregate function sourced on the existing built-in aggregate function, `AVG(INTEGER)`.
2. You can also create function that return a row. These are known as row functions and can only be used as a transform function for structured types. The output of a row function is a single row.

**Related concepts:**

- "Benefits of using routines" in *Developing SQL and External Routines*
- "External function and method features" on page 249
- "External scalar function and method processing model" on page 252
- "OLE DB user-defined table functions" in *Developing SQL and External Routines*
- "Scratchpads for external functions and methods" on page 256
- "External table functions" on page 252

**Related tasks:**

- "Invoking scalar functions or methods" in *Developing SQL and External Routines*
- "Invoking user-defined table functions" in *Developing SQL and External Routines*

**Related reference:**
- "CREATE FUNCTION statement" in *SQL Reference, Volume 2*

## External scalar function and method processing model

The processing model for methods and scalar UDFs that are defined with the FINAL CALL specification is as follows:

**FIRST call**
: This is a special case of the NORMAL call, identified as FIRST to enable the function to perform any initial processing. Arguments are evaluated and passed to the function. Normally, the function will return a value on this call, but it can return an error, in which case no NORMAL or FINAL call is made. If an error is returned on a FIRST call, the method or UDF must clean up before returning, because no FINAL call will be made.

**NORMAL call**
: These are the second through second-last calls to the function, as dictated by the data and the logic of the statement. The function is expected to return a value with each NORMAL call after arguments are evaluated and passed. If NORMAL call returns an error, no further NORMAL calls are made, but the FINAL call is made.

**FINAL call**
: This is a special call, made at end-of-statement processing (or CLOSE of a cursor), provided that the FIRST call succeeded. No argument values are passed on a FINAL call. This call is made so that the function can clean up any resources. The function does not return a value on this call, but can return an error.

For methods or scalar UDFs not defined with FINAL CALL, only NORMAL calls are made to the function, which normally returns a value for each call. If a NORMAL call returns an error, or if the statement encounters another error, no more calls are made to the function.

**Note:** This model describes the ordinary error processing for methods and scalar UDFs. In the event of a system failure or communication problem, a call indicated by the error processing model cannot be made. For example, for a FENCED UDF, if the `db2udf` fenced process is somehow prematurely terminated, DB2 cannot make the indicated calls.

**Related concepts:**
- "External function and method features" on page 249
- "External scalar functions" on page 250

## External table functions

A user-defined table function delivers a table to the SQL in which it is referenced. A table UDF reference is only valid in a FROM clause of a SELECT statement. When using table functions, observe the following:
- Even though a table function delivers a table, the physical interface between DB2 and the UDF is one-row-at-a-time. There are five types of calls made to a table function: OPEN, FETCH, CLOSE, FIRST, and FINAL. The existence of FIRST and FINAL calls depends on how you define the UDF. The same *call-type* mechanism that can be used for scalar functions is used to distinguish these calls.

- Not every result column defined in the RETURNS clause of the CREATE FUNCTION statement for the table function has to be returned. The DBINFO keyword of CREATE FUNCTION, and corresponding *dbinfo* argument enable the optimization that only those columns needed for a particular table function reference need be returned.
- The individual column values returned conform in format to the values returned by scalar functions.
- The CREATE FUNCTION statement for a table function has a CARDINALITY specification. This specification enables the definer to inform the DB2 optimizer of the approximate size of the result so that the optimizer can make better decisions when the function is referenced.

  Regardless of what has been specified as the CARDINALITY of a table function, exercise caution against writing a function with infinite cardinality, that is, a function that always returns a row on a FETCH call. There are many situations where DB2 expects the end-of-table condition, as a catalyst within its query processing. Using GROUP BY or ORDER BY are examples where this is the case. DB2 cannot form the groups for aggregation until end-of-table is reached, and it cannot sort until it has all the data. So a table function that never returns the end-of-table condition (SQL-state value ’02000’) can cause an infinite processing loop if you use it with a GROUP BY or ORDER BY clause.

**Related concepts:**
- “External function and method features” on page 249
- “External table function processing model” on page 253

**Related reference:**
- “CREATE FUNCTION statement” in *SQL Reference, Volume 2*
- “Passing arguments to C, C++, OLE, or COBOL routines” on page 311

## External table function processing model

The processing model for table UDFs that are defined with the FINAL CALL specification is as follows:

**FIRST call**
: This call is made before the first OPEN call, and its purpose is to enable the function to perform any initial processing. The scratchpad is cleared prior to this call. Arguments are evaluated and passed to the function. The function does not return a row. If the function returns an error, no further calls are made to the function.

**OPEN call**
: This call is made to enable the function to perform special OPEN processing specific to the scan. The scratchpad (if present) is not cleared prior to the call. Arguments are evaluated and passed. The function does not return a row on an OPEN call. If the function returns an error from the OPEN call, no FETCH or CLOSE call is made, but the FINAL call will still be made at end of statement.

**FETCH call**
: FETCH calls continue to be made until the function returns the SQLSTATE value signifying end-of-table. It is on these calls that the UDF develops and returns a row of data. Argument values can be passed to the function, but they are pointing to the same values that were passed on OPEN. Therefore, the argument values might not be current and should not be relied upon. If you do need to maintain current values between the invocations of a

table function, use a scratchpad. The function can return an error on a FETCH call, and the CLOSE call will still be made.

**CLOSE call**
: This call is made at the conclusion of the scan or statement, provided that the OPEN call succeeded. Any argument values will not be current. The function can return an error.

**FINAL call**
: The FINAL call is made at the end of the statement, provided that the FIRST call succeeded. This call is made so that the function can clean up any resources. The function does not return a value on this call, but can return an error.

For table UDFs not defined with FINAL CALL, only OPEN, FETCH, and CLOSE calls are made to the function. Before each OPEN call, the scratchpad (if present) is cleared.

The difference between table UDFs that are defined with FINAL CALL and those defined with NO FINAL CALL can be seen when examining a scenario involving a join or a subquery, where the table function access is the "inner" access. For example, in a statement such as:

```
SELECT x,y,z,... FROM table_1 as A,
  TABLE(table_func_1(A.col1,...)) as B
  WHERE...
```

In this case, the optimizer would open a scan of table_func_1 for each row of table_1. This is because the value of table_1's col1, which is passed to table_func_1, is used to define the table function scan.

For NO FINAL CALL table UDFs, the OPEN, FETCH, FETCH, ..., CLOSE sequence of calls repeats for each row of table_1. Note that each OPEN call will get a clean scratchpad. Because the table function does not know at the end of each scan whether there will be more scans, it must clean up completely during CLOSE processing. This could be inefficient if there is significant one-time open processing that must be repeated.

FINAL CALL table UDFs, provide a one-time FIRST call, and a one-time FINAL call. These calls are used to amortize the expense of the initialization and termination costs across all the scans of the table function. As before, the OPEN, FETCH, FETCH, ..., CLOSE calls are made for each row of the outer table, but because the table function knows it will get a FINAL call, it does not need to clean everything up on its CLOSE call (and reallocate on subsequent OPEN). Also note that the scratchpad is not cleared between scans, largely because the table function resources will span scans.

At the expense of managing two additional call types, the table UDF can achieve greater efficiency in these join and subquery scenarios. Deciding whether to define the table function as FINAL CALL depends on how it is expected to be used.

**Related concepts:**

- "External function and method features" on page 249
- "Table function execution model for Java" on page 255
- "User-defined table functions" in *Developing SQL and External Routines*

**Related reference:**

- "CREATE FUNCTION (External Table) statement" in *SQL Reference, Volume 2*
- "CREATE FUNCTION (OLE DB External Table) statement" in *SQL Reference, Volume 2*
- "CREATE FUNCTION (SQL Scalar, Table, or Row) statement" in *SQL Reference, Volume 2*

## Table function execution model for Java

For table functions written in Java and using PARAMETER STYLE DB2GENERAL, it is important to understand what happens at each point in DB2's processing of a given statement. The following table details this information for a typical table function. Covered are both the NO FINAL CALL and the FINAL CALL cases, assuming SCRATCHPAD in both cases.

| **Point in scan time** | **NO FINAL CALL LANGUAGE JAVA SCRATCHPAD** | **FINAL CALL LANGUAGE JAVA SCRATCHPAD** |
|---|---|---|
| Before the first OPEN for the table function | No calls. | • Class constructor is called (means new scratchpad). UDF method is called with FIRST call.<br>• Constructor initializes class and scratchpad variables. Method connects to Web server. |
| At each OPEN of the table function | • Class constructor is called (means new scratchpad). UDF method is called with OPEN call.<br>• Constructor initializes class and scratchpad variables. Method connect to Web server, and opens the scan for Web data. | • UDF method is opened with OPEN call.<br>• Method opens the scan for whatever Web data it wants. (Might be able to avoid reopen after a CLOSE reposition, depending on what is saved in the scratchpad.) |
| At each FETCH for a new row of table function data | • UDF method is called with FETCH call.<br>• Method fetches and returns next row of data, or EOT. | • UDF method is called with FETCH call.<br>• Method fetches and returns new row of data, or EOT. |
| At each CLOSE of the table function | • UDF method is called with CLOSE call. `close()` method if it exists for class.<br>• Method closes its Web scan and disconnects from the Web server. `close()` does not need to do anything. | • UDF method is called with CLOSE call.<br>• Method might reposition to the top of the scan, or close the scan. It can save any state in the scratchpad, which will persist. |
| After the last CLOSE of the table function | No calls. | • UDF method is called with FINAL call. `close()` method is called if it exists for class.<br>• Method disconnects from the Web server. `close()` method does not need to do anything. |

**Notes:**

1. The term "UDF method" refers to the Java class method that implements the UDF. This is the method identified in the EXTERNAL NAME clause of the CREATE FUNCTION statement.

2. For table functions with NO SCRATCHPAD specified, the calls to the UDF method are as indicated in this table, but because the user is not asking for any continuity with a scratchpad, DB2 will cause a new object to be instantiated before each call, by calling the class constructor. It is not clear that table functions with NO SCRATCHPAD (and thus no continuity) can do useful things, but they are supported.

**Related concepts:**
- "DB2GENERAL routines" in *Developing SQL and External Routines*
- "External function and method features" on page 249
- "External table function processing model" on page 253
- "Java routines" in *Developing SQL and External Routines*

**Related tasks:**
- "Designing Java routines" in *Developing SQL and External Routines*

**Related reference:**
- "CREATE FUNCTION (External Table) statement" in *SQL Reference, Volume 2*

## Scratchpads for external functions and methods

A *scratchpad* enables a user-defined function or method to save its state from one invocation to the next. For example, here are two situations where saving state between invocations is beneficial:

1. Functions or methods that, to be correct, depend on saving state.

   An example of such a function or method is a simple counter function that returns a '1' the first time it is called, and increments the result by one each successive call. Such a function could, in some circumstances, be used to number the rows of a SELECT result:

   ```
   SELECT counter(), a, b+c, ...
     FROM tablex
     WHERE ...
   ```

   The function needs a place to store the current value for the counter between invocations, where the value will be guaranteed to be the same for the following invocation. On each invocation, the value can then be incremented and returned as the result of the function.

   This type of routine is NOT DETERMINISTIC. Its output does not depend solely on the values of its SQL arguments.

2. Functions or methods where the performance can be improved by the ability to perform some initialization actions.

   An example of such a function or method, which might be a part of a document application, is a *match* function, which returns 'Y' if a given document contains a given string, and 'N' otherwise:

   ```
   SELECT docid, doctitle, docauthor
     FROM docs
     WHERE match('myocardial infarction', docid) = 'Y'
   ```

   This statement returns all the documents containing the particular text string value represented by the first argument. What *match* would like to do is:

   - First time only.

     Retrieve a list of all the document IDs that contain the string 'myocardial infarction' from the document application, that is maintained outside of DB2. This retrieval is a costly process, so the function would like to do it only one time, and save the list somewhere handy for subsequent calls.

- On each call.

  Use the list of document IDs saved during the first call to see if the document ID that is passed as the second argument is contained in the list.

  This type of routine is DETERMINISTIC. Its answer only depends on its input argument values. What is shown here is a function whose performance, not correctness, depends on the ability to save information from one call to the next.

Both of these needs are met by the ability to specify a SCRATCHPAD in the CREATE statement:

```
CREATE FUNCTION counter()
  RETURNS int ... SCRATCHPAD;

CREATE FUNCTION match(varchar(200), char(15))
  RETURNS char(1) ... SCRATCHPAD 10000;
```

The SCRATCHPAD keyword tells DB2 to allocate and maintain a scratchpad for a routine. The default size for a scratchpad is 100 bytes, but you can determine the size (in bytes) for a scratchpad. The *match* example is 10000 bytes long. DB2 initializes the scratchpad to binary zeros before the first invocation. If the scratchpad is being defined for a table function, and if the table function is also defined with NO FINAL CALL (the default), DB2 refreshes the scratchpad before each OPEN call. If you specify the table function option FINAL CALL, DB2 does not examine or change the content of the scratchpad after its initialization. For scalar functions defined with scratchpads, DB2 also does not examine or change the scratchpad's content after its initialization. A pointer to the scratchpad is passed to the routine on each invocation, and DB2 preserves the routine's state information in the scratchpad.

So for the *counter* example, the last value returned could be kept in the scratchpad. And the *match* example could keep the list of documents in the scratchpad if the scratchpad is big enough, otherwise it could allocate memory for the list and keep the address of the acquired memory in the scratchpad. Scratchpads can be variable length: the length is defined in the CREATE statement for the routine.

The scratchpad only applies to the individual reference to the routine in the statement. If there are multiple references to a routine in a statement, each reference has its own scratchpad, thus scratchpads cannot be used to communicate between references. The scratchpad only applies to a single DB2 agent (an agent is a DB2 entity that performs processing of all aspects of a statement). There is no ″global scratchpad″ to coordinate the sharing of scratchpad information between the agents. This is especially important for situations where DB2 establishes multiple agents to process a statement (in either a single partition or multiple partition database). In these cases, even though there might only be a single reference to a routine in a statement, there could be multiple agents doing the work, and each would have its own scratchpad. In a multiple partition database, where a statement referencing a UDF is processing data on multiple partitions, and invoking the UDF on each partition, the scratchpad would only apply to a single partition. As a result, there is a scratchpad on each partition where the UDF is executed.

If the correct execution of a function depends on there being a single scratchpad per reference to the function, then register the function as DISALLOW PARALLEL. This will force the function to run on a single partition, thereby guaranteeing that only a single scratchpad will exist per reference to the function.

Because it is recognized that a UDF or method might require system resources, the UDF or method can be defined with the FINAL CALL keyword. This keyword tells DB2 to call the UDF or method at end-of-statement processing so that the UDF or method can release its system resources. It is vital that a routine free any resources it acquires; even a small leak can become a big leak in an environment where the statement is repetitively invoked, and a big leak can cause a DB2 crash.

Since the scratchpad is of a fixed size, the UDF or method can itself include a memory allocation and thus, can make use of the final call to free the memory. For example, the preceding *match* function cannot predict how many documents will match the given text string. So a better definition for *match* is:

```
CREATE FUNCTION match(varchar(200), char(15))
  RETURNS char(1) ... SCRATCHPAD 10000 FINAL CALL;
```

For UDFs or methods that use a scratchpad and are referenced in a subquery, DB2 might make a final call, if the UDF or method is so specified, and refresh the scratchpad between invocations of the subquery. You can protect yourself against this possibility, if your UDFs or methods are ever used in subqueries, by defining the UDF or method with FINAL CALL and using the call-type argument, or by always checking for the *binary zero* state of the scratchpad.

If you do specify FINAL CALL, note that your UDF or method receives a call of type FIRST. This could be used to acquire and initialize some persistent resource.

Following is a simple Java example of a UDF that uses a scratchpad to compute the sum of squares of entries in a column. This example takes in a column and returns a column containing the cumulative sum of squares from the top of the column to the current row entry:

```
CREATE FUNCTION SumOfSquares(INTEGER)
RETURNS INTEGER
EXTERNAL NAME 'UDFsrv!SumOfSquares'
DETERMINISTIC
NO EXTERNAL ACTION
FENCED
NOT NULL CALL
LANGUAGE JAVA
PARAMETER STYLE DB2GENERAL
NO SQL
SCRATCHPAD 10
FINAL CALL
DISALLOW PARALLEL
NO DBINFO@


// Sum Of Squares using Scratchpad UDF
public void SumOfSquares(int inColumn,
                         int outSum)
throws Exception
{
  int sum = 0;
  byte[] scratchpad = getScratchpad();

  // variables to read from SCRATCHPAD area
  ByteArrayInputStream byteArrayIn = new ByteArrayInputStream(scratchpad);
  DataInputStream dataIn = new DataInputStream(byteArrayIn);

  // variables to write into SCRATCHPAD area
  byte[] byteArrayCounter;
  int i;
  ByteArrayOutputStream byteArrayOut = new ByteArrayOutputStream(10);
  DataOutputStream dataOut = new DataOutputStream(byteArrayOut);
```

```
    switch(getCallType())
    {
      case SQLUDF_FIRST_CALL:
        // initialize data
     sum = (inColumn * inColumn);
        // save data into SCRATCHPAD area
        dataOut.writeInt(sum);
        byteArrayCounter = byteArrayOut.toByteArray();
        for(i = 0; i < byteArrayCounter.length; i++)
        {
          scratchpad[i] = byteArrayCounter[i];
        }
        setScratchpad(scratchpad);
     break;
      case SQLUDF_NORMAL_CALL:
        // read data from SCRATCHPAD area
        sum = dataIn.readInt();
        // work with data
        sum = sum + (inColumn * inColumn);
        // save data into SCRATCHPAD area
        dataOut.writeInt(sum);
        byteArrayCounter = byteArrayOut.toByteArray();
        for(i = 0; i < byteArrayCounter.length; i++)
        {
          scratchpad[i] = byteArrayCounter[i];
        }
        setScratchpad(scratchpad);
  break;
    }
    //set the output value
    set(2, sum);
  } // SumOfSquares UDF
```

Please note that there is a built-in DB2 function that performs the same task as the SumOfSquares UDF. This example was chosen to demonstrate the use of a scratchpad.

**Related concepts:**
- "Scratchpad as C or C++ function parameter" on page 298
- "Scratchpads on 32-bit and 64-bit operating systems" on page 259
- "External function and method features" on page 249
- "External scalar function and method processing model" on page 252

**Related reference:**
- "CREATE FUNCTION statement" in *SQL Reference, Volume 2*
- "CREATE METHOD statement" in *SQL Reference, Volume 2*
- "CREATE TYPE (Structured) statement" in *SQL Reference, Volume 2*

**Related samples:**
- "udfsrv.c -- Library of user-defined functions (UDFs) called by the client"
- "UDFCreate.db2 -- How to catalog the Java UDFs contained in UDFsrv.java "
- "UDFsrv.java -- Provide UDFs to be called by UDFcli.java (JDBC)"

## Scratchpads on 32-bit and 64-bit operating systems

To make your UDF or method code portable between 32-bit and 64-bit operating systems, you must take care in the way you create and use scratchpads that contain 64-bit values. It is recommended that you do not declare an explicit length variable for a scratchpad structure that contains one or more 64-bit values, such as 64-bit pointers or `sqlint64` BIGINT variables.

Following is a sample structure declaration for a scratchpad:

```
struct sql_scratchpad
{
   sqlint32 length;
   char data[100];
};
```

When defining its own structure for the scratchpad, a routine has two choices:

1. Redefine the entire scratchpad `sql_scratchpad`, in which case it needs to include an explicit length field. For example:
   ```
   struct sql_spad
   {
     sqlint32 length;
     sqlint32 int_var;
     sqlint64 bigint_var;
   };
   void SQL_API_FN routine( ..., struct sql_spad* scratchpad, ... )
   {
     /* Use scratchpad */
   }
   ```
2. Redefine just the data portion of the scratchpad `sql_scratchpad`, in which case no length field is needed.
   ```
   struct spaddata
   {
     sqlint32 int_var;
     sqlint64 bigint_var;
   };
   void SQL_API_FN routine( ..., struct sql_scratchpad* spad, ... )
   {
     struct spaddata* scratchpad = (struct spaddata*)spad→data;
     /* Use scratchpad */
   }
   ```

Since the application cannot change the value in the length field of the scratchpad, there is no significant benefit to coding the routine as shown in the first example. The second example is also portable between computers with different word sizes, so it is the preferred way of writing the routine.

**Related concepts:**

- "External function and method features" on page 249
- "Scratchpads for external functions and methods" on page 256
- "External scalar functions" on page 250
- "User-defined table functions" in *Developing SQL and External Routines*

**Related tasks:**

- "Invoking 32-bit routines on a 64-bit database server" in *Developing SQL and External Routines*

## Supported APIs and programming languages

### Supported APIs and programming languages for external routine development

You can develop DB2 external routines (procedures and functions) using the following APIs and associated programming languages:

- ADO.NET
  - .NET Common Language Runtime programming languages
- CLI
- Embedded SQL
  - C
  - C++
  - COBOL (Only supported for procedures)
- JDBC
  - Java
- OLE
  - Visual Basic
  - Visual C++
  - Any other programming language that supports this API.
- OLE DB (Only supported for table functions)
  - Any programming language that supports this API.
- SQLJ
  - Java

**Related concepts:**
- "Overview of external routines" on page 248
- "Support for external routine development in .NET CLR languages" in *Developing SQL and External Routines*
- "Support for external routine development in C" on page 285
- "Support for external routine development in C++" on page 285
- "Supported Java routine development software" in *Developing SQL and External Routines*
- "Choosing an application programming interface" in *Getting Started with Database Application Development*
- "Supported database application programming interfaces" in *Getting Started with Database Application Development*

**Related reference:**
- "Support for external procedure development in COBOL" on page 354
- "Supported programming languages and compilers for database application development" in *Getting Started with Database Application Development*

## Comparison of supported APIs and programming languages for external routine development

It is important to consider the characteristics and limitations of the various supported external routine application programming interfaces (APIs) and programming languages before you start implementing external routines. This will ensure that you choose the right implementation from the start and that the routine features that you require are available.

*Table 25. Comparison of external routine APIs and programming languages*

| API and programming language | Feature support | Performance | Security | Scalability | Limitations |
|---|---|---|---|---|---|
| SQL (includes SQL PL) | • SQL is a high level language that is easy to learn and use, which makes implementation go quickly.<br>• SQL Procedural Language (SQL PL) elements allow for control-flow logic around SQL operations and queries.<br>• Strong data type support. | • Very good.<br>• SQL routines perform better than Java routines.<br>• SQL routines perform as well as C and C++ external routines created with the NOT FENCED clause. | • Very safe.<br>• SQL procedures run in the same memory as the database manager. | • Highly scalable. | • Cannot access the database server file system.<br>• Cannot invoke applications that reside outside of the database. |

*Table 25. Comparison of external routine APIs and programming languages (continued)*

| API and programming language | Feature support | Performance | Security | Scalability | Limitations |
|---|---|---|---|---|---|
| Embedded SQL (includes C and C++) | • Low level, but powerful programming language. | • Very good.<br>• C and C++ routines perform better than Java routines.<br>• C and C++ routines created with the NOT FENCED clause perform as well as SQL routines. | • C and C++ routines are prone to programming errors.<br>• Programmers must be proficient in C to avoid making common memory and pointer manipulation errors which make routine implementation more tedious and time consuming.<br>• C and C++ routines should be created with the FENCED clause and the NOT THREADSAFE clause to avoid the disruption of the database manager should an exception occur in the routine at run time. These are default clauses. The use of these clauses can somewhat negatively impact performance, but ensure safe execution. See: Security of routines . | • Scalability is reduced when C and C++ routines are created with the FENCED and NOT THREADSAFE clauses. These routines are run in an isolated *db2fmp* process apart from the database manager process. One *db2fmp* process is required per concurrently executed routine. | • There are multiple supported parameter passing styles which can be confusing. Users should use parameter style SQL as much as possible. |

*Table 25. Comparison of external routine APIs and programming languages (continued)*

| API and programming language | Feature support | Performance | Security | Scalability | Limitations |
|---|---|---|---|---|---|
| Embedded SQL (COBOL) | • High-level programming language good for developing business, typically file oriented, applications.<br>• Pervasively used in the past for production business applications, although its popularity is decreasing.<br>• COBOL does not contain pointer support and is a linear iterative programming language. | • COBOL routines do not perform as well as routines created with any of the other external routine implementation options. | • No information at this time. | • No information at this time. | • You can create and invoke 32-bit COBOL procedures in 64-bit DB2 instances, however these routines will not perform as well as 64-bit COBOL procedures within a 64-bit DB2 instance. |
| JDBC (Java) and SQLJ (Java) | • High-level object-oriented programming language suitable for developing standalone applications, applets, and servlets.<br>• Java objects and data types facilitate the establishment of database connections, execution of SQL statements, and manipulation of data. | • Java routines do not perform as well as C and C++ routines or SQL routines. | • Java routines are safer than C and C++ routines, because the control of dangerous operations is handled by the Java Virtual Machine (JVM). This increases reliability and makes it very difficult for the code of one Java routine to harm another routine running in the same process. | • Good scalability<br>• Java routines created with the FENCED THREADSAFE clause (the default) scale well. All fenced Java routines will share a few JVMs. More than one JVM might be in use on the system if the Java heap of a particular db2fmp process is approaching exhaustion. | • To avoid potentially dangerous operations, Java Native Interface (JNI) calls from Java routines are not permitted. |

*Table 25. Comparison of external routine APIs and programming languages (continued)*

| API and programming language | Feature support | Performance | Security | Scalability | Limitations |
|---|---|---|---|---|---|
| .NET common language runtime supported languages (includes C#, Visual Basic, and others) | • Part of the Microsoft .NET model of managed code.<br>• Source code is compiled into intermediate language (IL) byte code that can be interpreted by the Microsoft .NET Framework common language runtime.<br>• CLR assemblies can be built up from sub-assemblies that were compiled from different .NET programming language source code, which allows users to re-use and integrate code modules written in various languages. | • CLR routines can only be created with the FENCED NOT THREADSAFE clause so as to minimize the possibility of database manager interruption at runtime. This can somewhat negatively impact performance | • CLR routines can only be created with the FENCED NOT THREADSAFE clause. They are therefore safe because they will be run outside of the database manager in a separate db2fmp process. | • No information available. | • Refer to the topic, ″Restrictions on .NET CLR routines″. |

*Table 25. Comparison of external routine APIs and programming languages (continued)*

| API and programming language | Feature support | Performance | Security | Scalability | Limitations |
|---|---|---|---|---|---|
| • OLE | • OLE routines can be implemented in Visual C++, Visual Basic, and other languages supported by OLE. | • The speed of OLE automated routines depends on the language used to implement them. In general they are slower than non-OLE C/C++ routines.<br>• OLE routines can only run in FENCED NOT THREADSAFE mode, and therefore OLE automated routines do not scale well. | • No information available. | • No information available. | • No information available. |

Table 25. Comparison of external routine APIs and programming languages (continued)

| API and programming language | Feature support | Performance | Security | Scalability | Limitations |
|---|---|---|---|---|---|
| • OLE DB | • OLE DB can be used to create user-defined table functions. • OLE DB functions connect to external OLE DB data sources. | • Performance of OLE DB functions depends on the OLE DB provider, however in general OLE DB functions perform better than logically equivalent Java functions, but slower than logically equivalent C, C++, or SQL functions. However some predicates from the query where the function is invoked might be evaluated at the OLE DB provider, therefore reducing the number of rows that DB2 has to process which can frequently result in improved performance. | • No information available. | • No information available. | • OLE DB can only be used to create user-defined table functions. |

**Related concepts:**

- "Supported database application programming interfaces" in *Getting Started with Database Application Development*
- "Support for external routine development in .NET CLR languages" in *Developing SQL and External Routines*
- "Support for external routine development in C" on page 285
- "Support for external routine development in C++" on page 285
- "Supported Java routine development software" in *Developing SQL and External Routines*
- "Supported APIs and programming languages for external routine development" on page 260

**Related reference:**

- "Supported programming languages and compilers for database application development" in *Getting Started with Database Application Development*
- "Support for external procedure development in COBOL" on page 354

## External routine creation

External routines are created in a similar way as routines with other implementations. However there are a few additional steps required because the routine implementation requires the coding, compilation, and deployment of source code.

There are two parts to an external routine:
- The CREATE statement that defines the routine.
- The external library or class that implements the routine-body

Upon the successful execution of a CREATE statement that defines a routine, the routine is created within the database. The statement must at a minimum define the name of the routine, the routine parameter signature that will be used in the routine implementation, and the location of the external library or class built from the routine implementation source code.

External routine implementation must be coded in one of the supported programming languages and then built into a library or class file that must be installed in the file system of the database server.

An external routine cannot be successfully invoked until it has been created in the database and the library or class associated with the routine has been put in the location specified by the EXTERNAL clause.

The development of external routines generally consists of the following tasks:
- Determining what functional type of routine to implement.
- Choosing one of the supported external routine programming languages for the routine implementation.
- Designing the routine.
- Connecting to a database and creating the routine in the database.
  - This is done by executing one of the CREATE PROCEDURE, CREATE FUNCTION, or CREATE METHOD statements or by using a graphical tool that automates this step.
  - This task, also known as defining or registering a routine, can occur at any time before you invoke the routine, except in the following circumstances:
    - For Java routines that reference an external JAR file or files, the external code and JAR files must be coded and compiled before the routine is created in the database using the routine type specific CREATE statement.
    - Routines that execute SQL statements and refer to themselves directly must be created in the database by issuing the CREATE statement before the external code associated with the routine is precompiled and bound. This also applies to situations where there is a cycle of references, for example, Routine A references Routine B, which references Routine A.
- Coding the routine logic such that it corresponds to the routine definition.
- Building the routine and generating a library or class file.
  - For embedded SQL routines this includes: precompiling , compiling, and link the code as well as binding the routine package to the target database.

  – For non-embedded SQL routines this includes: compiling and linking the code.
- Deploying the library or class file to the database server in the location specified in the routine definition.
- Granting the EXECUTE privilege on the routine to the routine invoker or invokers (if they are not the routine definer).
- Invoking, testing, and debugging the routine.

The steps required to create an external routine can all be done using the DB2 Command Line Processor or a DB2 Command Window. Tools can be of assistance in automating some or all of these steps.

**Related concepts:**
- "External routine features" on page 248
- "External routines" on page 247
- "Overview of external routines" on page 248

**Related tasks:**
- "Creating external routines" on page 282

## External routine parameter styles

External routine implementations must conform to a particular convention for the exchange of routine parameter values. These conventions are known as *parameter styles*. An external routine parameter style is specified when the routine is created by specifying the PARAMETER STYLE clause. Parameter styles characterize the specification and order in which parameter values will be passed to the external routine implementation. They also specify what if any additional values will be passed to the external routine implementation. For example, some parameter styles specify that for each routine parameter value that an additional separate null-indicator value be passed to the routine implementation to provide information about the parameters nullability which cannot otherwise be easily determined with a native programming language data type.

The table below provides a list of the available parameter styles, the routine implementations that support each parameter style, the functional routine types that support each parameter style, and a description of the parameter style:

*Table 26. Parameter styles*

| Parameter style | Supported language | Supported routine type | Description |
|---|---|---|---|
| SQL [1] | • C/C++<br>• OLE<br>• .NET common language runtime languages<br>• COBOL [2] | • UDFs<br>• stored procedures<br>• methods | In addition to the parameters passed during invocation, the following arguments are passed to the routine in the following order:<br>• A null indicator for each parameter or result declared in the CREATE statement.<br>• The SQLSTATE to be returned to DB2.<br>• The qualified name of the routine.<br>• The specific name of the routine.<br>• The SQL diagnostic string to be returned to DB2.<br><br>Depending on options specified in the CREATE statement and the routine type, the following arguments can be passed to the routine in the following order:<br>• A buffer for the scratchpad.<br>• The call type of the routine.<br>• The dbinfo structure (contains information about the database). |
| DB2SQL [1] | • C/C++<br>• OLE<br>• .NET common language runtime languages<br>• COBOL | • stored procedures | In addition to the parameters passed during invocation, the following arguments are passed to the stored procedure in the following order:<br>• A vector containing a null indicator for each parameter on the CALL statement.<br>• The SQLSTATE to be returned to DB2.<br>• The qualified name of the stored procedure.<br>• The specific name of the stored procedure.<br>• The SQL diagnostic string to be returned to DB2.<br><br>If the DBINFO clause is specified in the CREATE PROCEDURE statement, a dbinfo structure (it contains information about the database) is passed to the stored procedure. |
| JAVA | • Java | • UDFs<br>• stored procedures | PARAMETER STYLE JAVA routines use a parameter passing convention that conforms to the Java language and SQLJ Routines specification.<br><br>For stored procedures, INOUT and OUT parameters will be passed as single entry arrays to facilitate the returning of values. In addition to the IN, OUT, and INOUT parameters, Java method signatures for stored procedures include a parameter of type ResultSet[] for each result set specified in the DYNAMIC RESULT SETS clause of the CREATE PROCEDURE statement.<br><br>For PARAMETER STYLE JAVA UDFs and methods, no additional arguments to those specified in the routine invocation are passed.<br><br>PARAMETER STYLE JAVA routines do not support the DBINFO or PROGRAM TYPE clauses. For UDFs, PARAMETER STYLE JAVA can only be specified when there are no structured data types specified as parameters and no structured type, CLOB, DBCLOB, or BLOB data types specified as return types (SQLSTATE 429B8). Also, PARAMETER STYLE JAVA UDFs do not support table functions, call types, or scratchpads. |
| DB2GENERAL | • Java | • UDFs<br>• stored procedures<br>• methods | This type of routine will use a parameter passing convention that is defined for use with Java methods. Unless you are developing table UDFs, UDFs with scratchpads, or need access to the dbinfo structure, it is recommended that you use PARAMETER STYLE JAVA.<br><br>For PARAMETER STYLE DB2GENERAL routines, no additional arguments to those specified in the routine invocation are passed. |

Table 26. Parameter styles (continued)

| Parameter style | Supported language | Supported routine type | Description |
|---|---|---|---|
| GENERAL | • C/C++ <br> • .NET common language runtime languages <br> • COBOL | • stored procedures | A PARAMETER STYLE GENERAL stored procedure receives parameters from the CALL statement in the invoking application or routine. If the DBINFO clause is specified in the CREATE PROCEDURE statement, a dbinfo structure (it contains information about the database) is passed to the stored procedure. <br><br> GENERAL is the equivalent of SIMPLE stored procedures for DB2 Universal Database for z/OS and OS/390. |
| GENERAL WITH NULLS | • C/C++ <br> • .NET common language runtime languages <br> • COBOL | • stored procedures | A PARAMETER STYLE GENERAL WITH NULLS stored procedure receives parameters from the CALL statement in the invoking application or routine. Also included is a vector containing a null indicator for each parameter on the CALL statement. If the DBINFO clause is specified in the CREATE PROCEDURE statement, a dbinfo structure (it contains information about the database) is passed to the stored procedure. <br><br> GENERAL WITH NULLS is the equivalent of SIMPLE WITH NULLS stored procedures for DB2 Universal Database for z/OS and OS/390. |

**Note:**

1. For UDFs and methods, PARAMETER STYLE SQL is equivalent to PARAMETER STYLE DB2SQL.
2. COBOL can only be used to develop stored procedures.
3. .NET common language runtime methods are not supported.

**Related concepts:**

- "DB2GENERAL routines" in *Developing SQL and External Routines*
- "Java routines" in *Developing SQL and External Routines*
- "Overview of external routines" on page 248
- "Parameters in .NET CLR routines" in *Developing SQL and External Routines*
- "Parameters in C and C++ routines" on page 288
- "Parameters in Java routines" in *Developing SQL and External Routines*
- "Parameters in SQL procedures" in *Developing SQL and External Routines*

**Related tasks:**

- "Distinct types as UDF or method parameters" in *Developing SQL and External Routines*
- "Passing structured type parameters to external routines" in *Developing SQL and External Routines*

**Related reference:**

- "CREATE FUNCTION statement" in *SQL Reference, Volume 2*
- "CREATE METHOD statement" in *SQL Reference, Volume 2*
- "CREATE PROCEDURE statement" in *SQL Reference, Volume 2*
- "CREATE TYPE (Structured) statement" in *SQL Reference, Volume 2*
- "Passing arguments to C, C++, OLE, or COBOL routines" on page 311

## Routine library or class management

### External routine library and class management

To successfully develop and invoke external routines, external routine library and class files must be deployed and managed properly.

External routine library and class file management can be minimal if care is taken when external routines are first created and library and class files deployed.

The main external routine management considerations are the following:
- Deployment of external routine library and class files
- Security of external routine libary and class files
- Resolution of external routine libraries and classes
- Modifications to external routine library and class files
- Backup and restore of external routine library and class files

System administrators, database administrators and database application developers should all take responsibility to ensure that external routine library and class files are secure and correctly preserved during routine development and database administration tasks.

**Related concepts:**
- "Backup and restore of external routine library and class files" on page 275
- "Deployment of external routine libraries and classes" on page 272
- "Overview of external routines" on page 248
- "Resolution of external routine libraries and classes" on page 274
- "Restrictions on external routines" on page 279
- "Security of external routine library or class files" on page 273
- "Modifications to external routine library and class files" on page 275

**Related reference:**
- "ALTER FUNCTION statement" in *SQL Reference, Volume 2*
- "ALTER METHOD statement" in *SQL Reference, Volume 2*
- "ALTER PROCEDURE statement" in *SQL Reference, Volume 2*
- "CREATE FUNCTION statement" in *SQL Reference, Volume 2*
- "CREATE METHOD statement" in *SQL Reference, Volume 2*
- "CREATE PROCEDURE statement" in *SQL Reference, Volume 2*

### Deployment of external routine libraries and classes

Deployment of external routine libraries and classes refers to the copying of external routine libraries and classes to the database server once they have been built from source code.

External routine library, class, or assembly files must be copied into the DB2 *function directory* or a sub-directory of this directory on the database server. This is the recommended external routine deployment location. To find out more about the function directory, see the description of the EXTERNAL clause for either of the following SQL statements: CREATE PROCEDURE or CREATE FUNCTION.

You can copy the external routine class, library, or assembly to other directory locations on the server, depending on the API and programming language used to implement the routine, however this is generally discouraged. If this is done, to successfully invoke the routine you must take particular note of the fully qualified path name and ensure that this value is used with the EXTERNAL NAME clause.

Library and class files can be copied to the database server file system using most generally available file transfer tools. Java routines can be copied from a computer where a DB2 client is installed to a DB2 database server using special system-defined procedures designed specifically for this purpose. See the topics on Java routines for more details.

When executing the appropriate SQL language CREATE statement for the routine type: CREATE PROCEDURE or CREATE FUNCTION, be sure to specify the appropriate clauses, paying particular attention to the EXTERNAL NAME clause.

- Specify the `LANGUAGE` clause with the appropriate value for the chosen API or programming language. Examples include: CLR, C, JAVA.
- Specify the `PARAMETER STYLE` clause with the name of the supported parameter style that was implemented in the routine code.
- Specify the `EXTERNAL` clause with the name of the library, class, or assembly file to be associated with the routine using one of the following values:
  - the fully qualified path name of the routine library, class, or assembly file.
  - the relative path name of the routine library, class, or assembly file relative to the function directory.

By default DB2 will look for the library, class, or assembly file by name in the function directory unless a fully qualified or relative path name for it is specified in the EXTERNAL clause.

**Related concepts:**

- "External routine library management and performance" on page 276
- "Modifications to external routine library and class files" on page 275
- "Resolution of external routine libraries and classes" on page 274
- "Security of external routine library or class files" on page 273
- "Backup and restore of external routine library and class files" on page 275
- "External routine library and class management" on page 272

## Security of external routine library or class files

External routine libraries are stored in the file system on the database server and are not backed up or protected in any way by the DB2 database manager. For routines to continue to successfully be invoked, it is imperative that the library associated with the routine continue to exist in the location specified in the EXTERNAL clause of the CREATE statement used to create the routine. Do not move or delete routine libraries after creating routines; doing so will cause routine invocations to fail.

To prevent routine libraries from being accidentally or intentionally deleted or replaced, you must restrict access to the directories on the database server that contain routine libraries and restrict access to the routine library files. This can be done by using operating system commands to set directory and file permissions.

**Related concepts:**

- “Backup and restore of external routine library and class files” on page 275
- “External routine library and class management” on page 272
- “External routine library management and performance” on page 276
- “Modifications to external routine library and class files” on page 275

## Resolution of external routine libraries and classes

DB2 external routine library resolution is performed at the DB2 instance level. This means that in DB2 instances containing multiple DB2 databases, external routines can be created in one database that use external routine libraries already being used for a routine in another database.

Instance level external routine resolution supports code re-use by allowing multiple routine definitions to be associated with a single library. When external routine libraries are not re-used in this way, and instead copies of the external routine library exist in the file system of the database server, library name conflicts can happen. This can specifically happen when there are multiple databases in a single instance and the routines in each database are associated with their own copies of libraries and classes of routine bodies. A conflict arises when the name of a library or class used by a routine in one database is identical to the name of a library or class used by a routine in another database (in the same instance).

To minimize the likelihood of this happening, it is recommended that a single copy of a routine library be stored in the instance level function directory (sqllib/function directory) and that the EXTERNAL clause of all of the routine definitions in each of the databases reference the unique library.

If two functionally different routine libraries must be created with the same name, it is important to take additional steps to minimize the likelihood of library name conflicts.

**For C, C++, COBOL, and ADO.NET routines:**
Library name conflicts can be minimized or resolved by:

1. Storing the libraries with routine bodies in separate directories for each database.
2. Creating the routines with an EXTERNAL NAME clause value that specifies the full path of the given library (instead of a relative path).

**For Java routines:**
Class name conflicts cannot be resolved by moving the class files in question into different directories, because the CLASSPATH environment variable is instance-wide. The first class encountered in the CLASSPATH is the one that is used. Therefore, if you have two different Java routines that reference a class with the same name, one of the routines will use the incorrect class. There are two possible solutions: either rename the affected classes, or create a separate instance for each database.

**Related concepts:**

- “Backup and restore of external routine library and class files” on page 275
- “External routine library and class management” on page 272
- “External routine library management and performance” on page 276
- “Modifications to external routine library and class files” on page 275
- “Security of external routine library or class files” on page 273

## Modifications to external routine library and class files

Modifications to an existing external routine's logic might be necessary after an external routine has been deployed and it is in use in a production database system environment. Modifications to existing routines can be made, but it is important that they be done carefully so as to define a clear takeover point in time for the updates and to minimize the risk of interrupting any concurrent invocations of the routine.

If an external routine library requires an update, do not recompile and relink the routine to the same target file (for example, `sqllib/function/foo.a`) that the current routine is using while the database manager is running. If a current routine invocation is accessing a cached version of the routine process and the underlying library is replaced, this can cause the routine invocation to fail. If it is necessary to change the body of a routine without stopping and restarting DB2, complete the following steps:

1. Create the new external routine library with a different library or class file name.
2. If it is an embedded SQL routine, bind the routine package to the database using the BIND command.
3. Use the ALTER ROUTINE statement to change the routine definition so that the EXTERNAL NAME clause references the updated routine library or class. If the routine body to be updated is used by routines cataloged in multiple databases, the actions prescribed in this section must be completed for each affected database.
4. For updating Java routines that are built into JAR files, you must issue a CALL SQLJ.REFRESH_CLASSES() statement to force DB2 to load the new classes. If you do not issue the CALL SQLJ.REFRESH_CLASSES() statement after you update Java routine classes, DB2 continues to use the previous versions of the classes. DB2 refreshes the classes when a COMMIT or ROLLBACK occurs.

Once the routine definition has been updated, all subsequent invocations of the routine will load and run the new external routine library or class.

**Related concepts:**

- "Backup and restore of external routine library and class files" on page 275
- "External routine library and class management" on page 272
- "External routine library management and performance" on page 276
- "Security of external routine library or class files" on page 273

## Backup and restore of external routine library and class files

External routine libraries are not backed up with other database objects when a database backup is performed. They are similarly not restored when a database is restored.

If the purpose of a database backup and restore is to re-deploy a database, then external routine library files must be copied from the original database server file system to the target database server file system in such a way as to preserve the relative path names of the external routine libraries.

**Related concepts:**

- "External routine library and class management" on page 272
- "External routine library management and performance" on page 276

- “Modifications to external routine library and class files” on page 275
- “Security of external routine library or class files” on page 273

### External routine library management and performance

External routine library management can impact routine performance, because the DB2 database manager dynamically caches external routine libraries in an effort to improve performance in accordance with routine usage. For optimal external routine performance consider the following:

- Keep the number of routines in each library as small as possible. It is better to have numerous small external routine libraries than a few large ones.
- Group together within source code the routine functions of routines that are commonly invoked together. When the code is compiled into an external routine library the entry points of commonly invoked routines will be closer together which allows the database manager to provide better caching support. The improved caching support is due to the efficiency that can be gained by loading a single external routine libary once and then invoking multiple external routine functions within that library.

  For external routines implemented in the C or C++ programming language, the cost of loading a library is paid only once for libraries that are consistently in use by C routines. After a routine is invoked once, all subsequent invocations from the same thread in the process, do not need to re-load the routine’s library.

**Related concepts:**

- “Backup and restore of external routine library and class files” on page 275
- “External routine library and class management” on page 272
- “Modifications to external routine library and class files” on page 275

**Related tasks:**

- “Replacing an AIX shared library” in *Developing SQL and External Routines*

## 32-bit and 64-bit support for external routines

Support for 32-bit and 64-bit external routines is determined by the specification of one of the following two clauses in the CREATE statement for the routines: FENCED clause or NOT FENCED clause.

The routine-body of an external routine is written in a programming language and compiled into a library or class file that is loaded and run when the routine is invoked. The specification of the FENCED or NOT FENCED clause determines whether the external routine runs in a fenced environment distinct from the database manager or in the same addressing space as the database manager which can yield better performance through the use of shared memory instead of TCPIP for communications. By default routines are always created as fenced regardless of the other clauses selected.

The following table illustrates DB2’s support for running fenced and unfenced 32-bit and 64-bit routines on 32-bit and 64-bit database servers that are running the same operating system.

*Table 27. Support for 32-bit and 64-bit external routines*

| Bit-width of routine | 32-bit server | 64-bit server |
|---|---|---|
| 32-bit fenced procedure or UDF | Supported | Supported |

*Table 27. Support for 32-bit and 64-bit external routines (continued)*

| **Bit-width of routine** | **32-bit server** | **64-bit server** |
|---|---|---|
| 64-bit fenced procedure or UDF | Not supported (4) | Supported |
| 32-bit unfenced procedure or UDF | Supported | Supported (2) |
| 64-bit unfenced procedure or UDF | Not supported (4) | Supported |

The footnotes in the table above correspond to:

- (1) Running a 32-bit routine on a 64-bit server is not as fast as running a 64-bit routine on a 64-bit server.
- (2) 32-bit routines must be created as FENCED and NOT THREADSAFE to work on a 64-bit server.
- (3) It is not possible to invoke 32-bit routines on Linux IA 64-bit database servers.
- (4) 64-bit applications and routines cannot be run in 32-bit addressing spaces.

The important thing to note in the table is that 32-bit unfenced procedures cannot run on a 64-bit DB2 server. If you must deploy 32-bit unfenced routines to 64-bit platforms, remove the NOT FENCED clause from the CREATE statements for these routines before you catalog them.

**Related concepts:**

- "External routine library and class management" on page 272
- "External routines" on page 247
- "Overview of external routines" on page 248
- "External routine creation" on page 268
- "External routine features" on page 248
- "External routine implementation" in *Developing SQL and External Routines*

**Related tasks:**

- "Creating external routines" on page 282

## Performance of routines with 32-bit libraries on 64-bit database servers

It is possible to invoke routines with 32-bit routine libraries on 64-bit DB2 database servers. However, this does not perform as well as invoking a 64-bit routine on a 64-bit server. The reason for the performance degradation is that before each attempt to execute a 32-bit routine on a 64-bit server, an attempt is first made to invoke it as a 64-bit library. If this fails, the library is then invoked as a 32-bit library. A failed attempt to invoke a 32-bit library as a 64-bit library produces an error message (SQLCODE -444) in the db2diag.log.

Java classes are bit-width independent. Only Java virtual machines (JVMs) are classified as 32-bit or 64-bit. DB2 only supports the use of JVMs that are the same bit width as the instance in which they are used. In other words, in a 32-bit DB2 instance only a 32-bit JVM can be used, and in a 64-bit DB2 instance only a 64-bit JVM can be used. This ensures proper functioning of Java routines and the best possible performance.

**Related concepts:**

- “Overview of external routines” on page 248

## XML data type support in external routines

External procedures and functions written in the following programming languages support parameters and variables of data type XML:

- C
- C++
- COBOL
- Java
- .NET CLR languages

External OLE and OLEDB routines do not support parameters of data type XML.

XML data type values are represented in external routine code in the same way as CLOB data types.

When declaring external routine parameters of data type XML, the CREATE PROCEDURE and CREATE FUNCTION statements that will be used to create the routines in the database must specify that the XML data type is to be stored as a CLOB data type. The size of the CLOB value should be close to the size of the XML document represented by the XML parameter.

The CREATE PROCEDURE statement below shows a CREATE PROCEDURE statement for an external procedure implemented in the C programming language with an XML parameter named `parm1`:

```
CREATE PROCEDURE myproc(IN parm1 XML AS CLOB(2M), IN parm2 VARCHAR(32000))
LANGUAGE C
FENCED
PARAMETER STYLE SQL
EXTERNAL NAME 'mylib!myproc';
```

Similar considerations apply when creating external UDFs, as shown in the example below:

```
CREATE FUNCTION myfunc (IN parm1 XML AS CLOB(2M))
RETURNS SMALLINT
LANGUAGE C
PARAMETER STYLE SQL
DETERMINISTIC
NOT FENCED
NULL CALL
NO SQL
NO EXTERNAL ACTION
EXTERNAL NAME 'mylib1!myfunc'
```

XML values are serialized before they are passed to external routines.

Within external routine code, XML parameter and variable values are accessed, set, and modified in the same way as in database applications.

**Related concepts:**

- “Overview of external routines” on page 248
- “Encoding considerations for passing XML data in routine parameters” in *XML Guide*

- "Parameters and variables of data type XML in SQL functions" in *Developing SQL and External Routines*
- "XML and XQuery support in SQL procedures" in *Developing SQL and External Routines*

**Related tasks:**

- "Specifying a column data type" in *Developing SQL and External Routines*

**Related reference:**

- "Supported SQL data types for the DB2 .NET Data Provider" in *Developing SQL and External Routines*
- "Supported SQL data types in DB2GENERAL routines" in *Developing SQL and External Routines*
- "Supported SQL data types in Java routines" in *Developing SQL and External Routines*

## Restrictions on external routines

The following restrictions apply to external routines and should be considered when developing or debugging external routines.

**Restrictions that apply to all external routines::**

- New threads cannot be created in external routines.
- Connection level APIs cannot be called from within external functions or external methods.
- Receiving inputs from the keyboard and displaying outputs to standard output is not possible from external routines. Do not use standard input-output streams. For example:
  - In external Java routine code, do not issue the `System.out.println()` methods.
  - In external C or C++ routine code, do not issue `printf()`.
  - In external COBOL routine code, do not issue `display`

  Although external routines cannot display data to standard output, they can include code that writes data to a file on the database server file system.

  For fenced routines that run in UNIX environments, the target directory where the file is to be created, or the file itself, must have the appropriate permissions such that the owner of the `sqllib/adm/.fenced` file can create it or write to it. For not fenced routines, the instance owner must have create, read, and write permissions for the directory in which the file is opened.

  **Note:** DB2 does not attempt to synchronize any external input or output performed by a routine with DB2's own transactions. So, for example, if a UDF writes to a file during a transaction, and that transaction is later backed out for some reason, no attempt is made to discover or undo the writes to the file.
- Connection-related statements or commands cannot be executed in external routines. This restriction applies to the following statements: including:
  - BACKUP
  - CONNECT
  - CONNECT TO
  - CONNECT RESET

  - CREATE DATABASE
  - DROP DATABASE
  - FORWARD RECOVERY
  - RESTORE
- Operating system function usage within routines is not recommended. The use of these functions is not restricted except in the following cases:
  - **User-defined signal handlers must not be installed for external routines. Failure to adhere to this restriction can result in unexpectedexternal routine run-time failures, database abends, or other problems. Installing signal handlers can also interfere with operation of the JVM for Java routines.**
  - System calls that terminate a process can abnormally terminate one of DB2's processes and result in database system or database application failure.

    Other system calls can also cause problems if they interfere with the normal operation of the DB2; database manager. For example, a function that attempts to unload a library containing a user-defined function from memory could cause severe problems. Be careful in coding and testing external routines containing system calls.
- External routines must not contain commands that would terminate the current process. An external routine must always return control to the DB2 database manager without terminating the current process.
- External routine libraries, classes, or assemblies must not be updated while the database is active except in special cases. If an update is required while the DB2 database manager is active, and stopping and starting the instance is not an option, create the new library, class, or assembly for the routine with a different. Then, use the ALTER statement to change the external routine's EXTERNAL NAME clause value so that it references the name of the new library, class, or assembly file.
- Environment variable `DB2CKPTR` is not available in external routines. All other environment variables with names beginning with 'DB2' are captured at the time the database manager is started and are available for use in external routines.
- Some environment variables with names that do not start with 'DB2' are not available to external routines that are fenced. For example, the `LIBPATH` environment variable is not available for use. However these variables are available to external routines that are not fenced.
- Environment variable values that were set after the DB2 database manager is started are not available to external routines.
- Use of protected resources, resources that can only be accessed by one process at a time, within external routines should be limited. If used, try to reduce the likelihood of deadlocks when two external routines try to access the protected resource. If two or more external routines deadlock while attempting to access the protected resource, the DB2 database manager will not be able to detect or resolve the situation. This will result in hung external routine processes.
- Memory for external routine parameters should not be explicitly allocated on the DB2 database server. The DB2 database manager automatically allocates storage based upon the parameter declaration in the CREATE statement for the routine. Do not alter any storage pointers for parameters in external routines. Attempting to change a pointer with a locally created storage pointer can result in memory leaks, data corruption, or abends.
- Do not use static or global data in external routines. DB2 cannot guarantee that the memory used by static or global variables will be untouched between external routine invocations. For UDFs and methods, you can use scratchpads to store values for use between invocations.

- All SQL parameter values are buffered. This means that a copy of the value is made and passed to the external routine. If there are changes made to the input parameters of an external routine, these changes will have no effect on SQL values or processing. However, if an external routine writes more data to an input or output parameter than is specified by the CREATE statement, memory corruption has occurred, and the routine can abend.

**Restrictions that apply to external procedures only:**

- When returning result sets from nested stored procedures, you can open a cursor with the same name on multiple nesting levels. However, pre-version 8 applications will only be able to access the first result set that was opened. This restriction does not apply to cursors that are opened with a different package level.

**Restrictions that apply to external functions only:**

- External functions cannot return result sets. All cursors opened within an external function must be closed by the time the final-call invocation of the function completes.
- Dynamic allocations of memory in an external routine should be freed before the external routine returns. Failure to do so will result in a memory leak and the continuous growth in memory consumption of a DB2 process that could result in the database system running out of memory.

  For external user-defined functions and external methods, scratchpads can be used to allocate dynamic memory required for multiple function invocations. When scratchpads are used in this way, specify the FINAL CALL attribute in the CREATE FUNCTION or CREATE METHOD statement statement. This ensures that allocated memory is freed before the routine returns.

**Related concepts:**

- "Overview of external routines" on page 248
- "SQL data type handling in C and C++ routines" on page 303

**Related reference:**

- "Supported SQL data types in OLE automation" in *Developing SQL and External Routines*
- "Supported SQL data types in OLE DB" in *Developing SQL and External Routines*
- "Data type mappings between DB2 and OLE DB" in *Developing ADO.NET and OLE DB Applications*
- "Supported SQL data types in C and C++ embedded SQL applications" on page 53
- "ALTER FUNCTION statement" in *SQL Reference, Volume 2*
- "ALTER METHOD statement" in *SQL Reference, Volume 2*
- "ALTER PROCEDURE statement" in *SQL Reference, Volume 2*
- "CALL statement" in *SQL Reference, Volume 2*
- "CREATE FUNCTION statement" in *SQL Reference, Volume 2*
- "CREATE METHOD statement" in *SQL Reference, Volume 2*
- "CREATE PROCEDURE statement" in *SQL Reference, Volume 2*

## Creating external routines

External routines including procedures and functions are created in a similar way as routines with other implementations, however there are a few more steps required, because the routine implementation requires the coding, compilation, and deployment of source code.

You would choose to implement an external routine if:

- You want to encapsulate complex logic into a routine that accesses the database or that performs an action outside of the database.
- You require the encapsulated logic to be invoked from any of: multiple applications, the CLP, another routine (procedure, function (UDF), or method), or a trigger.
- You are most comfortable coding this logic in a programming language rather than using SQL and SQL PL statements.
- You require the routine logic to perform operations external to the database such as writing or reading to a file on the database server, the running of another application, or logic that cannot be represented with SQL and SQL PL statements.

**Prerequisites:**

- Knowledge of external routine implementation. To learn about external routines in general, see the topic:
  - "External routines" on page 247
  - "External routine creation" on page 268
- The DB2 DB2 Client must be installed.
- The database server must be running an operating system that supports the chosen implementation programming language compilers and development software.
- The required compilers and runtime support for the chosen programming language must be installed on the database server
- Authority to execute the CREATE PROCEDURE, CREATE FUNCTION, or CREATE METHOD statement.

**Restrictions:**

For a list of restrictions associated with external routines see:

- "Restrictions on external routines" on page 279

**Procedure:**

1. Code the routine logic in the chosen programming language.
   - For general information about external routines, routine features, and routine feature implementation, see the topics referenced in the Prerequisites section.
   - Use or import any required header files required to support the execution of SQL statemets.
   - Declare variables and parameters correctly using programming language data types that map to DB2 SQL data types.
2. Parameters must be declared in accordance with the format required by the parameter style for the chosen programming language. For more on parameters and prototype declarations see:
   - "External routine parameter styles" on page 269

3. Build your code into a library or class file.
4. Copy the library or class file into the DB2 *function directory* on the database server. It is recommended that you store assemblies or libraries associated with DB2 routines in the function directory. To find out more about the function directory, see the EXTERNAL clause of either of the following statements: CREATE PROCEDURE or CREATE FUNCTION.

   You can copy the assembly to another directory on the server if you wish, but to successfully invoke the routine you must note the fully qualified path name of your assembly as you will require it for the next step.
5. Execute either dynamically or statically the appropriate SQL language CREATE statement for the routine type: CREATE PROCEDURE or CREATE FUNCTION.
   - Specify the `LANGUAGE` clause with the appropriate value for the chosen API or programming language. Examples include: CLR, C, JAVA.
   - Specify the `PARAMETER STYLE` clause with the name of the supported parameter style that was implemented in the routine code.
   - Specify the `EXTERNAL` clause with the name of the library, class, or assembly file to be associated with the routine using one of the following values:
     – the fully qualified path name of the routine library, class, or assembly file .
     – the relative path name of the routine library, class, or assembly file relative to the function directory.

     By default DB2 will look for the library, class, or assembly file by name in the function directory unless a fully qualified or relative path name for it is specified in the EXTERNAL clause.
   - Specify DYNAMIC RESULT SETS with a numeric value if your routine is a procedure and it will return one or more result sets to the caller.
   - Specify any other clauses required to characterize the routine.

To invoke your external routine, see Routine invocation

**Related concepts:**
- "External routines" on page 247
- "External routine creation" on page 268
- "Restrictions on external routines" on page 279
- "External routine parameter styles" on page 269
- "Routine invocation" in *Developing SQL and External Routines*
- "Overview of external routines" on page 248

## C and C++ routines

### C and C++ routines

C and C++ routines are external routines that are created by executing a CREATE PROCEDURE, CREATE FUNCTION, or CREATE METHOD statement that references a library built from C or C++ source code as its external code body.

C and C++ routines can optionally execute SQL statements by including embedded SQL statements.

The following terms are important in the context of C and C++ routines:

**CREATE statement**
The SQL language CREATE statement used to create the routine in the database.

**Routine-body source code**
The source code file containing the C or C++ routine implementation that corresponds to the CREATE statement EXTERNAL clause specification.

**Precompiler**
The DB2 utility that pre-parses the routine source code implementation to validate SQL statements contained in the code and generates a package.

**Compiler**
The programming language specific software required to compile and link the source code implementation.

**Package**
The file containing the runtime access path information that DB2 will use at routine runtime to execute the SQL statements contained in the routine code implementation.

**Routine library**
A file that contains the compiled form of the routine source code. In Windows this is sometimes called a DLL, because these files have .dll file extensions.

Before developing a C or C++ routine, it is important to both understand the basics of routines and the unique features and characteristics specific to C and C++ routines. An understanding of the Embedded SQL API and the basics of embedded SQL application development is also important. To learn more about these subjects, refer to the following topics:

- External routines
- Embedded SQL
- Include files for C and C++ routines
- Parameters in C and C++ routines
- Restrictions on C and C++ routines

Developing a C or C++ routines involves following a series of step by step instructions and looking at C or C++ routine examples. Refer to:

- Creating C and C++ routines
- Examples of C procedures
- Examples of C user-defined functions

**Related concepts:**

- "Tools for developing C and C++ routines" on page 285
- "External routines" on page 247
- "Support for external routine development in C" on page 285
- "Support for external routine development in C++" on page 285

**Related tasks:**

- "Building C and C++ routine code" on page 328
- "Creating C and C++ routines" on page 326
- "Designing C and C++ routines" on page 286

## Support for external routine development in C

To develop external routines in C you must use supported compilers and development software.

The supported compilers and development software for DB2 database application development in C can all be used for external routine development in C.

**Related concepts:**
- “C and C++ routines” on page 283
- “Supported APIs and programming languages for external routine development” on page 260

**Related reference:**
- “Support for database application development in C” in *Getting Started with Database Application Development*
- “Supported programming languages and compilers for database application development” in *Getting Started with Database Application Development*

## Support for external routine development in C++

To develop external routines in C++ you must use supported compilers and development software.

The supported compilers and development software for DB2 database application development in C can all be used for external routine development in C++.

**Related concepts:**
- “Supported APIs and programming languages for external routine development” on page 260
- “C and C++ routines” on page 283

**Related reference:**
- “Support for database application development in C++” in *Getting Started with Database Application Development*
- “Supported programming languages and compilers for database application development” in *Getting Started with Database Application Development*

## Tools for developing C and C++ routines

The tools supported for C and C++ routines are the same as those supported for embedded SQL C and C++ applications.

There are no DB2 development environments or graphical user interface tools for developing, debugging, or deploying embedded SQL applications or routines.

The following command line interfaces are commonly used for developing, debugging, and deploying embedded SQL applications and routines:
- DB2 Command Line Processor
- DB2 Command Window

These interfaces support the execution of the SQL statements required to create routines in a database. The PREPARE command and the BIND command required to build C and C++ routines that contain embedded SQL can also be issued from these interfaces.

**Related concepts:**
- "C and C++ routines" on page 283
- "DB2 Command Line Processor (CLP)" in *Developing SQL and External Routines*
- "Tools for developing routines" in *Developing SQL and External Routines*

**Related tasks:**
- "Building C and C++ routine code" on page 328

## Designing C and C++ routines

### Designing C and C++ routines

Designing C and C++ routines is a task that should precede creating C and C++ routines. Designing C and C++ routines is generally related to both designing external routines implemented in other programming languages and designing embedded SQL applications.

**Prerequisites:**
- General knowledge of external routines
- C or C++ programming experience
- Optional: Knowledge of and experience with embedded SQL or CLI application development (if the routine will execute SQL statements)

The following topics can provide you with some of the required prerequisite information.

For more information on the features and uses of external routines:
- Refer to the topic, External routines

For more information on the characteristics of the embedded SQL API:
- Refer to the topic, Embedded SQL

**Procedure:**

With the prerequisite knowledge, designing embedded SQL routines consists mainly of learning about the unique features and characteristics of C and C++ routines:
- "Include file required for C and C++ routine development (sqludf.h)" on page 287
- "Parameters in C and C++ routines" on page 288
- "Parameter style SQL C and C++ procedures" on page 290
- "Parameter style SQL C and C++ functions" on page 293
- "SQL data type handling in C and C++ routines" on page 303
- "Graphic host variables in C and C++ routines" on page 322
- "Returning result sets from C and C++ procedures" on page 325
- "C++ type decoration" on page 323

- “Restrictions on external routines” on page 279

After having learned about the C and C++ characteristics, you might want to:

- “Creating C and C++ routines” on page 326

**Related concepts:**

- “External routine implementation” in *Developing SQL and External Routines*
- “Introduction to embedded SQL” on page 3
- “Parameter style SQL C and C++ procedures” on page 290
- “Parameter style SQL C and C++ functions” on page 293
- “SQL data type handling in C and C++ routines” on page 303
- “Graphic host variables in C and C++ routines” on page 322
- “Returning result sets from C and C++ procedures” on page 325
- “C++ type decoration” on page 323
- “Restrictions on external routines” on page 279
- “Precompilation and bind considerations for embedded SQL routines” on page 326
- “C and C++ routines” on page 283
- “Include file required for C and C++ routine development (sqludf.h)” on page 287
- “Parameters in C and C++ routines” on page 288

**Related tasks:**

- “Creating C and C++ routines” on page 326

## Include file required for C and C++ routine development (sqludf.h)

The `sqludf.h` include file contains structures, definitions, and values that are useful when writing routines. Although this file has ’udf’ in its name, (for historical reasons) it is also useful for stored procedures and methods. When compiling your routine, you need to reference the directory that contains this file. This directory is `sqllib/include`.

The `sqludf.h` include file is self-describing. The following is a brief summary of its content:

1. Structure definitions for arguments that are represented by structures in C or C++:
   - VARCHAR FOR BIT DATA arguments and result
   - LONG VARCHAR (with or without FOR BIT DATA) arguments and result
   - LONG VARGRAPHIC arguments and result
   - All the LOB types, SQL arguments and result
   - The scratchpad
   - The dbinfo structure
2. C language type definitions for all the SQL data types, for use in the definition of routine arguments corresponding to SQL arguments and result having the data types. These are the definitions with names SQLUDF_x and SQLUDF_x_FBD where x is a SQL data type name, and FBD represents For Bit Data.

   Also included is a C language type for an argument or result that is defined with the AS LOCATOR clause. This is applicable only to UDFs and methods.

3. Definition of C language types for the *scratchpad* and *call-type* arguments, with an `enum` type definition of the *call-type* argument.
4. Macros for defining the standard *trailing* arguments, both with and without the inclusion of *scratchpad* and *call-type* arguments. This corresponds to the presence and absence of SCRATCHPAD and FINAL CALL keywords in the function definition. These are the *SQL-state*, *function-name*, *specific-name*, *diagnostic-message*, *scratchpad*, and *call-type* UDF invocation arguments. Also included are definitions for referencing these constructs, and the various valid SQLSTATE values.
5. Macros for testing whether the SQL arguments are null.

A corresponding include file for COBOL exists: `sqludf.cbl`. This file only includes definitions for the scratchpad and dbinfo structures.

**Related concepts:**
- "SQL data type handling in C and C++ routines" on page 303

**Related tasks:**
- "Designing C and C++ routines" on page 286

**Related reference:**
- "Passing arguments to C, C++, OLE, or COBOL routines" on page 311
- "Supported SQL data types in C and C++ routines" on page 300

## Parameters in C and C++ routines

**Parameters in C and C++ routines:** Parameter declaration in C and C++ routines must conform to the requirements of one of the supported parameter styles and the program type. If the routine is to use a scratchpad, the dbinfo structure, or to have a PROGRAM TYPE MAIN parameter interface, there are additional details to consider including:
- Parameter styles for C and C++ routines
- "Parameter style SQL C and C++ procedures" on page 290
- "Parameter style SQL C and C++ functions" on page 293
- "Parameter null indicators in C and C++ routines" on page 289
- "Passing parameters by value or by reference in C and C++ routines" on page 295
- "Parameters are not required for C and C++ procedure result sets" on page 295
- Dbinfo structure as C or C++ routine parameter
- "Scratchpad as C or C++ function parameter" on page 298
- Program type MAIN supported for C and C++ procedures

It is very important that you implement the parameter interface to C and C++ routines correctly. This can be easily done with just a bit of care taken to ensure that the correct parameter style and data types are chosen and implemented according to the specification.

**Related concepts:**
- "XML data type support in external routines" on page 278

**Related tasks:**
- "Designing C and C++ routines" on page 286

**Parameter styles supported for C and C++ routines:** The following parameter styles are supported for C and C++ routines:
- SQL (Supported for procedures and functions; recommended)
- GENERAL (Supported for procedures)
- GENERAL WITH NULLS (Supported for procedures)

It is strongly recommended that the parameter style SQL be used for all C and C++ routines. This parameter style supports NULL values, provides a standard interface for reporting errors, as well as supporting scratchpads and call types.

To specify the parameter style to be used for a routine, you must specify the PARAMETER STYLE clause in the CREATE statement for the routine at routine creation time.

The parameter style must be accurately reflected in the implementation of the C or C++ routine code.

For more information about these parameter styles refer to: ″Syntax for passing parameters to C and C++ routines″.

**Related concepts:**
- "Dbinfo structure as C or C++ routine parameter" on page 295
- "Parameter style SQL C and C++ functions" on page 293
- "Parameter style SQL C and C++ procedures" on page 290
- "Parameters in C and C++ routines" on page 288
- "Passing parameters by value or by reference in C and C++ routines" on page 295

**Related reference:**
- "Passing arguments to C, C++, OLE, or COBOL routines" on page 311

**Parameter null indicators in C and C++ routines:** If the parameter style chosen for a C or C++ routine (procedure or function) requires that a null indicator parameter be specified for each of the SQL parameters, as is required by parameter style SQL, the null indicators are to be passed as parameters of data type SQLUDF_NULLIND. For parameter style GENERAL WITH NULLS, they must be passed as an array of type SQUDF_NULLIND. This data type is defined in embedded SQL application and routine include file: sqludf.h.

Null-indicator parameters indicate whether the corresponding parameter value is equivalent to NULL in SQL or if it has a literal value. If the null indicator value for a parameter is 0, this indicates that the parameter value is not null. If the null-indicator value for a parameter is -1, the parameter is to be considered to have a value equivalent to the SQL value NULL.

When null indicators are used it is important to include code within your routine that:
- Checks null-indicator values for input parameters before using them.
- Sets null indicator values for output parameters before the routine returns.

For more information about parameter SQL refer to:
- "External routine parameter styles" on page 269

**Related concepts:**
- "Parameter styles supported for C and C++ routines" on page 289
- "Parameters are not required for C and C++ procedure result sets" on page 295
- "Parameters in C and C++ routines" on page 288
- "Passing parameters by value or by reference in C and C++ routines" on page 295
- "XML data type support in external routines" on page 278

**Parameter style SQL C and C++ procedures:** C and C++ procedures should be created using the PARAMETER STYLE SQL clause in the CREATE PROCEDURE statement. The parameter passing conventions of this parameter style should be implemented in the corresponding procedure code implementation.

The C and C++ PARAMETER STYLE SQL signature implementation required for procedures follows this format:

```
    SQL_API_RC SQL_API_FN function-name (
                                         SQL-arguments,
                                         SQL-argument-inds,
                                         sqlstate,
                                         routine-name,
                                         specific-name,
                                         diagnostic-message )
```

`SQL_API_RC SQL_API_FN`
: `SQL_API_RC` and `SQL_API_FN` are macros that specify the return type and calling convention for a C or C++ procedure, which can vary across supported operating systems. The use of these macros is required for C and C++ routines. The macros are declared in embedded SQL application and routine include file `sqlsystm.h`.

*function-name*
: Name of the C or C++ function within the code file. This value does not have to be the same as the name of the procedure specified within the corresponding CREATE PROCEDURE statement. This value in combination with the library name however must be specified in the EXTERNAL NAME clause to identify the correct function entry point within the library to be used. For C++ routines, the C++ compiler applies type decoration to the entry point name. Either the type decorated name needs to be specified in the EXTERNAL NAME clause, or the entry point should be defined as `extern "C"` in the user code. The function name must be explicitly exported.

*SQL-arguments*
: C or C++ arguments that correspond to the set of SQL parameters specified in the CREATE PROCEDURE statement. IN, OUT, and INOUT mode parameters are passed using individual pointer values.

*SQL-argument-inds*
: C or C++ null indicators that correspond to the set of SQL parameters specified in the CREATE PROCEDURE statement. For each IN, OUT, and INOUT mode parameter, there must be an associated null-indicator parameter. Null indicators can be passed as individual arguments of type SQLUDF_NULLIND or as part of a single array of null indicators defined as SQLUDF_NULLIND*.

*sqlstate*
: Input-output parameter value used by the routine to signal warning or error conditions. Typically this argument is used to assign a user-defined SQLSTATE vallue that corresponds to an error or a warning that can be

passed back to the caller. SQLSTATE values of the form 38xxx, where xxx is any numeric value are available for user-defined SQLSTATE error values. SQLSTATE values of the form 01Hxx where xx is any numeric value are available for user-defined SQLSTATE warning values.

*routine-name*
: Input parameter value that contains the qualified routine name. This value is generated by DB2 and passed to the routine in the form `<schema-name>.<routine-name>` where <schema-name> and <routine-name> correspond respectively to the ROUTINESCHEMA column value and ROUTINENAME column value for the routine within the SYSCAT.ROUTINES catalog view. This value can be useful if a single routine implementation is used by multiple different routine definitions. When the routine definition name is passed into the routine, logic can be conditionally executed based on which definition was used. The routine name can also be useful when formulating diagnostic information including error messages, or when writing to a log file.

*specific-name*
: Input parameter value that contains the unique routine specific name. This value is generated by DB2 and passed to the routine. This value corresponds to the SPECIFICNAME column value for the routine in the SYSCAT.ROUTINES view. It can be useful in the same way as the routine-name.

*diagnostic-message*
: Output parameter value optionally used by the routine to return message text to the invoking application or routine. This parameter is intended to be used as a compliment to the SQLSTATE argument. It can be used to assign a user-defined error-message to accompany a user-defied SQLSTATE value which can provide more detailed diagnostic error or warning information to the caller of the routine.

**Note:** To simplify the writing of C and C++ procedure signatures the macro definition SQLUDF_TRAIL_ARGS defined in sqludf.h can be used in the procedure signature in place of using individual arguments to implement the non-SQL data type arguments.

The following is an example of a C or C++ procedure implementation that accepts a single input parameter, and returns a single output parameter and a result set:

```
/****************************************************************
 Routine:   cstp

  Purpose:  Returns an output parameter value based on an input
            parameter value

            Shows how to:
             - define a procedure using PARAMETER STYLE SQL
             - define NULL indicators for the parameter
             - execute an SQL statement
             - how to set a NULL indicator when parameter is
               not null

  Parameters:

  IN:       inParm
  OUT:      outParm

            When PARAMETER STYLE SQL is defined for the routine
            (see routine registration script spcreate.db2), in
            addition to the parameters passed during invocation,
            the following arguments are passed to the routine
            in the following order:

             - one null indicator for each IN/INOUT/OUT parameter
               ordered to match order of parameter declarations
             - SQLSTATE to be returned to DB2 (output)
             - qualified name of the routine (input)
             - specific name of the routine (input)
             - SQL diagnostic string to return an optional
               error message text to DB2 (output)

            See the actual parameter declarations below to see
            the recommended datatypes and sizes for them.

            CODE TIP:
            --------
            Instead of coding the 'extra' parameters:
              sqlstate, qualified name of the routine,
              specific name of the routine, diagnostic message,
            a macro SQLUDF_TRAIL_ARGS can be used instead.
            This macro is defined in DB2 include file sqludf.h

            TIP EXAMPLE:
            ------------
            The following is equivalent to the actual prototype
            used that makes use of macro definitions inclded in
            sqludf.h.  The form actually implemented is simpler
            and removes datatype concerns.

   extern "C"  SQL_API_RC SQL_API_FN OutLanguage(
                                                 sqlint16 *inParm,
                                                 double *outParm,
                                                 sqlint16 *inParmNullInd,
                                                 sqlint16 *outParmNullInd,
                                                 char sqlst[6],
                                                 char qualName[28],
                                                 char specName[19],
                                                 char diagMsg[71])
                                                )

******************************************************************/

extern "C" SQL_API_RC SQL_API_FN cstp ( sqlint16 *inParm,
                                        double *outParm,
                                        SQLUDF_NULLIND *inParmNullInd,
                                        SQLUDF_NULLIND *outParmNullInd,
                                        SQLUDF_TRAIL_ARGS )
{
  EXEC SQL INCLUDE SQLCA;

  EXEC SQL BEGIN DECLARE SECTION;
     sqlint16 sql_inParm;
  EXEC SQL END DECLARE SECTION;

  sql_inParm = *inParm;

  EXEC SQL DECLARE cur1 CURSOR FOR
   SELECT value
   FROM table01
    WHERE index = :sql_inParm;

  *outParm = (*inParm) + 1;
  *outParmNullInd = 0;
  EXEC SQL OPEN cur1;

  return (0);
}
```

The corresponding CREATE PROCEDURE statement for this procedure follows:

```
 CREATE PROCEDURE cproc( IN inParm INT, OUT outParm INT )
 LANGUAGE c
 PARAMETER STYLE sql
 DYNAMIC RESULT SETS 1
 FENCED
 THREADSAFE
 RETURNS NULL ON NULL INPUT
 EXTERNAL NAME 'c_rtns!cstp'
```

The preceding statement assumes that the C or C++ procedure implementation is in a library file named `c_rtns` and a function named `cstp`.

**Parameter style SQL C and C++ functions:** C and C++ user-defined functions should be created using the PARAMETER STYLE SQL clause in the CREATE FUNCTION statement. This The parameter passing conventions of this parameter style should be implemented in the corresponding source code implementation. The C and C++ PARAMETER STYLE SQL signature implementation required for user-defined functions follows this format:

```
 SQL_API_RC SQL_API_FN function-name ( SQL-arguments,
  SQL-argument-inds,
  SQLUDF_TRAIL_ARGS )
```

`SQL_API_RC SQL_API_FN`
: `SQL_API_RC` and `SQL_API_FN` are macros that specify the return type and calling convention for a C or C++ user-defined function, which can vary across supported operating systems. The use of these macros is required for C and C++ routines. The macros are declared in embedded SQL application and routine include file `sqlsystm.h`.

*function-name*
: Name of the C or C++ function within the code file. This value does not have to be the same as the name of the function specified within the corresponding CREATE FUNCTION statement. This value in combination with the library name however must be specified in the EXTERNAL NAME clause to identify the correct function entry point within the library to be used. For C++ routines, the C++ compiler applies type decoration to the entry point name. Either the type decorated name needs to be specified in the EXTERNAL NAME clause, or the function declaration within the source code file should be prefixed with `extern "C"` as shown in the following example: extern ″C″ SQL_API_RC SQL_API_FN OutLanguage( char *, sqlint16 *, char *, char *, char *, char *);

*SQL-arguments*
: C or C++ arguments that correspond to the set of SQL parameters specified in the CREATE FUNCTION statement.

*SQL-argument-inds*
: For each SQL-argument a null indicator parameter is required to specify whether the parameter value is intended to be interpreted within the routine implementation as a NULL value in SQL. Null indicators must be specified with data type SQLUDF_NULLIND. This data type is defined in embedded SQL routine include file `sqludf.h`.

SQLUDF_TRAIL_ARGS

A macro defined in embedded SQL routine include file sqludf.h that once expanded defines the additional trailing arguments required for a complete parameter style SQL signature. There are two macros that can be used: SQLUDF_TRAIL_ARGS and SQLUDF_TRAIL_ARGS_ALL. SQLUDF_TRAIL_ARGS when expanded, as defined in sqludf.h, is equivalent to the addition of the following routine arguments:

```
SQLUDF_CHAR *sqlState,
SQLUDF_CHAR qualName,
SQLUDF_CHAR specName,
SQLUDF_CHAR *sqlMessageText,
```

In general these arguments are not required or generally used as part of user-defined function logic. They represent the output SQLSTATE value to be passed back to the function invoker, the input fully qualified function name, input function specific name, and output message text to be returned with the SQLSTATE. SQLUDF_TRAIL_ARGS_ALL when expanded, as defined in sqludf.h, is equivalent to the addition of the following routine arguments:

```
SQLUDF_CHAR  qualName,
SQLUDF_CHAR  specName,
SQLUDF_CHAR  sqlMessageText,
SQLUDF_SCRAT *scratchpad
SQLUDF_CALLT *callType
```

If the UDF CREATE statement includes the SCRATCHPAD clause or the FINAL CALL clause, then the macro SQLUDF_TAIL_ARGS_ALL must be used. In addition to arguments provided with SQLUDF_TRAIL_ARGS, this macro also contains pointers to a scratchpad structure, and a call type value.

The following is an example of a simple C or C++ UDF that returns in an output parameter the value of the product of its two input parameter values:

```
SQL_API_RC SQL_API_FN product ( SQLUDF_DOUBLE *in1,
                                SQLUDF_DOUBLE *in2,
                                SQLUDF_DOUBLE *outProduct,
                                SQLUDF_NULLIND *in1NullInd,
                                SQLUDF_NULLIND *in2NullInd,
                                SQLUDF_NULLIND *productNullInd,
                                SQLUDF_TRAIL_ARGS )
{

  /* Check that input parameter values are not null
     by checking the corresponding null indicator values
         0  : indicates parameter value is not NULL
         -1 : indicates parameter value is NULL

    If values are not NULL, calculate the product.
    If values are NULL, return a NULL output value. */

  if ((*in1NullInd != -1) &&
      *in2NullInd != -1))
  {
    *outProduct = (*in1) * (*in2);
    *productNullInd = 0;
  }
  else
  {
    *productNullInd = -1;
  }
  return (0);
}
```

The corresponding CREATE FUNCTION statement that can be used to create this UDF could be:

```
CREATE FUNCTION product( in1 DOUBLE, in2 DOUBLE )
  RETURNS DOUBLE
  LANGUAGE C
  PARAMETER STYLE SQL
  NO SQL
  FENCED THREADSAFE
  DETERMINISTIC
  RETURNS NULL ON NULL INPUT
  NO EXTERNAL ACTION
  EXTERNAL NAME 'c_rtns!product'
```

The preceding SQL statement assumes that the C or C++ function is in a library file in the function directory named `c_rtns`.

**Passing parameters by value or by reference in C and C++ routines:** For C and C++ routines, parameter values must always be passed by reference to routines using pointers. This is required for input-only, input-output, and output parameters. by reference.

Null-indicator parameters must also be passed by reference to routines using pointers.

**Note:** DB2 controls the allocation of memory for all parameters and maintains C or C++ references to all parameters passed into or out of a routine. There is no need to allocate or free memory associated with routine parameters and null indicators.

**Parameters are not required for C and C++ procedure result sets:** No parameter is required in the CREATE PROCEDURE statement signature for a procedure or in the associated procedure implementation in order to return a result set to the caller.

Result sets returned from C procedures, are returned using cursors.

For more on returning result sets from LANGUAGE C procedures, see:
- “Returning result sets from C and C++ procedures” on page 325

**Related concepts:**
- “Parameter style SQL C and C++ functions” on page 293
- “Parameter null indicators in C and C++ routines” on page 289
- “Parameter styles supported for C and C++ routines” on page 289
- “Parameter style SQL C and C++ procedures” on page 290
- “Parameters in C and C++ routines” on page 288

**Related tasks:**
- “Returning result sets from .NET CLR procedures” in *Developing SQL and External Routines*

**Dbinfo structure as C or C++ routine parameter:** The `dbinfo` structure is a structure that contains database and routine information that can be passed to and from a routine implementation as an extra argument if and only if the DBINFO clause is included in the CREATE statement for the routine.

The dbinfo structure is supported LANGUAGE C routines through the use of the `sqludf_dbinfo` structure. This C structure is defined in the DB2 include file `sqludf.h` located in the `sqllib\include` direcory.

The `sqludf_dbinfo` structure is defined as follows:

```
SQL_STRUCTURE sqludf_dbinfo
{
   unsigned short  dbnamelen;                       /* Database name length     */
   unsigned char   dbname[SQLUDF_MAX_IDENT_LEN];    /* Database name            */
   unsigned short  authidlen;                       /* Authorization ID length  */
   unsigned char   authid[SQLUDF_MAX_IDENT_LEN];    /* Authorization ID         */
   union db_cdpg   codepg;                          /* Database code page       */
   unsigned short  tbschemalen;                     /* Table schema name length */
   unsigned char   tbschema[SQLUDF_MAX_IDENT_LEN];  /* Table schema name        */
   unsigned short  tbnamelen;                       /* Table name length        */
   unsigned char   tbname[SQLUDF_MAX_IDENT_LEN];    /* Table name               */
   unsigned short  colnamelen;                      /* Column name length       */
   unsigned char   colname[SQLUDF_MAX_IDENT_LEN];   /* Column name              */
   unsigned char   ver_rel[SQLUDF_SH_IDENT_LEN];    /* Database version/release */
   unsigned char   resd0[2];                        /* Alignment                */
   sqluint32       platform;                        /* Platform                 */
   unsigned short  numtfcol;                        /* # of entries in TF column*/
                                                    /* List array               */
   unsigned char   resd1[2];                        /* Reserved                 */
   sqluint32       procid;                          /* Current procedure ID     */
   unsigned char   resd2[32];                       /* Reserved                 */
   unsigned short  *tfcolumn;                       /* Tfcolumn to be allocated */
                                                    /* dynamically if a table   */
                                                    /* function is defined;     */
                                                    /* else a NULL pointer      */
   char            *appl_id;                        /* Application identifier   */
   sqluint32       dbpartitionnum;                  /* Database partition number*/
                                                    /* where routine executed   */
   unsigned char   resd3[16];                       /* Reserved                 */
};
```

Although, not all of the fields in the `dbinfo` structure might be useful within a routine, several of the values in this structure's fields might be useful when formulating diagnostic error message information. For example, if an error occurs within a routine, it might be useful to return the database name, database name length, the database code page, the current authorization ID, and the length of the current authorization ID.

To reference the `sqludf_dbinfo` structure in a LANGUAGE C routine implementation:

- Add the DBINFO clause to the CREATE statement that defines the routine.
- Include the `sqludf.h` header file at the top of the file containing the routine implementation.
- Add a parameter of type `sqludf_dbinfo` to the routine signature in the position specified by the parameter style used.

Here is an example of a C procedure with PARAMETER STYLE GENERAL that demonstrates the use of the `dbinfo` structure. Here is the CREATE PROCEDURE statement for the procedure. Note that as specified by the EXTENAL NAME clause, the procedure implementation is located in a library file named `spserver` that contains a C function named `DbinfoExample`:

```
  CREATE PROCEDURE DBINFO_EXAMPLE (IN job CHAR(8),
                                   OUT salary DOUBLE,
                                   OUT dbname CHAR(128),
                                   OUT dbversion CHAR(8),
                                   OUT errorcode INTEGER)
  DYNAMIC RESULT SETS 0
  LANGUAGE C
  PARAMETER STYLE GENERAL
  DBINFO
  FENCED NOT THREADSAFE
  READS SQL DATA
  PROGRAM TYPE SUB
  EXTERNAL NAME 'spserver!DbinfoExample'@
```

Here is the C procedure implementation that corresponds to the procedure definition:

```
/***************************************************************
 Routine:    DbinfoExample

   IN:       inJob         - a job type, used in a SELECT predicate
   OUT:      salary        - average salary of employees with job injob
             dbname        - database name retrieved from DBINFO
             dbversion     - database version retrieved from DBINFO
             outSqlError   - sqlcode of error raised (if any)
             sqludf_dbinfo - pointer to DBINFO structure

  Purpose:  This routine takes in a job type and returns the
            average salary of all employees with that job, as
            well as information about the database (name,
            version of database).  The database information
            is retrieved from the dbinfo object.

  Shows how to:
              - define IN/OUT parameters in PARAMETER STYLE GENERAL
              - declare a parameter pointer to the dbinfo structure
              - retrieve values from the dbinfo structure
***************************************************************/
SQL_API_RC SQL_API_FN DbinfoExample(char inJob[9],
                                    double *salary,
                                    char dbname[129],
                                    char dbversion[9],
                                    sqlint32 *outSqlError,
                                    struct sqludf_dbinfo * dbinfo
                                    )
{
  /* Declare a local SQLCA */
  struct sqlca sqlca;

  EXEC SQL WHENEVER SQLERROR GOTO return_error;

  /* SQL host variable declaration section */
  /* Each host variable names must be unique within a code
     file, or the the precompiler raises SQL0307 error */
  EXEC SQL BEGIN DECLARE SECTION;
  char dbinfo_injob[9];
  double dbinfo_outsalary;
  sqlint16 dbinfo_outsalaryind;
  EXEC SQL END DECLARE SECTION;

  /* Initialize output parameters - se strings to NULL */
  memset(dbname, '\0', 129);
  memset(dbversion, '\0', 9);
  *outSqlError = 0;

  /* Copy input parameter into local host variable */
  strcpy(dbinfo_injob, inJob);

  EXEC SQL SELECT AVG(salary) INTO:dbinfo_outsalary
             FROM employee
               WHERE job =:dbinfo_injob;
  *salary = dbinfo_outsalary;

  /* Copy values from the DBINFO structure into the output parameters
     You must explicitly null-terminate the strings.
     Information such as the database name, and the version of the
     database product can be found in the DBINFO structure as well as
     other information fields. */

  strncpy(dbname, (char *)(dbinfo->dbname), dbinfo->dbnamelen);
  dbname[dbinfo->dbnamelen] = '\0';
  strncpy(dbversion, (char *)(dbinfo->ver_rel), 8);
  dbversion[8] = '\0';

  return 0;

  /* Copy SQLCODE to OUT parameter if SQL error occurs */

  return_error:
  {
    *outSqlError = SQLCODE;
    EXEC SQL WHENEVER SQLERROR CONTINUE;
    return 0;
  }
} /* DbinfoExample function */
```

**Scratchpad as C or C++ function parameter:** The scratchpad structure, used for storing UDF values between invocations for each UDF input value, is supported in C and C++ routines through the use of the sqludf_scrat structure. This C structure is defined in the DB2 include file sqludf.h.

To reference the sqludf_scrat structure, include the sqludf.h header file at the top of the file containing the C or C++ function implementation, and use the SQLUDF_TRAIL_ARGS_ALL macro within the signature of the routine implementation.

The following example demonstrates a C scalar function implementation that includes a parameter of type SQLUDF_TRAIL_ARGS_ALL:

```
  #ifdef __cplusplus
  extern "C"
  #endif
  void SQL_API_FN ScratchpadScUDF(SQLUDF_INTEGER *outCounter,
                                  SQLUDF_SMALLINT *counterNullInd,
                                  SQLUDF_TRAIL_ARGS_ALL)
  {
    struct scalar_scratchpad_data *pScratData;

    /* SQLUDF_CALLT and SQLUDF_SCRAT are */
    /* parts of SQLUDF_TRAIL_ARGS_ALL */

    pScratData = (struct scalar_scratchpad_data *)SQLUDF_SCRAT->data;
    switch (SQLUDF_CALLT)
    {
      case SQLUDF_FIRST_CALL:
        pScratData->counter = 1;
        break;
      case SQLUDF_NORMAL_CALL:
        pScratData->counter = pScratData->counter + 1;
        break;
      case SQLUDF_FINAL_CALL:
        break;
    }
    *outCounter = pScratData->counter;
    *counterNullInd = 0;
  } /* ScratchpadScUDF */
```

The SQLUDF_TRAIL_ARGS_ALL macro expands to define other parameter values including one called SQLUDF_SCRAT that defines a buffer parameter to be used as a scratchpad. When the scalar function is invoked for a set of values, for each time the scalar function is invoked, the buffer is passed as a parameter to the function. The buffer can be used to be accessed

The SQLUDF_TRAIL_ARGS_ALL macro value also defines another parameter SQLUDF_CALLT. This parameter is used to indicate a call type value. Call type values can be used to identify if a function is being invoked for the first time for a set of values, the last time, or at a time in the middle of the processing.

**Program type MAIN support for C and C++ procedures:** Although the default PROGRAM TYPE clause value SUB is generally recommended for C procedures, the PROGRAM TYPE clause value MAIN is supported in CREATE PROCEDURE statements where the LANGUAGE clause value is C.

The PROGRAM TYPE clause value MAIN is required for routines with greater than ninety parameters.

When a PROGRAM TYPE MAIN clause is specified, procedures must be implemented using a signature that is consistent with the default style for a main routine in a C source code file. This does not mean that the routine must be implemented by a function named `main`, but rather that the parameters be passed in the format generaly associated with a default type main routine application implementation that uses typical C programming `argc` and `argv` arguments.

Here is an example of a C or C++ routine signature that adheres to the PGRAM TYPE MAIN specification:

```
SQL_API_RC SQL_API_FN functionName(int argc, char **argv)
{
  ...
}
```

The total number of arguments to the function is specified by the value of argc. The argument values are passed as array elements within the argv array. The number and order of the arguments depends on the PARAMETER STYLE clause value specified in the CREATE PROCEDURE statement.

As an example, consider the following CREATE PROCEDURE statement for a C procedure specified to have a PROGRAM TYPE MAIN style and the recommended PARAMETER STYLE SQL:

```
CREATE PROCEDURE MAIN_EXAMPLE (
  IN job CHAR(8),
  OUT salary DOUBLE)
SPECIFIC CPP_MAIN_EXAMPLE
DYNAMIC RESULT SETS 0
NOT DETERMINISTIC
LANGUAGE C
PARAMETER STYLE SQL
NO DBINFO
FENCED NOT THREADSAFE
READS SQL DATA
PROGRAM TYPE MAIN
EXTERNAL NAME 'spserver!MainExample'@
```

The routine signature implementation that corresponds to this CREATE PROCEDURE statement follows:

```
//****************************************************
//  Stored Procedure: MainExample
//
//  SQL parameters:
//     IN:      argv[1] - job    (char[8])
//     OUT:     argv[2] - salary (double)
//****************************************************
SQL_API_RC SQL_API_FN MainExample(int argc, char **argv)
{
  ...
}
```

Because PARAMETER STYLE SQL is used, in addition to the SQL parameter values passed at procedure invocation time, the additional parameters required for that style are also passed to the routine.

Parameter values can be accessed by referencing the argv array element of interest within the source code. For the example given above, the argc and the argv array elements contain the following values:

```
    argc   : Number of argv array elements
    argv[0]: The function name
    argv[1]: Value of parameter job (char[8], input)
    argv[2]: Value of parameter salary (double, output)
    argv[3]: null indicator for parameter job
    argv[4]: null indicator for parameter salary
    argv[5]: sqlstate (char[6], output)
    argv[6]: qualName (char[28], output)
    argv[7]: specName (char[19], output)
    argv[8]: diagMsg (char[71], output)
```

## Supported SQL data types in C and C++ routines

The following table lists the supported mappings between SQL data types and C data types for routines. Accompanying each C/C++ data type is the corresponding defined type from sqludf.h.

*Table 28. SQL Data Types Mapped to C/C++ Declarations*

| SQL Column Type | C/C++ Data Type | SQL Column Type Description |
|---|---|---|
| SMALLINT | sqlint16<br>SQLUDF_SMALLINT | 16-bit signed integer |
| INTEGER | sqlint32<br>SQLUDF_INTEGER | 32-bit signed integer |
| BIGINT | sqlint64<br>SQLUDF_BIGINT | 64-bit signed integer |
| REAL<br>FLOAT(*n*) where 1<=n<=24 | float<br>SQLUDF_REAL | Single-precision floating point |
| DOUBLE<br>FLOAT<br>FLOAT(*n*) where 25<=n<=53 | double<br>SQLUDF_DOUBLE | Double-precision floating point |
| DECIMAL(*p*, *s*) | Not supported. | To pass a decimal value, define the parameter to be of a data type castable from DECIMAL (for example CHAR or DOUBLE) and explicitly cast the argument to this type. |

*Table 28. SQL Data Types Mapped to C/C++ Declarations (continued)*

| SQL Column Type | C/C++ Data Type | SQL Column Type Description |
|---|---|---|
| CHAR(*n*) | char[*n+1*] where n is large enough to hold the data<br><br>1<=*n*<=254<br><br>SQLUDF_CHAR | Fixed-length, null-terminated character string |
| CHAR(*n*) FOR BIT DATA | char[*n*] where n is large enough to hold the data<br><br>1<=*n*<=254<br><br>SQLUDF_CHAR | Fixed-length, not null-terminated character string |
| VARCHAR(*n*) | char[*n+1*] where n is large enough to hold the data<br><br>1<=*n*<=32 672<br><br>SQLUDF_VARCHAR | Null-terminated varying length string |
| VARCHAR(*n*) FOR BIT DATA | struct {<br>  sqluint16 length;<br>  char[*n*]<br>}<br><br>1<=*n*<=32 672<br><br>SQLUDF_VARCHAR_FBD | Not null-terminated varying length character string |
| LONG VARCHAR | struct {<br>  sqluint16 length;<br>  char[*n*]<br>}<br><br>1<=*n*<=32 700<br><br>SQLUDF_LONG | Not null-terminated varying length character string |
| CLOB(*n*) | struct {<br>  sqluint32 length;<br>  char data[*n*];<br>}<br><br>1<=*n*<=2 147 483 647<br><br>SQLUDF_CLOB | Not null-terminated varying length character string with 4-byte string length indicator |
| BLOB(*n*) | struct {<br>  sqluint32 length;<br>  char data[n];<br>}<br><br>1<=*n*<=2 147 483 647<br><br>SQLUDF_BLOB | Not null-terminated varying binary string with 4-byte string length indicator |
| DATE | char[11]<br>SQLUDF_DATE | Null-terminated character string of the following format:<br>`yyyy-mm-dd` |
| TIME | char[9]<br>SQLUDF_TIME | Null-terminated character string of the following format:<br>`hh.mm.ss` |
| TIMESTAMP | char[27]<br>SQLUDF_STAMP | Null-terminated character string of the following format:<br>`yyyy-mm-dd-hh.mm.ss.nnnnnn` |

*Table 28. SQL Data Types Mapped to C/C++ Declarations (continued)*

| SQL Column Type | C/C++ Data Type | SQL Column Type Description |
|---|---|---|
| LOB LOCATOR | sqluint32<br>SQLUDF_LOCATOR | 32-bit signed integer |
| DATALINK | `struct {`<br>`   sqluint32  version;`<br>`   char       linktype[4];`<br>`   sqluint32  url_length;`<br>`   sqluint32  comment_length;`<br>`   char       reserve2[8];`<br>`   char       url_plus_comment[230];`<br>`}`<br><br>SQLUDF_DATALINK | |
| GRAPHIC(*n*) | sqldbchar[*n+1*] where n is large enough to hold the data<br><br>1<=*n*<=127<br><br>SQLUDF_GRAPH | Fixed-length, null-terminated double-byte character string |
| VARGRAPHIC(*n*) | sqldbchar[*n+1*] where n is large enough to hold the data<br><br>1<=*n*<=16 336<br><br>SQLUDF_GRAPH | Null-terminated, variable-length double-byte character string |
| LONG VARGRAPHIC | `struct {`<br>`   sqluint16  length;`<br>`   sqldbchar[n]`<br>`}`<br><br>1<=*n*<=16 350<br><br>SQLUDF_LONGVARG | Not null-terminated, variable-length double-byte character string |
| DBCLOB(*n*) | `struct {`<br>`   sqluint32   length;`<br>`   sqldbchar  data[n];`<br>`}`<br><br>1<=*n*<=1 073 741 823<br><br>SQLUDF_DBCLOB | Not null-terminated varying length character string with 4-byte string length indicator |
| XML AS CLOB | `struct {`<br>`   sqluint32  length;`<br>`   char       data[n];`<br>`}`<br><br>1<=*n*<=2 147 483 647<br><br>SQLUDF_CLOB | Not null-terminated varying length serialized character string with 4-byte string length indicator. |

**Note:** XML data types can only be implemented as CLOB data types in external routines implemented in C or C++.

**Note:** The following data types are only available in the DBCS or EUC environment when precompiled with the WCHARTYPE NOCONVERT option:

- GRAPHIC(*n*)
- VARGRAPHIC(*n*)
- LONG VARGRAPHIC

- DBCLOB(*n*)

**Related concepts:**

- "Include file required for C and C++ routine development (sqludf.h)" on page 287
- "Parameters in C and C++ routines" on page 288
- "SQL data type handling in C and C++ routines" on page 303

**Related tasks:**

- "Designing C and C++ routines" on page 286

## SQL data type handling in C and C++ routines

This section identifies the valid types for routine parameters and results, and it specifies how the corresponding argument should be defined in your C or C++ language routine. All arguments in the routine must be passed as pointers to the appropriate data type. Note that if you use the `sqludf.h` include file and the types defined there, you can automatically generate language variables and structures that are correct for the different data types and compilers. For example, for BIGINT you can use the SQLUDF_BIGINT data type to hide differences in the type required for BIGINT representation between different compilers.

It is the data type for each parameter defined in the routine's CREATE statement that governs the format for argument values. Promotions from the argument's data type might be needed to get the value in the appropriate format. Such promotions are performed automatically by DB2 on argument values. However, if incorrect data types are specified in the routine code, then unpredictable behavior, such as loss of data or abends, will occur.

For the result of a scalar function or method, it is the data type specified in the CAST FROM clause of the CREATE FUNCTION statement that defines the format. If no CAST FROM clause is present, then the data type specified in the RETURNS clause defines the format.

In the following example, the presence of the CAST FROM clause means that the routine body returns a SMALLINT and that DB2 casts the value to INTEGER before passing it along to the statement where the function reference occurs:

```
... RETURNS INTEGER CAST FROM SMALLINT ...
```

In this case, the routine must be written to generate a SMALLINT, as defined later in this section. Note that the CAST FROM data type must be *castable* to the RETURNS data type, therefore, it is not possible to arbitrarily choose another data type.

The following is a list of the SQL types and their C/C++ language representations. It includes information on whether each type is valid as a parameter or a result. Also included are examples of how the types could appear as an argument definition in your C or C++ language routine:

- SMALLINT

  **Valid**. Represent in C as `SQLUDF_SMALLINT` or `sqlint16`.

  Example:

  ```
  sqlint16    *arg1;          /* example for SMALLINT */
  ```

When defining integer routine parameters, consider using INTEGER rather than SMALLINT because DB2 does not promote INTEGER arguments to SMALLINT. For example, suppose you define a UDF as follows:

```
CREATE FUNCTION SIMPLE(SMALLINT)...
```

If you invoke the SIMPLE function using INTEGER data, (`... SIMPLE(1)...`), you will receive an SQLCODE -440 (SQLSTATE 42884) error indicating that the function was not found, and end-users of this function might not perceive the reason for the message. In the preceding example, 1 is an INTEGER, so you can either cast it to SMALLINT or define the parameter as INTEGER.

- INTEGER or INT

  **Valid**. Represent in C as `SQLUDF_INTEGER` or `sqlint32`. You must `#include sqludf.h` or `#include sqlsystm.h` to pick up this definition.

  Example:

  ```
  sqlint32 *arg2;          /* example for INTEGER */
  ```

- BIGINT

  **Valid**. Represent in C as `SQLUDF_BIGINT` or `sqlint64`.

  Example:

  ```
  sqlint64 *arg3;          /* example for INTEGER */
  ```

  DB2 defines the `sqlint64` C language type to overcome differences between definitions of the 64-bit signed integer in compilers and operating systems. You must `#include sqludf.h` or `#include sqlsystm.h` to pick up the definition.

- REAL or FLOAT(*n*) where 1 <= *n* <= 24

  **Valid**. Represent in C as `SQLUDF_REAL` or `float`.

  Example:

  ```
  float *result;          /* example for REAL */
  ```

- DOUBLE or DOUBLE PRECISION or FLOAT or FLOAT(*n*) where 25 <= *n* <= 53

  **Valid**. Represent in C as `SQLUDF_DOUBLE` or `double`.

  Example:

  ```
  double  *result;        /* example for DOUBLE   */
  ```

- DECIMAL(p,s) or NUMERIC(p,s)

  **Not valid** because there is no C language representation. If you want to pass a decimal value, you must define the parameter to be of a data type castable from DECIMAL (for example CHAR or DOUBLE) and explicitly cast the argument to this type. In the case of DOUBLE, you do not need to explicitly cast a decimal argument to a DOUBLE parameter, because DB2 promotes it automatically.

  Example:

  Suppose you have two columns, WAGE as DECIMAL(5,2) and HOURS as DECIMAL(4,1), and you wish to write a UDF to calculate weekly pay based on wage, number of hours worked and some other factors. The UDF could be as follows:

  ```
  CREATE FUNCTION WEEKLY_PAY (DOUBLE, DOUBLE, ...)
         RETURNS DECIMAL(7,2) CAST FROM DOUBLE
         ...;
  ```

  For the preceding UDF, the first two parameters correspond to the wage and number of hours. You invoke the UDF WEEKLY_PAY in your SQL select statement as follows:

  ```
  SELECT WEEKLY_PAY (WAGE, HOURS, ...) ...;
  ```

  Note that no explicit casting is required because the DECIMAL arguments are castable to DOUBLE.

Alternatively, you could define WEEKLY_PAY with CHAR arguments as follows:

```
CREATE FUNCTION WEEKLY_PAY (VARCHAR(6), VARCHAR(5), ...)
      RETURNS DECIMAL (7,2) CAST FROM VARCHAR(10)
      ...;
```

You would invoke it as follows:

```
SELECT WEEKLY_PAY (CHAR(WAGE), CHAR(HOURS), ...) ...;
```

Observe that explicit casting is required because DECIMAL arguments are not promotable to VARCHAR.

An advantage of using floating point parameters is that it is easy to perform arithmetic on the values in the routine; an advantage of using character parameters is that it is always possible to exactly represent the decimal value. This is not always possible with floating point.

- CHAR(n) or CHARACTER(n) with or without the FOR BIT DATA modifier.

  **Valid**. Represent in C as `SQLUDF_CHAR` or `char...[n+1]` (this is a C null-terminated string).

  Example:

  ```
  char     arg1[14];      /* example for CHAR(13)   */
  char    *arg1;          /* also acceptable */
  ```

  Input routine parameters of data type CHAR are always automatically null terminated. For a CHAR(n) input parameter, where *n* is the length of the CHAR data type, *n* bytes of data are moved to the buffer in the routine implementation and the character in the *n + 1* position is set to the ASCII null terminator character (X'00').

  Output parameters of procedures and return values of functions of data type CHAR must be explicitly null terminated by the routine. For a return value of a UDF specified by the RETURNS clause, such as RETURNS CHAR(n), or a procedure output parameter specified as CHAR(n), where n is the length of the CHAR value, a null terminator character must exist within the first *n+1* bytes of the buffer. If a null terminator is found within the first *n+1* bytes of the buffer, the remaining bytes, up to byte *n*, are set to ASCII blank characters (X'20'). If no null terminator is found, an SQL error (SQLSTATE 39501) results.

  For input and output parameters of procedures or function return values of data type CHAR that also specify the FOR BIT DATA clause, which indicates that the data is to be manipulated in its binary form, null terminators are not used to indicate the end of the parameter value. For either a RETURNS CHAR(*n*) FOR BIT DATA function return value or a CHAR(*n*) FOR BIT DATA output parameter, the first *n* bytes of the buffer are copied over regardless of any occurrences of string null terminators within the first *n* bytes. Null terminator characters identified within the buffer are ignored as null terminators and instead are simply treated as normal data.

  Exercise caution when using the normal C string handling functions in a routine that manipulates a FOR BIT DATA value, because many of these functions look for a null terminator to delimit a string argument and null terminators (X'00') can legitimately appear in the middle of a FOR BIT DATA value. Using the C functions on FOR BIT DATA values might cause the undesired truncation of the data value.

  When defining character routine parameters, consider using VARCHAR rather than CHAR as DB2 does not promote VARCHAR arguments to CHAR and string literals are automatically considered as VARCHARs. For example, suppose you define a UDF as follows:

  ```
  CREATE FUNCTION SIMPLE(INT,CHAR(1))...
  ```

  If you invoke the SIMPLE function using VARCHAR data, (`... SIMPLE(1,'A')...`), you will receive an SQLCODE -440 (SQLSTATE 42884) error indicating that the function was not found, and end-users of this function might not perceive the reason for the message. In the preceding example, 'A' is VARCHAR, so you can either cast it to CHAR or define the parameter as VARCHAR.

- VARCHAR(n) FOR BIT DATA or LONG VARCHAR with or without the FOR BIT DATA modifier.

  **Valid**. Represent VARCHAR(n) FOR BIT DATA in C as SQLUDF_VARCHAR_FBD. Represent LONG VARCHAR in C as SQLUDF_LONG. Otherwise represent these two SQL types in C as a structure similar to the following from the sqludf.h include file:

  ```
  struct sqludf_vc_fbd
  {
     unsigned short length;         /* length of data */
     char           data[1];        /* first char of data */
  };
  ```

  The [1] indicates an array to the compiler. It does not mean that only one character is passed; because the address of the structure is passed, and not the actual structure, it provides a way to use array logic.

  These values are not represented as C null-terminated strings because the null-character could legitimately be part of the data value. The length is explicitly passed to the routine for parameters using the structure variable length. For the RETURNS clause, the length that is passed to the routine is the length of the buffer. What the routine body must pass back, using the structure variable length, is the actual length of the data value.

  Example:

  ```
  struct sqludf_vc_fbd *arg1;  /* example for VARCHAR(n) FOR BIT DATA */
  struct sqludf_vc_fbd *result; /* also for LONG VARCHAR FOR BIT DATA */
  ```

- VARCHAR(n) without FOR BIT DATA.

  **Valid**. Represent in C as SQLUDF_VARCHAR or char...[n+1]. (This is a C null-terminated string.)

  For a VARCHAR(n) parameter, DB2 will put a null in the (k+1) position, where k is the length of the particular string. The C string-handling functions are well suited for manipulation of these values. For a RETURNS VARCHAR(n) value or an output parameter of a stored procedure, the routine body must delimit the actual value with a null because DB2 will determine the result length from this null character.

  Example:

  ```
  char     arg2[51];      /* example for VARCHAR(50)   */
  char    *result;        /* also acceptable */
  ```

- DATE

  **Valid**. Represent in C same as SQLUDF_DATE or CHAR(10), that is as char...[11]. The date value is always passed to the routine in ISO format:

  yyyy-mm-dd

  Example:

  ```
  char     arg1[11];      /* example for DATE       */
  char    *result;        /* also acceptable */
  ```

  **Note:** For DATE, TIME and TIMESTAMP return values, DB2 demands the characters be in the defined form, and if this is not the case the value could be misinterpreted by DB2 (For example, 2001-04-03 will be interpreted as April 3 even if March 4 is intended) or will cause an error (SQLCODE -493, SQLSTATE 22007).

- TIME

**Valid**. Represent in C same as SQLUDF_TIME or CHAR(8), that is, as char...[9]. The time value is always passed to the routine in ISO format:

```
hh.mm.ss
```

Example:

```
char    *arg;          /* example for TIME          */
char     result[9];    /* also acceptable */
```

- TIMESTAMP

  **Valid**. Represent in C as SQLUDF_STAMP or CHAR(26), that is, as char...[27]. The timestamp value is always passed with format:

```
yyyy-mm-dd-hh.mm.ss.nnnnnn
```

  Example:

```
char     arg1[27];     /* example for TIMESTAMP  */
char    *result;       /* also acceptable */
```

- GRAPHIC(n)

  **Valid**. Represent in C as SQLUDF_GRAPH or sqldbchar[n+1]. (This is a null-terminated graphic string). Note that you can use wchar_t[n+1] on operating systems where wchar_t is defined to be 2 bytes in length; however, sqldbchar is recommended.

  For a GRAPHIC(n) parameter, DB2 moves *n* double-byte characters to the buffer and sets the following two bytes to null. Data passed from DB2 to a routine is in DBCS format, and the result passed back is expected to be in DBCS format. This behavior is the same as using the WCHARTYPE NOCONVERT precompiler option. For a RETURNS GRAPHIC(*n*) value or an output parameter of a stored procedure, DB2 looks for an embedded GRAPHIC null CHAR, and if it finds it, pads the value out to *n* with GRAPHIC blank characters.

  When defining graphic routine parameters, consider using VARGRAPHIC rather than GRAPHIC as DB2 does not promote VARGRAPHIC arguments to GRAPHIC. For example, suppose you define a routine as follows:

```
CREATE FUNCTION SIMPLE(GRAPHIC)...
```

  If you invoke the SIMPLE function using VARGRAPHIC data, (... SIMPLE('*graphic_literal*')...), you will receive an SQLCODE -440 (SQLSTATE 42884) error indicating that the function was not found, and end-users of this function might not understand the reason for this message. In the preceding example, *graphic_literal* is a literal DBCS string that is interpreted as VARGRAPHIC data, so you can either cast it to GRAPHIC or define the parameter as VARGRAPHIC.

  Example:

```
sqldbchar   arg1[14];      /* example for GRAPHIC(13)   */
sqldbchar  *arg1;          /* also acceptable */
```

- VARGRAPHIC(n)

  **Valid**. Represent in C as SQLUDF_GRAPH or sqldbchar[n+1]. (This is a null-terminated graphic string). Note that you can use wchar_t[n+1] on operating systems where wchar_t is defined to be 2 bytes in length; however, sqldbchar is recommended.

  For a VARGRAPHIC(*n*) parameter, DB2 will put a graphic null in the (k+1) position, where *k* is the length of the particular occurrence. A graphic null refers to the situation where all the bytes of the last character of the graphic string contain binary zeros ('\0's). Data passed from DB2 to a routine is in DBCS format, and the result passed back is expected to be in DBCS format. This behavior is the same as using the WCHARTYPE NOCONVERT precompiler option. For a RETURNS VARGRAPHIC(*n*) value or an output parameter of a stored procedure, the routine body must delimit the actual value with a graphic null, because DB2 will determine the result length from this graphic null character.

Example:

```
sqldbchar   args[51],    /* example for VARGRAPHIC(50) */
sqldbchar   *result,     /* also acceptable  */
```

- LONG VARGRAPHIC

  **Valid**. Represent in C as `SQLUDF_LONGVARG` or a structure:

  ```
  struct sqludf_vg
  {
     unsigned short length;         /* length of data */
     sqldbchar      data[1];        /* first char of data */
  };
  ```

  Note that in the preceding structure, you can use `wchar_t` in place of `sqldbchar` on operating systems where `wchar_t` is defined to be 2 bytes in length, however, the use of `sqldbchar` is recommended.

  The [1] merely indicates an array to the compiler. It does not mean that only one graphic character is passed. Because the address of the structure is passed, and not the actual structure, it provides a way to use array logic.

  These are not represented as null-terminated graphic strings. The length, in double-byte characters, is explicitly passed to the routine for parameters using the structure variable `length`. Data passed from DB2 to a routine is in DBCS format, and the result passed back is expected to be in DBCS format. This behavior is the same as using the WCHARTYPE NOCONVERT precompiler option. For the RETURNS clause or an output parameter of a stored procedure, the length that is passed to the routine is the length of the buffer. What the routine body must pass back, using the structure variable `length`, is the actual length of the data value, in double byte characters.

  Example:

  ```
  struct sqludf_vg *arg1;  /* example for VARGRAPHIC(n)   */
  struct sqludf_vg *result; /* also for LONG VARGRAPHIC   */
  ```

- BLOB(n) and CLOB(n)

  **Valid**. Represent in C as `SQLUDF_BLOB`, `SQLUDF_CLOB`, or a structure:

  ```
  struct sqludf_lob
  {
      sqluint32     length;      /* length in bytes */
      char          data[1];      /* first byte of lob */
  };
  ```

  The [1] merely indicates an array to the compiler. It does not mean that only one character is passed; because the address of the structure is passed, and not the actual structure, it provides a way to use array logic.

  These are not represented as C null-terminated strings. The length is explicitly passed to the routine for parameters using the structure variable `length`. For the RETURNS clause or an output parameter of a stored procedure, the length that is passed back to the routine, is the length of the buffer. What the routine body must pass back, using the structure variable `length`, is the actual length of the data value.

  Example:

  ```
  struct sqludf_lob *arg1;  /* example for BLOB(n), CLOB(n) */
  struct sqludf_lob *result;
  ```

- DBCLOB(n)

  **Valid**. Represent in C as `SQLUDF_DBCLOB` or a structure:

```
struct sqludf_lob
{
   sqluint32 length;      /* length in graphic characters */
   sqldbchar data[1];         /* first byte of lob */
};
```

Note that in the preceding structure, you can use `wchar_t` in place of `sqldbchar` on operating systems where `wchar_t` is defined to be 2 bytes in length, however, the use of `sqldbchar` is recommended.

The [1] merely indicates an array to the compiler. It does not mean that only one graphic character is passed; because the address of the structure is passed, and not the actual structure, it provides a way to use array logic.

These are not represented as null-terminated graphic strings. The length is explicitly passed to the routine for parameters using the structure variable `length`. Data passed from DB2 to a routine is in DBCS format, and the result passed back is expected to be in DBCS format. This behavior is the same as using the WCHARTYPE NOCONVERT precompiler option. For the RETURNS clause or an output parameter of a stored procedure, the length that is passed to the routine is the length of the buffer. What the routine body must pass back, using the structure variable `length`, is the actual length of the data value, with all of these lengths expressed in double byte characters.

Example:

```
struct sqludf_lob *arg1;  /* example for DBCLOB(n) */
struct sqludf_lob *result;
```

- Distinct Types

  **Valid or invalid depending on the base type**. Distinct types will be passed to the UDF in the format of the base type of the UDT, so can be specified if and only if the base type is valid.

  Example:

  ```
  struct sqludf_lob *arg1;  /* for distinct type based on BLOB(n) */
  double            *arg2;  /* for distinct type based on DOUBLE  */
  char              res[5]; /* for distinct type based on CHAR(4) */
  ```

- XML

  **Valid**. Represent in C as `SQLUDF_XML` or in the way as a CLOB data type is represented; that is with a structure:

  ```
  struct sqludf_lob
  {
     sqluint32     length;      /* length in bytes */
     char          data[1];      /* first byte of lob */
  };
  ```

  The [1] merely indicates an array to the compiler. It does not mean that only one character is passed; because the address of the structure is passed, and not the actual structure, it provides a way to use array logic.

  These are not represented as C null-terminated strings. The length is explicitly passed to the routine for parameters using the structure variable `length`. For the RETURNS clause or an output parameter of a stored procedure, the length that is passed back to the routine, is the length of the buffer. What the routine body must pass back, using the structure variable `length`, is the actual length of the data value.

  Example:

  ```
  struct sqludf_lob *arg1;  /* example for XML(n) */
  struct sqludf_lob *result;
  ```

  The assignment and access of XML parameter and variable values in C and C++ external routine code is done in the same way as for CLOB values.

- Distinct Types AS LOCATOR, or any LOB type AS LOCATOR

**Valid for parameters and results of UDFs and methods.** It can only be used to modify LOB types or any distinct type that is based on a LOB type. Represent in C as SQLUDF_LOCATOR or a four byte integer.

The locator value can be assigned to any locator host variable with a compatible type and then be used in an SQL statement. This means that locator variables are only useful in UDFs and methods defined with an SQL access indicator of CONTAINS SQL or higher. For compatibility with existing UDFs and methods, the locator APIs are still supported for NOT FENCED NO SQL UDFs. Use of these APIs is not encouraged for new functions.

Example:

```
     sqludf_locator       *arg1;  /* locator argument */
     sqludf_locator       *result; /* locator result */

EXEC SQL BEGIN DECLARE SECTION;
   SQL TYPE IS CLOB LOCATOR arg_loc;
   SQL TYPE IS CLOB LOCATOR res_loc;
EXEC SQL END DECLARE SECTION;

/* Extract some characters from the middle */
/* of the argument and return them         */
*arg_loc = arg1;
EXEC SQL VALUES SUBSTR(arg_loc, 10, 20) INTO :res_loc;
*result = res_loc;
```

- Structured Types

  Valid for parameters and results of UDFs and methods where an appropriate transform function exists. Structured type parameters will be passed to the function or method in the result type of the FROM SQL transform function. Structured type results will be passed in the parameter type of the TO SQL transform function.
- DATALINK

  **Valid**. Represent in C as SQLUDF_DATALINK or a structure similar to the following from the sqludf.h include file:

```
     struct sqludf_datalink {
       sqluint32 version;
       char      linktype[4];
       sqluint32 url_length;
       sqluint32 comment_length;
       char      reserve2[8];
       char      url_plus_comment[230];
     }
```

**Related concepts:**

- "Graphic host variables in C and C++ routines" on page 322
- "Include file required for C and C++ routine development (sqludf.h)" on page 287
- "Transform functions and transform groups" in *Developing SQL and External Routines*

**Related reference:**

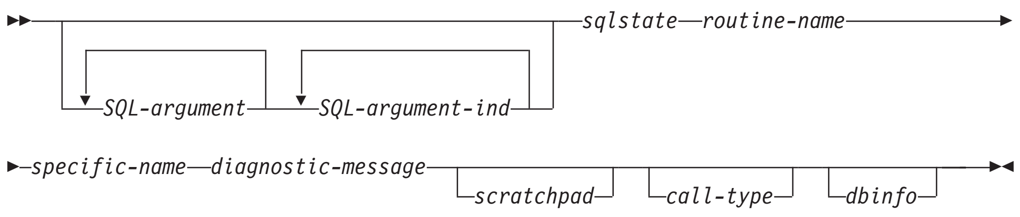
- "Supported SQL data types in C and C++ embedded SQL applications" on page 53
- "CREATE FUNCTION statement" in *SQL Reference, Volume 2*
- "CREATE PROCEDURE statement" in *SQL Reference, Volume 2*
- "Supported SQL data types in C and C++ routines" on page 300

## Passing arguments to C, C++, OLE, or COBOL routines

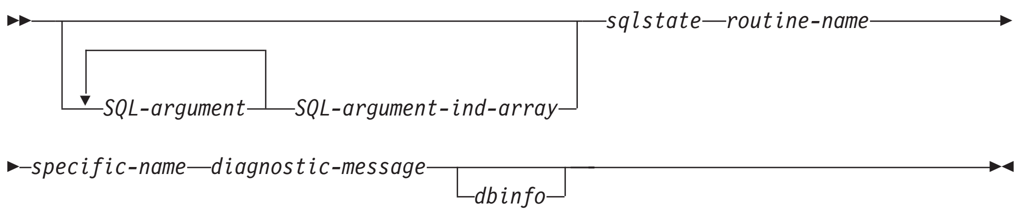
In addition to the SQL arguments that are specified in the DML reference for a routine, DB2 passes additional arguments to the external routine body. The nature and order of these arguments is determined by the parameter style with which you registered your routine. To ensure that information is exchanged correctly between invokers and the routine body, you must ensure that your routine accepts arguments in the order they are passed, according to the parameter style being used. The `sqludf` include file can aid you in handling and using these arguments.

The following parameter styles are applicable only to LANGUAGE C, LANGUAGE OLE, and LANGUAGE COBOL routines.

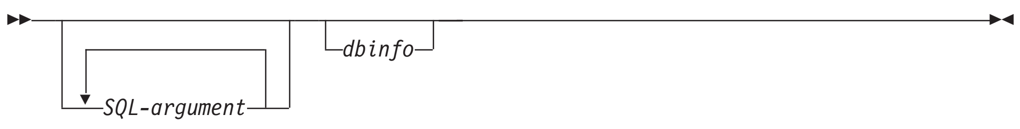
### PARAMETER STYLE SQL routines



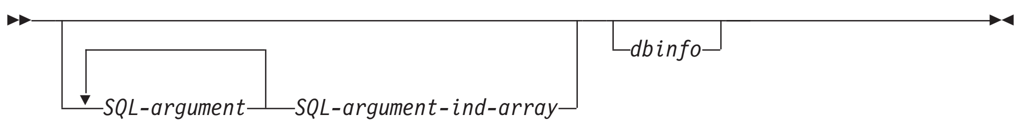
### PARAMETER STYLE DB2SQL procedures



### PARAMETER STYLE GENERAL procedures



### PARAMETER STYLE GENERAL WITH NULLS procedures



**Note:** For UDFs and methods, PARAMETER STYLE SQL is equivalent to PARAMETER STYLE DB2SQL.

The arguments for the above parameter styles are described as follows:

*SQL-argument...*
: Each *SQL-argument* represents one input or output value defined when the routine was created. The list of arguments is determined as follows:

- For a scalar function, one argument for each input parameter to the function followed by one *SQL-argument* for the result of the function.
- For a table function, one argument for each input parameter to the function followed by one *SQL-argument* for each column in the result table of the function.
- For a method, one *SQL-argument* for the subject type of the method, then one argument for each input parameter to the method followed by one *SQL-argument* for the result of the method.
- For a stored procedure, one *SQL-argument* for each parameter to the stored procedure.

Each *SQL-argument* is used as follows:

- Input parameter of a function or method, subject type of a method, or an IN parameter of a stored procedure

  This argument is set by DB2 before calling the routine. The value of each of these arguments is taken from the expression specified in the routine invocation. It is expressed in the data type of the corresponding parameter definition in the CREATE statement.
- Result of a function or method or an OUT parameter of a stored procedure

  This argument is set by the routine before returning to DB2. DB2 allocates the buffer and passes its address to the routine. The routine puts the result value into the buffer. Enough buffer space is allocated by DB2 to contain the value expressed in the data type. For character types and LOBs, this means the maximum size, as defined in the create statement, is allocated.

  For scalar functions and methods, the result data type is defined in the CAST FROM clause, if it is present, or in the RETURNS clause, if no CAST FROM clause is present.

  For table functions, DB2 defines a performance optimization where every defined column does not have to be returned to DB2. If you write your UDF to take advantage of this feature, it returns only the columns required by the statement referencing the table function. For example, consider a CREATE FUNCTION statement for a table function defined with 100 result columns. If a given statement referencing the function is only interested in two of them, this optimization enables the UDF to return only those two columns for each row and not spend time on the other 98 columns. See the dbinfo argument below for more information on this optimization.

  For each value returned, the routine should not return more bytes than is required for the data type and length of the result. Maximums are defined during the creation of the routine's catalog entry. An overwrite by the routine can cause unpredictable results or an abnormal termination.
- INOUT parameter of a stored procedure

  This argument behaves as both an IN and an OUT parameter and therefore follows both sets of rules shown above. DB2 will set the argument before calling the stored procedure. The buffer allocated by DB2 for the argument is large enough to contain the maximum size of the data type of the parameter defined in the CREATE PROCEDURE statement. For example, an INOUT parameter of a CHAR type could have a 10 byte varchar going in to the stored procedure, and a 100 byte varchar coming out of the stored procedure. The buffer is set by the stored procedure before returning to DB2.

DB2 aligns the data for *SQL-argument* according to the data type and the server operating system, also known as platform.

*SQL-argument-ind...*

There is an *SQL-argument-ind* for each *SQL-argument* passed to the routine. The *n*th *SQL-argument-ind* corresponds to the *n*th *SQL-argument* and indicates whether the *SQL-argument* has a value or is NULL.

Each *SQL-argument-ind* is used as follows:

- Input parameter of a function or method, subject type of a method, or an IN parameter of a stored procedure

  This argument is set by DB2 before calling the routine. It contains one of the following values:

  **0** The argument is present and not NULL.

  **-1** The argument is present and its value is NULL.

  If the routine is defined with RETURNS NULL ON NULL INPUT, the routine body does not need to check for a NULL value. However, if it is defined with CALLED ON NULL INPUT, any argument can be NULL and the routine should check *SQL-argument-ind* before using the corresponding *SQL-argument*.

- Result of a function or method or an OUT parameter of a stored procedure

  This argument is set by the routine before returning to DB2. This argument is used by the routine to signal if the particular result value is NULL:

  **0** The result is not NULL.

  **-1** The result is the NULL value.

  Even if the routine is defined with RETURNS NULL ON NULL INPUT, the routine body must set the *SQL-argument-ind* of the result. For example, a divide function could set the result to null when the denominator is zero.

  For scalar functions and methods, DB2 treats a NULL result as an arithmetic error if the following is true:

  – The database configuration parameter *dft_sqlmathwarn* is YES

  – One of the input arguments is a null because of an arithmetic error

  This is also true if you define the function with the RETURNS NULL ON NULL INPUT option

  For table functions, if the UDF takes advantage of the optimization using the result column list, then only the indicators corresponding to the required columns need be set.

- INOUT parameter of a stored procedure

  This argument behaves as both an IN and an OUT parameter and therefore follows both sets of rules shown above. DB2 will set the argument before calling the stored procedure. The *SQL-argument-ind* is set by the stored procedure before returning to DB2.

Each *SQL-argument-ind* takes the form of a SMALLINT value. DB2 aligns the data for *SQL-argument-ind* according to the data type and the server operating system.

*SQL-argument-ind-array*

There is an element in *SQL-argument-ind-array* for each SQL-argument passed to the stored procedure. The *n*th element in *SQL-argument-ind-array* corresponds to the *n*th SQL-argument and indicates whether the *SQL-argument* has a value or is NULL

Each element in *SQL-argument-ind-array* is used as follows:

- IN parameter of a stored procedure

  This element is set by DB2 before calling the routine. It contains one of the following values:

  **0** The argument is present and not NULL.

  **-1** The argument is present and its value is NULL.

  If the stored procedure is defined with RETURNS NULL ON NULL INPUT, the stored procedure body does not need to check for a NULL value. However, if it is defined with CALLED ON NULL INPUT, any argument can be NULL and the stored procedure should check *SQL-argument-ind* before using the corresponding *SQL-argument*.

- OUT parameter of a stored procedure

  This element is set by the routine before returning to DB2. This argument is used by the routine to signal if the particular result value is NULL:

  **0 or positive**
  The result is not NULL.

  **negative**
  The result is the NULL value.

- INOUT parameter of a stored procedure

  This element behaves as both an IN and an OUT parameter and therefore follows both sets of rules shown above. DB2 will set the argument before calling the stored procedure. The element of *SQL-argument-ind-array* is set by the stored procedure before returning to DB2.

Each element of *SQL-argument-ind-array* takes the form of a SMALLINT value. DB2 aligns the data for *SQL-argument-ind-array* according to the data type and the server operating system.

*sqlstate* This argument is set by the routine before returning to DB2. It can be used by the routine to signal warning or error conditions. The routine can set this argument to any value. The value '00000' means that no warning or error situations were detected. Values that start with '01' are warning conditions. Values that start with anything other than '00' or '01' are error conditions. When the routine is called, the argument contains the value '00000'.

For error conditions, the routine returns an SQLCODE of -443. For warning conditions, the routine returns an SQLCODE of +462. If the SQLSTATE is 38001 or 38502, then the SQLCODE is -487.

The *sqlstate* takes the form of a CHAR(5) value. DB2 aligns the data for *sqlstate* according to the data type and the server operating system.

*routine-name*

This argument is set by DB2 before calling the routine. It is the qualified function name, passed from DB2 to the routine

The form of the *routine-name* that is passed is:

```
schema.routine
```

The parts are separated by a period. Two examples are:

```
PABLO.BLOOP   WILLIE.FINDSTRING
```

This form enables you to use the same routine body for multiple external routines, and still differentiate between the routines when it is invoked.

**Note:** Although it is possible to include the period in object names and schema names, it is not recommended. For example, if a function, `ROTATE` is in a schema, `OBJ.OP`, the routine name that is passed to the function is `OBJ.OP.ROTATE`, and it is not obvious if the schema name is `OBJ` or `OBJ.OP`.

The *routine-name* takes the form of a VARCHAR(257) value. DB2 aligns the data for *routine-name* according to the data type and the server operating system.

*specific-name*

This argument is set by DB2 before calling the routine. It is the specific name of the routine passed from DB2 to the routine.

Two examples are:

```
WILLIE_FIND_FEB99   SQL9904281052440430
```

This first value is provided by the user in his CREATE statement. The second value is generated by DB2 from the current timestamp when the user does not specify a value.

As with the *routine-name* argument, the reason for passing this value is to give the routine the means of distinguishing exactly which specific routine is invoking it.

The *specific-name* takes the form of a VARCHAR(18) value. DB2 aligns the data for *specific-name* according to the data type and the server operating system.

*diagnostic-message*

This argument is set by the routine before returning to DB2. The routine can use this argument to insert message text in a DB2 message.

When the routine returns either an error or a warning, using the *sqlstate* argument described previously, it can include descriptive information here. DB2 includes this information as a token in its message.

DB2 sets the first character to null before calling the routine. Upon return, it treats the string as a C null-terminated string. This string will be included in the SQLCA as a token for the error condition. At least the first part of this string will appear in the SQLCA or DB2 CLP message. However, the actual number of characters that will appear depends on the lengths of the other tokens, because DB2 truncates the tokens to conform to the limit on total token length imposed by the SQLCA. Avoid using X'FF' in the text since this character is used to delimit tokens in the SQLCA.

The routine should not return more text than will fit in the VARCHAR(70) buffer that is passed to it. An overwrite by the routine can cause unpredictable results or an abend.

DB2 assumes that any message tokens returned from the routine to DB2 are in the same code page as the routine. Your routine should ensure that this is the case. If you use the 7-bit invariant ASCII subset, your routine can return the message tokens in any code page.

The *diagnostic-message* takes the form of a VARCHAR(70) value. DB2 aligns the data for *diagnostic-message* according to the data type and the server operating system.

*scratchpad*

This argument is set by DB2 before invoking the UDF or method. It is only present for functions and methods that specified the SCRATCHPAD keyword during registration. This argument is a structure, exactly like the structure used to pass a value of any of the LOB data types, with the following elements:

- An INTEGER containing the length of the scratchpad. Changing the length of the scratchpad will result in SQLCODE -450 (SQLSTATE 39501)
- The actual scratchpad initialized to all binary 0s as follows:
  - For scalar functions and methods, it is initialized before the first call, and not generally looked at or modified by DB2 thereafter.
  - For table functions, the scratchpad is initialized prior to the FIRST call to the UDF if FINAL CALL is specified on the CREATE FUNCTION. After this call, the scratchpad content is totally under control of the table function. If NO FINAL CALL was specified or defaulted for a table function, then the scratchpad is initialized for each OPEN call, and the scratchpad content is completely under control of the table function between OPEN calls. (This can be very important for a table function used in a join or subquery. If it is necessary to maintain the content of the scratchpad across OPEN calls, then FINAL CALL must be specified in your CREATE FUNCTION statement. With FINAL CALL specified, in addition to the normal OPEN, FETCH and CLOSE calls, the table function will also receive FIRST and FINAL calls, for the purpose of scratchpad maintenance and resource release.)

The scratchpad can be mapped in your routine using the same type as either a CLOB or a BLOB, since the argument passed has the same structure.

Ensure your routine code does not make changes outside of the scratchpad buffer. An overwrite by the routine can cause unpredictable results, an abend, and might not result in a graceful failure by DB2.

If a scalar UDF or method that uses a scratchpad is referenced in a subquery, DB2 might decide to refresh the scratchpad between invocations of the subquery. This refresh occurs after a final-call is made, if FINAL CALL is specified for the UDF.

DB2 initializes the scratchpad so that the data field is aligned for the storage of any data type. This can result in the entire scratchpad structure, including the length field, being improperly aligned.

*call-type*

This argument, if present, is set by DB2 before invoking the UDF or method. This argument is present for all table functions and for scalar functions and methods that specified FINAL CALL during registration

All the current possible values for *call-type* follow. Your UDF or method should contain a switch or case statement that explicitly tests for all the expected values, rather than containing “if A do AA, else if B do BB, else it must be C so do CC” type logic. This is because it is possible that additional call types will be added in the future, and if you do not explicitly test for condition C you will have trouble when new possibilities are added.

**Notes:**

1. For all values of *call-type*, it might be appropriate for the routine to set a *sqlstate* and *diagnostic-message* return value. This information will not be repeated in the following descriptions of each *call-type*. For all calls DB2 will take the indicated action as described previously for these arguments.
2. The include file `sqludf.h` is intended for use with routines. The file contains symbolic defines for the following *call-type* values, which are spelled out as constants.

For scalar functions and methods *call-type* contains:

**SQLUDF_FIRST_CALL (-1)**
This is the FIRST call to the routine for this statement. The *scratchpad* (if any) is set to binary zeros when the routine is called. All argument values are passed, and the routine should do whatever one-time initialization actions are required. In addition, a FIRST call to a scalar UDF or method is like a NORMAL call, in that it is expected to develop and return an answer.

**Note:** If SCRATCHPAD is specified but FINAL CALL is not, then the routine will not have this *call-type* argument to identify the very first call. Instead, it will have to rely on the all-zero state of the scratchpad.

**SQLUDF_NORMAL_CALL (0)**
This is a NORMAL call. All the SQL input values are passed, and the routine is expected to develop and return the result. The routine can also return *sqlstate* and *diagnostic-message* information.

**SQLUDF_FINAL_CALL (1)**
This is a FINAL call, that is no *SQL-argument* or *SQL-argument-ind* values are passed, and attempts to examine these values can cause unpredictable results. If a *scratchpad* is also passed, it is untouched from the previous call. The routine is expected to release resources at this point.

**SQLUDF_FINAL_CRA (255)**
This is a FINAL call, identical to the FINAL call described previously, with one additional characteristic, namely that it is made to routines that are defined as being able to issue SQL, and it is made at such a time that the routine must not issue any SQL except CLOSE cursor. (SQLCODE -396, SQLSTATE 38505) For example, when DB2 is in the middle of COMMIT processing, it can not tolerate new SQL, and any FINAL call issued to a routine at that time would be a 255 FINAL call. Routines that are not defined as containing any level of SQL access will never receive a 255 FINAL call, whereas routines that do use SQL might be given either type of FINAL call.

*Releasing resources*

A scalar UDF or method is expected to release resources it has required, for example, memory. If FINAL CALL is specified for the routine, then that FINAL call is a natural place to release resources, provided that SCRATCHPAD is also specified and is used to track the resource. If FINAL CALL is not specified, then any resource acquired should be released on the same call.

For table functions *call-type* contains:

**SQLUDF_TF_FIRST (-2)**
: This is the FIRST call, which only occurs if the FINAL CALL keyword was specified for the UDF. The *scratchpad* is set to binary zeros before this call. Argument values are passed to the table function. The table function can acquire memory or perform other one-time only resource initialization. This is not an OPEN call, that call follows this one. On a FIRST call the table function should not return any data to DB2 as DB2 ignores the data.

**SQLUDF_TF_OPEN (-1)**
: This is the OPEN call. The *scratchpad* will be initialized if NO FINAL CALL is specified, but not necessarily otherwise. All SQL argument values are passed to the table function on OPEN. The table function should not return any data to DB2 on the OPEN call.

**SQLUDF_TF_FETCH (0)**
: This is a FETCH call, and DB2 expects the table function to return either a row comprising the set of return values, or an end-of-table condition indicated by SQLSTATE value '02000'. If *scratchpad* is passed to the UDF, then on entry it is untouched from the previous call.

**SQLUDF_TF_CLOSE (1)**
: This is a CLOSE call to the table function. It balances the OPEN call, and can be used to perform any external CLOSE processing (for example, closing a source file), and resource release (particularly for the NO FINAL CALL case).

  In cases involving a join or a subquery, the OPEN/FETCH.../CLOSE call sequences can repeat within the execution of a statement, but there is only one FIRST call and only one FINAL call. The FIRST and FINAL call only occur if FINAL CALL is specified for the table function.

**SQLUDF_TF_FINAL (2)**
: This is a FINAL call, which only occurs if FINAL CALL was specified for the table function. It balances the FIRST call, and occurs only once per execution of the statement. It is intended for the purpose of releasing resources.

**SQLUDF_TF_FINAL_CRA (255)**
: This is a FINAL call, identical to the FINAL call described above, with one additional characteristic, namely that it is made to UDFs which are defined as being able to issue SQL, and it is made at such a time that the UDF must not issue any SQL except CLOSE cursor. (SQLCODE -396,

SQLSTATE 38505) For example, when DB2 is in the middle of COMMIT processing, it can not tolerate new SQL, and any FINAL call issued to a UDF at that time would be a 255 FINAL call. Note that UDFs which are not defined as containing any level of SQL access will never receive a 255 FINAL call, whereas UDFs which do use SQL can be given either type of FINAL call.

*Releasing resources*

Write routines to release any resources that they acquire. For table functions, there are two natural places for this release: the CLOSE call and the FINAL call. The CLOSE call balances each OPEN call and can occur multiple times in the execution of a statement. The FINAL call only occurs if FINAL CALL is specified for the UDF, and occurs only once per statement.

If you can apply a resource across all OPEN/FETCH/CLOSE sequences of the UDF, write the UDF to acquire the resource on the FIRST call and free it on the FINAL call. The scratchpad is a natural place to track this resource. For table functions, if FINAL CALL is specified, the scratchpad is initialized only before the FIRST call. If FINAL CALL is not specified, then it is reinitialized before each OPEN call.

If a resource is specific to each OPEN/FETCH/CLOSE sequence, write the UDF to free the resource on the CLOSE call.

**Note:** When a table function is in a subquery or join, it is very possible that there will be multiple occurrences of the OPEN/FETCH/CLOSE sequence, depending on how the DB2 Optimizer chooses to organize the execution of the statement.

The *call-type* takes the form of an INTEGER value. DB2 aligns the data for *call-type* according to the data type and the server operating system.

*dbinfo* This argument is set by DB2 before calling the routine. It is only present if the CREATE statement for the routine specifies the DBINFO keyword. The argument is the `sqludf_dbinfo` structure defined in the header file `sqludf.h`. The variables in this structure that contain names and identifiers might be longer than the longest value possible in this release of DB2, but they are defined this way for compatibility with future releases. You can use the length variable that complements each name and identifier variable to read or extract the portion of the variable that is actually used. The *dbinfo* structure contains the following elements:

1. Database name length (dbnamelen)

   The length of *database name* below. This field is an unsigned short integer.
2. Database name (dbname)

   The name of the currently connected database. This field is a long identifier of 128 characters. The *database name length* field described previously identifies the actual length of this field. It does not contain a null terminator or any padding.
3. Application Authorization ID Length (authidlen)

   The length of *application authorization ID* below. This field is an unsigned short integer.
4. Application authorization ID (authid)

The application run-time authorization ID. This field is a long identifier of 128 characters. It does not contain a null terminator or any padding. The *application authorization ID length* field described above identifies the actual length of this field.

5. Environment code pages (codepg)

   This is a union of three 48-byte structures; one is common to all DB2 database products (cdpg_db2), one is used by routines written for older versions of DB2 database (cdpg_cs), and the last is for use by older versions of DB2 UDB for z/OS and OS/390 (cdpg_mvs). For portability, it is recommended that the common structure, cdpg_db2, be used in all routines.

   The cdgp_db2 structure is made up of an array (db2_ccsids_triplet) of three sets of code page information representing the possible encoding schemes in the database as follows:

   a. ASCII encoding scheme. Note that for compatibility with previous version of DB2 database, if the database is a Unicode database then the information for the Unicode encoding scheme will be placed here as well as appearing in the third element.
   b. EBCDIC encoding scheme
   c. Unicode encoding scheme

   Following the encoding scheme information is the array index of the encoding scheme for the routine (db2_encoding_scheme).

   Each element of the array is composed of three fields:
   - db2_sbcs. Single byte code page, an unsigned long integer.
   - db2_dbcs. Double byte code page, an unsigned long integer.
   - db2_mixed. Composite code page (also called mixed code page), an unsigned long integer.
6. Schema name length (tbschemalen)

   The length of *schema name* below. Contains 0 (zero) if a table name is not passed. This field is an unsigned short integer.
7. Schema name (tbschema)

   Schema for the *table name* below. This field is a long identifier of 128 characters. It does not contain a null terminator or any padding. The *schema name length* field described previously identifies the actual length of this field.
8. Table name length (tbnamelen)

   The length of the *table name* below. Contains 0 (zero) if a table name is not passed. This field is an unsigned short integer.
9. Table name (tbname)

   This is the name of the table being updated or inserted. This field is set only if the routine reference is the right-side of a SET clause in an UPDATE statement, or an item in the VALUES list of an INSERT statement. This field is a long identifier of 128 characters. It does not contain a null terminator or any padding. The *table name length* field described previously, identifies the actual length of this field. The *schema name* field described previously, together with this field form the fully qualified table name.
10. Column name length (colnamelen)

    Length of *column name* below. It contains a 0 (zero) if a column name is not passed. This field is an unsigned short integer.
11. Column name (colname)

Under the exact same conditions as for table name, this field contains the name of the column being updated or inserted; otherwise, it is not predictable. This field is a long identifier of 128 characters. It does not contain a null terminator or any padding. The *column name length* field described above, identifies the actual length of this field.

12. Version/Release number (ver_rel)

    An 8 character field that identifies the product and its version, release, and modification level with the format *pppvvrrm* where:
    - *ppp* identifies the product as follows:

      | | |
      |---|---|
      | **DSN** | DB2 Universal Database for z/OS or OS/390 |
      | **ARI** | SQL/DS or DB2 for VM or VSE |
      | **QSQ** | DB2 Universal Database for iSeries™ |
      | **SQL** | DB2 Database for Linux, UNIX, and Windows |

    - *vv* is a two digit version identifier.
    - *rr* is a two digit release identifier.
    - *m* is a one digit modification level identifier.

13. Reserved field (resd0)

    This field is for future use.

14. Platform (platform)

    The operating system (platform) for the application server, as follows:

    | | |
    |---|---|
    | **SQLUDF_PLATFORM_AIX** | AIX |
    | **SQLUDF_PLATFORM_HP** | HP-UX |
    | **SQLUDF_PLATFORM_LINUX** | Linux |
    | **SQLUDF_PLATFORM_MVS** | OS/390 |
    | **SQLUDF_PLATFORM_NT** | Windows 2000, Windows XP |
    | **SQLUDF_PLATFORM_SUN** | Solaris operating system |
    | **SQLUDF_PLATFORM_WINDOWS95** | Windows 95, Windows 98, Windows Me |
    | **SQLUDF_PLATFORM_UNKNOWN** | Unknown operating system or platform |

    For additional operating systems that are not contained in the above list, see the contents of the `sqludf.h` file.

15. Number of table function column list entries (numtfcol)

    The number of non-zero entries in the table function column list specified in the *table function column list* field below.

16. Reserved field (resd1)

    This field is for future use.

17. Routine id of the stored procedure that invoked the current routine (procid)

    The stored procedure's routine id matches the ROUTINEID column in SYSCAT.ROUTINES, which can be used to retrieve the name of the invoking stored procedure. This field is a 32-bit signed integer.

18. Reserved field (resd2)

    This field is for future use.

19. Table function column list (tfcolumn)

    If this is a table function, this field is a pointer to an array of short integers that is dynamically allocated by DB2. If this is any other type of routine, this pointer is null.

This field is used only for table functions. Only the first *n* entries, where *n* is specified in the *number of table function column list entries* field, `numtfcol`, are of interest. *n* can be equal to 0, and *n* is less than or equal to the number of result columns defined for the function in the RETURNS TABLE(...) clause of the CREATE FUNCTION statement. The values correspond to the ordinal numbers of the columns that this 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, and the values can be in any order. Note that *n* could be equal to zero, that is, the variable `numtfcol` might be zero, for a statement similar to `SELECT COUNT(*) FROM TABLE(TF(...)) AS QQ`, where no actual column values are needed by the query.

This array represents an opportunity for optimization. The UDF need not return all values for all the result columns of the table function, only those needed in the particular context, and these are the columns identified (by number) in the array. Since this optimization can complicate the UDF logic in order to gain the performance benefit, the UDF can choose to return every defined column.

20. Unique application identifier (appl_id)

    This field is a pointer to a C null-terminated string that uniquely identifies the application's connection to DB2. It is generated by DB2 at connect time.

    The string has a maximum length of 32 characters, and its exact format depends on the type of connection established between the client and DB2. Generally it takes the form:

    ```
    x.y.ts
    ```

    where the *x* and *y* vary by connection type, but the *ts* is a 12 character time stamp of the form YYMMDDHHMMSS, which is potentially adjusted by DB2 to ensure uniqueness.

    ```
    Example:  *LOCAL.db2inst.980707130144
    ```

21. Reserved field (resd3)

    This field is for future use.

**Related concepts:**

- "External routine parameter styles" on page 269
- "Include file required for C and C++ routine development (sqludf.h)" on page 287

**Related reference:**

- "appl_id - Application ID monitor element" in *System Monitor Guide and Reference*
- "CREATE FUNCTION statement" in *SQL Reference, Volume 2*
- "CREATE METHOD statement" in *SQL Reference, Volume 2*
- "CREATE PROCEDURE statement" in *SQL Reference, Volume 2*
- "CREATE TYPE (Structured) statement" in *SQL Reference, Volume 2*

## Graphic host variables in C and C++ routines

Any routine written in C or C++ that receives or returns graphic data through its parameter input or output should generally be precompiled with the WCHARTYPE NOCONVERT option. This is because graphic data passed through these parameters is considered to be in DBCS format, rather than the `wchar_t` process code format. Using NOCONVERT means that graphic data manipulated in SQL statements in the routine will also be in DBCS format, matching the format of the parameter data.

With WCHARTYPE NOCONVERT, no character code conversion occurs between the graphic host variable and the database manager. The data in a graphic host variable is sent to, and received from, the database manager as unaltered DBCS characters. If you do not use WCHARTYPE NOCONVERT, it is still possible for you to manipulate graphic data in `wchar_t` format in a routine; however, you must perform the input and output conversions manually.

CONVERT can be used in FENCED routines, and it will affect the graphic data in SQL statements within the routine, but not data passed through the routine's parameters. NOT FENCED routines must be built using the NOCONVERT option.

In summary, graphic data passed to or returned from a routine through its input or output parameters is in DBCS format, regardless of how it was precompiled with the WCHARTYPE option.

**Related concepts:**

- "WCHARTYPE precompiler option for graphic data in C and C++ embedded SQL applications" on page 88

**Related reference:**

- "PRECOMPILE command" in *Command Reference*

## C++ type decoration

The names of C++ functions can be overloaded. Two C++ functions with the same name can coexist if they have different arguments, for example:

```
int func( int i )
```

and

```
int func( char c )
```

C++ compilers type-decorate or 'mangle' function names by default. This means that argument type names are appended to their function names to resolve them, as in `func__Fi` and `func__Fc` for the two earlier examples. The mangled names will be different on each operating system, so code that explicitly uses a mangled name is not portable.

On Windows operating systems, the type-decorated function name can be determined from the `.obj` (object) file.

With the Microsoft Visual C++ compiler on Windows, you can use the `dumpbin` command to determine the type-decorated function name from the `.obj` (object) file, as follows:

```
dumpbin /symbols myprog.obj
```

where `myprog.obj` is your program object file.

On UNIX operating systems, the type-decorated function name can be determined from the `.o` (object) file, or from the shared library, using the `nm` command. This command can produce considerable output, so it is suggested that you pipe the output through `grep` to look for the right line, as follows:

```
nm myprog.o | grep myfunc
```

where myprog.o is your program object file, and myfunc is the function in the program source file.

The output produced by all of these commands includes a line with the mangled function name. On UNIX, for example, this line is similar to the following:

```
myfunc__FPlT1PsT3PcN35|     3792|unamex|         | ...
```

Once you have obtained the mangled function name from one of the preceding commands, you can use it in the appropriate command. This is demonstrated later in this section using the mangled function name obtained from the preceding UNIX example. A mangled function name obtained on Windows would be used in the same way.

When registering a routine with the CREATE statement, the EXTERNAL NAME clause must specify the mangled function name. For example:

```
CREATE FUNCTION myfunco(...) RETURNS...
       ...
       EXTERNAL NAME '/whatever/path/myprog!myfunc__FPlT1PsT3PcN35'
       ...
```

If your routine library does not contain overloaded C++ function names, you have the option of using extern "C" to force the compiler to not type-decorate function names. (Note that you can always overload the SQL function names given to UDFs, because DB2 resolves what library function to invoke based on the name and the parameters it takes.)

```
#include <string.h>
#include <stdlib.h>
#include "sqludf.h"

/*---------------------------------------------------------------------*/
/* function fold: output = input string is folded at point indicated   */
/*                         by the second argument.                     */
/*        inputs: CLOB,                input string                    */
/*                LONG                 position to fold on             */
/*        output: CLOB                 folded string                   */
/*---------------------------------------------------------------------*/
extern "C" void fold(
      SQLUDF_CLOB      *in1,                  /* input CLOB to fold */
   ...
   ...
}
/* end of UDF: fold */

/*---------------------------------------------------------------------*/
/* function find_vowel:                                                */
/*            returns the position of the first vowel.                 */
/*            returns error if no vowel.                               */
/*            defined as NOT NULL CALL                                 */
/*        inputs: VARCHAR(500)                                         */
/*        output: INTEGER                                              */
/*---------------------------------------------------------------------*/
extern "C" void findvwl(
      SQLUDF_VARCHAR    *in,                  /* input smallint */
   ...
   ...
}
/* end of UDF: findvwl */
```

In this example, the UDFs `fold` and `findvwl` are not type-decorated by the compiler, and should be registered in the CREATE FUNCTION statement using their plain names. Similarly, if a C++ stored procedure or method is coded with `extern "C"`, its undecorated function name would be used in the CREATE statement.

**Related concepts:**

- "Parameter handling in PROGRAM TYPE MAIN or PROGRAM TYPE SUB procedures" in *Developing SQL and External Routines*
- "External routine parameter styles" on page 269

## Returning result sets from C and C++ procedures

You can develop C and C++ procedures that return result sets to a calling routine or application that is implemented using an API that supports the retrieval of procedure result sets. Most APIs support the retrieval of proedure result sets, however embedded SQL does not.

The C and C++ representation of a result set is an SQL cursor. Any SQL cursor that has been declared, opened, and not explicitly closed within a procedure, prior to the return of the procedure can be returned to the caller. The order in which result sets are returned to the caller is the same as the order in which cursor objects are opened within the routine. No additional parameters are required in the CREATE PROCEDURE statement or in the procedure implementation in order to return a result set.

**Prerequisites:**

A general understanding of how to create C and C++ routines will help you to follow the steps in the procedure below for returning results from a C or C++ procedure.

Creating C and C++ routines

Cursors declared in C or C++ embedded SQL procedures are not scrollable cursors.

**Procedure:**

To return a result set from a C or C++ procedure:

1. In the CREATE PROCEDURE statement for the C or C++ procedure you must specify along with any other appropriate clauses, the DYNAMIC RESULT SETS clause with a value equal to the maximum number of result sets that are to be returned by the procedure.
2. No parameter marker is required in the procedure declaration for a result set that is to be returned to the caller.
3. In the C or C++ procedure implementation of your routine, declare a cursor using the DECLARE CURSOR statement within the declaration section in which host variables are declared. The cursor declaration associates an SQL with the cursor.
4. Within the C or C++ routine code, open the cursor by executing the OPEN statement. This executes the query specified in the DECLARE CURSOR statement and associates the result of the query with the cursor.
5. Optional: Fetch rows in the result set associated with the cursor using the FETCH statement.

6. Do not execute the CLOSE statement used for closing the cursor at any point prior to the procedure's return to the caller. The open cursor will be returned as a result set to the caller when the procedure returns.

   When more than one cursor is left open upon the return of a procedure, the result sets associated with the cursors are returned to the caller in the order in which they were opened. No more than the maximum number of result sets specified by the DYNAMIC RESULT SETS clause value can be returned with the procedure. If the number of cursors left open in the procedure implementation is greater than the value specified by the DYNAMIC RESULT SETS clause, the excess result sets are simply not returned. No error or warning will be raised by DB2 in this situation.

Once the creation of the C or C++ procedure is completed successfully, you can invoke the procedure with the CALL statement from the DB2 Command Line Processor or a DB2 Command Window to verify that the result sets are successfully being returned to the caller.

For information on calling procedures and other types of routines:
- Routine invocation

### Precompilation and bind considerations for embedded SQL routines

Concept text

## Creating C and C++ routines

Procedures and functions that reference a C or C++ library are created in a similar way to external routines with other implementations. This task comprises a few steps including the formulation of the CREATE statement for the routine, the coding of the routine implementation, pre-compilation, compilation and linking of code, and the deployment of source code.

You would choose to implement a C or C++ routine if:
- You want to encapsulate complex logic into a routine that accesses the database or that performs an action outside of the database.
- You require the encapsulated logic to be invoked from any of: multiple applications, the CLP, another routine (procedure, function (UDF), or method), or a trigger.
- You are most comfortable coding this logic using an embedded SQL programming language such as C or C++.

**Prerequisites:**
- Knowledge of C and C++ routine implementation. To learn about C and C++ routines in general see:
  – C and C++ routines "C and C++ routines" on page 283
- The DB2 Client which includes application development support must be installed on the client computer.
- The database server must be running an operating system that supports a DB2 supported C or C++ compiler for routine development.
- The required compilers must be installed on the database server.
- Authority to execute the CREATE statement for the external routine. For the privileges required to execute the CREATE PROCEDURE statement or the CREATE FUNCTION statement, see the documentation for the statement.

**Procedure:**

1. Code the routine logic in the chosen programming language: C or C++.
   - For general information about C and C++ routines and C and C++ routine features, see the topics referenced in the Prerequisites section.
   - Include any C or C++ header files required for additional C functionality as well as the DB2 C or C++ header files required for SQL data type and SQL execution support. Include the following header files: sqludf.h, sql.h, sqlda.h, sqlca.h, and memory.h.
   - A routine parameter signature must be implemented using one of the supported parameter styles. It is strongly recommended that parameter style SQL be used for all C and C++ routines. Scratchpads and dbinfo structures are passed into C and C++ routines as parameters. For more on parameter signatures and parameter implementations see:
     - “Parameters in C and C++ routines” on page 288
     - “Parameter style SQL C and C++ procedures” on page 290
     - “Parameter style SQL C and C++ procedures” on page 290
   - Declare host variables and parameter markers in the same manner as is done for embedded SQL C and C++ applications. Be careful to correctly use data types that map to DB2 SQL data types. For more on data type mapping between DB2 and C or C++ data types refer to:
     - Supported SQL data types for C and C++ applications and routines
   - Include routine logic. Routine logic can consist of any code supported in the C or C++ programming language. It can also include the execution of embedded SQL statements which is implemented in the same way as for embedded SQL applications. For more on executing SQL statements in embedded SQL see:
     - ″Executing SQL statements in embedded SQL applications″
   - If the routine is a procedure and you want to return a result set to the caller of the routine, you do not require any parameters for the result set. For more on returning result sets from routines:
     - Returning result sets from C and C++ procedures
   - Set a routine return value at the end of the routine.
2. Build your code to produce a library file. For information on how to build embedded SQL C and C++ routines, see:
   - “Building C and C++ routine code” on page 328
3. Copy the library into the DB2 *function directory* on the database server. It is recommended that you store libraries associated with DB2 routines in the function directory. To find out more about the function directory, see the EXTERNAL clause of either of the following statements: CREATE PROCEDURE or CREATE FUNCTION.

   You can copy the library to another directory on the server, but to successfully invoke the routine you must note the fully qualified path name of your library as you will require it for the next step.
4. Execute either dynamically or statically the appropriate SQL language CREATE statement for the routine type: CREATE PROCEDURE or CREATE FUNCTION.
   - Specify the `LANGUAGE` clause with value: C
   - Specify the `PARAMETER STYLE` clause with the name of the supported parameter style that was implemented in the routine code. It is strongly recommended that PARAMETER STYLE SQL be used.

- Specify the `EXTERNAL` clause with the name of the library to be associated with the routine using one of the following values:
  - the fully qualified path name of the routine library
  - the relative path name of the routine library relative to the function directory.

  By default DB2 will look for the library in the function directory unless a fully qualified or relative path name for it is specified in the EXTERNAL clause.
- Specify DYNAMIC RESULT SETS with a numeric value if your routine is a procedure and it will return one or more result sets to the caller.
- Specify any other non-default clause values in the CREATE statement to be used to characterize the routine.

To invoke your C or C++ routine, see Routine invocation

**Related concepts:**
- "Parameters in C and C++ routines" on page 288
- "Parameter style SQL C and C++ procedures" on page 290
- "Parameter style SQL C and C++ functions" on page 293
- "Routine invocation" in *Developing SQL and External Routines*
- "C and C++ routines" on page 283

**Related tasks:**
- "Building C and C++ routine code" on page 328

**Related reference:**
- "Supported SQL data types in C and C++ routines" on page 300

## Building C and C++ routine code

### Building C and C++ routine code

Once embedded SQL C or C++ routine implementation code has been written, it must be built into a library and deployed before the routine can be invoked. Although the steps required to build embedded SQL C and C++ routines are similar to those required to build embedded SQL C and C++ applications, there are some differences. The same steps can be followed if there are no embedded SQL statements within the routines - the procedure will be faster and simpler.

There are two ways to build C and C++ routines:
- Using DB2 sample build scripts (UNIX) or build batch files (Windows)
- Entering DB2 and C or C++ compiler commands from a DB2 Command Window

The DB2 sample build scripts and batch files for routines are designed for building DB2 sample routines (procedures and user-defined functions) as well as user created routines for a particular operating system using the default supported compilers.

There is a separate set of DB2 sample build scripts and batch files for C and C++. In general it is easiest to build embedded SQL routines using the build scripts or batch files, which can easily be modified if required, however it is often helpful to know how to build routines from DB2 Command Windows as well.

For more information on each of the methods for building routines, refer to the related links.

**Related concepts:**
- "Binding embedded SQL packages to a database" on page 190
- "Building applications and routines written in C and C++" on page 203
- "Package storage and maintenance" on page 198
- "C and C++ routines" on page 283

**Related tasks:**
- "Building C and C++ routine code from DB2 Command Windows" on page 336
- "Building C and C++ routine code using sample bldrtn scripts" on page 329

**Related reference:**
- "AIX IBM COBOL routine compile and link options" on page 354
- "AIX C routine compile and link options" on page 338
- "AIX C++ routine compile and link options" on page 339
- "AIX Micro Focus COBOL routine compile and link options" on page 355
- "Windows C and C++ routine compile and link options" on page 348
- "Windows IBM COBOL routine compile and link options" on page 358
- "Windows Micro Focus COBOL routine compile and link options" on page 359

## Building C and C++ routine code using the sample bldrtn script

**Building C and C++ routine code using sample bldrtn scripts:** Building C and C++ routine source code is a sub-task of creating C and C++ routines. This task can be done quickly and easily using DB2 sample build scripts (UNIX) and batch files (Windows). The sample build scripts can be used for source code with or without embedded SQL statements. The build scripts take care of the pre-compilation, compilation, and linking of C and C++ source code that would otherwise have to be done in individual steps from the command line. They also take care of binding any packages to the specified database.

The sample build scripts for building C and C++ routines are named `bldrtn`. They are located in DB2 directories along with sample programs that can be built with them as follows:
- For C: `sqllib/samples/c/`
- For C++: `sqllib/samples/cpp/`

The `bldrtn` script can be used to build a source code file containing both procedures and function implementations. The script does the following:
- Establishes a connection with a user-specified database
- Precompiles the user-specified source code file
- Binds the package to the current database
- Compiles and links the source code to generate a shared library
- Copies the shared library to the DB2 function directory on the database server

The `bldrtn` scripts accept two arguments:

- The name of a source code file without any file extension
- The name of a database to which a connection will be established

The database parameter is optional. If no database name is supplied, the program uses the default sample database. Since routines must be built on the same instance where the database resides, no arguments are required for a user ID and password.

**Prerequisites:**
- Source code file containing one or more routine implementations.
- The name of the database within the current DB2 instance in which the routines are to be created.

**Procedure:**

To build a source code file that contains one or more routine code implementations, follow the steps below.
1. Open a DB2 Command Window.
2. Copy your source code file into the same directory as the `bldrtn` script file.
3. If the routines will be created in the sample database, enter the build script name followed by the name of the source code file without the .sqc or .sqC file extension:
   ```
   bldrtn <file-name>
   ```
   If the routines will be created in another database, enter the build script name, the source code file name without any file extension, and the database name:
   ```
   bldrtn <file-name> <database-name>
   ```
   The script precompiles, compiles and links the source code and produces a shared library. The script then copies the shared library to the function directory on the database server
4. If this is not the first time that the source code file containing the routine implementations was built, stop and restart the database to ensure the new version of the shared library is used by DB2. You can do this by entering `db2stop` followed by db2start on the command line.

Once you have successfully built the routine shared library and deployed it to the function directory on the database server, you should complete the steps associated with the task of creating C and C++ routines. After routine creation is completed you will be able to invoke your routines.

**Related reference:**
- “Windows Micro Focus COBOL routine compile and link options” on page 359
- “AIX C routine compile and link options” on page 338
- “AIX C++ routine compile and link options” on page 339
- “AIX IBM COBOL routine compile and link options” on page 354
- “AIX Micro Focus COBOL routine compile and link options” on page 355
- “Windows C and C++ routine compile and link options” on page 348
- “Windows IBM COBOL routine compile and link options” on page 358

**Building routines in C or C++ using the sample build script (UNIX):** DB2 provides build scripts for compiling and linking C and C++ programs. These are located in the `sqllib/samples/c` directory for routines in C and `sqllib/samples/cpp` directory for routines in C++, along with sample programs that can be built with these files.

The script, `bldrtn`, contains the commands to build routines (stored procedures and user-defined functions). The script compiles the routines into a shared library that can be loaded by the database manager and called by a client application.

The first parameter, `$1`, specifies the name of your source file. The second parameter, `$2`, specifies the name of the database to which you want to connect.

The database parameter is optional. If no database name is supplied, the program uses the default `sample` database. And since the stored procedure must be built on the same instance where the database resides, there are no parameters for user ID and password.

**Procedure:**

The following examples show you how to build routine shared libraries with:
- stored procedures
- non-embedded SQL user-defined functions (UDFs)
- embedded SQL user-defined functions (UDFs)

**Stored procedure shared library**

To build the sample program `spserver` from the source file `spserver.sqc` for C and `spserver.sqC` for C++:

1. If connecting to the `sample` database, enter the build script name and program name:

   ```
   bldrtn spserver
   ```

   If connecting to another database, also enter the database name:

   ```
   bldrtn spserver database
   ```

   The script copies the shared library to the server in the path `sqllib/function`.
2. Next, catalog the routines by running the `spcat` script on the server:

   ```
   spcat
   ```

   This script connects to the sample database, uncatalogs the routines if they were previously cataloged by calling `spdrop.db2`, then catalogs them by calling `spcreate.db2`, and finally disconnects from the database. You can also call the `spdrop.db2` and `spcreate.db2` scripts individually.
3. Then, if this is not the first time the stored procedure is built, stop and restart the database to ensure the new version of the shared library is recognized. You can do this by entering `db2stop` followed by `db2start` on the command line.

Once you build the shared library, `spserver`, you can build the client application, `spclient`, that accesses the shared library.

You can build `spclient` by using the script, `bldapp`.

To call the stored procedures in the shared library, run the sample client application by entering: `spclient` *database userid password* where

**database**
Is the name of the database to which you want to connect. The name could be `sample`, or its alias, or another database name.

**userid** Is a valid user ID.

**password**
Is a valid password for the user ID.

The client application accesses the shared library, `spserver`, and executes a number of stored procedure functions on the server database. The output is returned to the client application.

**Embedded SQL UDF shared library**

To build the embedded SQL user-defined function program, `udfemsrv`, from the source file `udfemsrv.sqc` for C and `udfemsrv.sqC` for C++, if connecting to the `sample` database, enter the build script name and program name:

```
bldrtn udfemsrv
```

If connecting to another database, also enter the database name:

```
bldrtn udfemsrv database
```

The script copies the UDF to the `sqllib/function` directory.

Once you build `udfemsrv`, you can build the client application, `udfemcli`, that calls it. You can build the `udfemcli` client program from the source file `udfemcli.sqc`, in `sqllib/samples/c`, using the script, `bldapp`.

To call the UDFs in the shared library, run the client application by entering:
`udfemcli` *database userid password*

where

**database**
Is the name of the database to which you want to connect. The name could be `sample`, or its alias, or another database name.

**userid** Is a valid user ID.

**password**
Is a valid password for the user ID.

The client application accesses the shared library, `udfemsrv`, and executes the user-defined functions on the server database. The output is returned to the client application.

**Related concepts:**
- "Building embedded SQL applications using the sample build script" on page 200

**Related tasks:**
- "Building applications in C or C++ using the sample build script (UNIX)" on page 203

**Related reference:**

- "AIX C routine compile and link options" on page 338
- "HP-UX C routine compile and link options" on page 340
- "Linux C routine compile and link options" on page 343
- "Solaris C routine compile and link options" on page 346

**Related samples:**
- "bldrtn -- Builds AIX C routines (stored procedures and UDFs) (C)"
- "bldrtn -- Builds HP-UX C routines (stored procedures and UDFs) (C)"
- "bldrtn -- Builds Linux C routines (stored procedures or UDFs) (C)"
- "bldrtn -- Builds Solaris C routines (stored procedures or UDFs) (C)"
- "embprep -- To prep and bind C/C++ and Micro Focus COBOL embedded SQL programs (C)"
- "spclient.sqc -- Call various stored procedures (C)"
- "spserver.sqc -- Definition of various types of stored procedures (C)"
- "udfemcli.sqc -- Call a variety of types of embedded SQL user-defined functions. (C)"
- "udfemsrv.sqc -- Call a variety of types of embedded SQL user-defined functions. (C)"

**Building C/C++ routines on Windows:** DB2 provides build scripts for compiling and linking DB2 API and embedded SQL programs in C and C++. These are located in the `sqllib\samples\c` and `sqllib\samples\cpp` directories, along with sample programs that can be built with these files.

The batch file `bldrtn.bat` contains the commands to build embedded SQL routines (stored procedures and user-defined functions). The batch file builds a DLL on the server. It takes two parameters, represented inside the batch file by the variables `%1` and `%2`.

The first parameter, `%1`, specifies the name of your source file. The batch file uses the source file name for the DLL name. The second parameter, `%2`, specifies the name of the database to which you want to connect. Since the DLL must be built on the same instance where the database resides, there are no parameters for user ID and password.

Only the first parameter, the source file name, is required. Database name is optional. If no database name is supplied, the program uses the default `sample` database.

**Procedure:**

The following examples show you how to build routine DLLs with:
- stored procedures
- non-embedded SQL user-defined functions (UDFs)
- embedded SQL user-defined functions (UDFs)

**Stored procedure DLL**

To build the `spserver` DLL from either the C source file, `spserver.sqc`, or the C++ source file, `spserver.sqx`:

1. Enter the batch file name and program name:
   ```
   bldrtn spserver
   ```

If connecting to another database, also enter the database name:

```
bldrtn spserver database
```

The batch file uses the module definition file `spserver.def`, contained in the same directory as the sample programs, to build the DLL. The batch file copies the DLL, `spserver.dll`, to the server in the path `sqllib\function`.

2. Next, catalog the routines by running the `spcat` script on the server:

```
spcat
```

This script connects to the sample database, uncatalogs the routines if they were previously cataloged by calling `spdrop.db2`, then catalogs them by calling `spcreate.db2`, and finally disconnects from the database. You can also call the `spdrop.db2` and `spcreate.db2` scripts individually.

3. Then, stop and restart the database to allow the new DLL to be recognized. If necessary, set the file mode for the DLL so the DB2 instance can access it.

Once you build the DLL, `spserver`, you can build the client application `spclient` that calls it.

You can build `spclient` by using the batch file, `bldapp.bat`.

To call the DLL, run the sample client application by entering:

`spclient` *database userid password*

where

**database**
Is the name of the database to which you want to connect. The name could be `sample`, or its alias, or another database name.

**userid** Is a valid user ID.

**password**
Is a valid password for the user ID.

The client application accesses the DLL, `spserver`, and executes a number of routines on the server database. The output is returned to the client application.

**Non-embedded SQL UDF DLL**

To build the user-defined function `udfsrv` from the source file `udfsrv.c`, enter:

```
bldrtn udfsrv
```

The batch file uses the module definition file, `udfsrv.def`, contained in the same directory as the sample program files, to build the user-defined function DLL. The batch file copies the user-defined function DLL, `udfsrv.dll`, to the server in the path `sqllib\function`.

Once you build `udfsrv`, you can build the client application, `udfcli`, that calls it. DB2 CLI, as well as embedded SQL C and C++ versions of this program are provided.

You can build the DB2 CLI `udfcli` program from the `udfcli.c` source file in `sqllib\samples\cli` using the batch file `bldapp`.

You can build the embedded SQL C `udfcli` program from the `udfcli.sqc` source file in `sqllib\samples\c` using the batch file `bldapp`.

You can build the embedded SQL C++ `udfcli` program from the `udfcli.sqx` source file in `sqllib\samples\cpp` using the batch file `bldapp`.

To run the UDF, enter:

```
udfcli
```

The calling application calls the `ScalarUDF` function from the `udfsrv` DLL.

**Embedded SQL UDF DLL**

To build the embedded SQL user-defined function library `udfemsrv` from the C source file `udfemsrv.sqc` in `sqllib\samples\c`, or from the C++ source file `udfemsrv.sqx` in `sqllib\samples\cpp`, enter:

```
bldrtn udfemsrv
```

If connecting to another database, also enter the database name:

```
bldrtn udfemsrv database
```

The batch file uses the module definition file, `udfemsrv.def`, contained in the same directory as the sample programs, to build the user-defined function DLL. The batch file copies the user-defined function DLL, `udfemsrv.dll`, to the server in the path `sqllib\function`.

Once you build `udfemsrv`, you can build the client application, `udfemcli`, that calls it. You can build `udfemcli` from the C source file `udfemcli.sqc` in `sqllib\samples\c`, or from the C++ source file `udfemcli.sqx` in `sqllib\samples\cpp` using the batch file `bldapp`.

To run the UDF, enter:

```
udfemcli
```

The calling application calls the UDFs in the `udfemsrv` DLL.

**Related reference:**

- "Windows C and C++ routine compile and link options" on page 348

**Related samples:**

- "bldrtn.bat -- Builds C routines (stored procedures and UDFs) on Windows"
- "embprep.bat -- Prep and binds a C/C++ or Micro Focus COBOL embedded SQL program on Windows"
- "spclient.sqc -- Call various stored procedures (C)"
- "spserver.sqc -- Definition of various types of stored procedures (C)"
- "udfcli.sqc -- Call a variety of types of user-defined functions (C)"
- "udfemcli.sqc -- Call a variety of types of embedded SQL user-defined functions. (C)"
- "udfemsrv.sqc -- Call a variety of types of embedded SQL user-defined functions. (C)"
- "udfsrv.c -- Defines a variety of types of user-defined functions (C)"
- "bldrtn.bat -- Builds C++ routines (stored procedures and UDFs) on Windows"
- "spclient.sqC -- Call various stored procedures (C++)"

- “spserver.sqC -- Definition of various types of stored procedures (C++)”
- “udfcli.sqC -- Call a variety of types of user-defined functions (C++)”
- “udfemcli.sqC -- Call a variety of types of embedded SQL user-defined functions. (C++)”
- “udfemsrv.sqC -- Call a variety of types of embedded SQL user-defined functions. (C++)”
- “udfsrv.C -- Defines a variety of types of user-defined functions (C++)”

## Building C and C++ routine code from DB2 Command Windows

Building C and C++ routine source code is a sub-task of creating C and C++ routines. This task can be done manually from the command line.The same procedure can be followed regardless of whether there are embedded SQL statements within the C or C++ routine code or not. The task steps include pre-compilation, compilation, and linking of C and C++ source code containing routine implementations, binding the generated package (if there were embedded SQL statements), and deploying the routine library. You might choose to do this task from a DB2 Command Window as part of testing the use of a precompiler, compiler, or bind option, if you want to defer binding the routine packages until a later time, or if you are developing customized build scripts.

As an alternative, you can use DB2 sample build scripts to simplify this task. Refer to: Building embedded SQL C and C++ routine code using sample build scripts.

**Prerequisites:**

- Source code file containing one or more embedded SQL C or C++ routine implementations.
- The name of the database within the current DB2 instance in which the routines are to be created.
- The operating specific compile and link options required for building C and C++ routines. Refer to the topics referenced in the related links at the bottom of this topic.

**Procedure:**

To build a source code file that contains one or more routine code implementations, follow the steps below. An example follows that demonstrates each of the steps:

1. Open a DB2 Command Window.
2. Navigate to the directory that contains your source code file.
3. Establish a connection with the database in which the routines will be created.
4. Precompile the source code file.
5. Bind the package that was generated to the database.
6. Compile the source code file.
7. Link the source code file to generate a shared library. This requires the use of some DB2 specific compile and link options for the compiler being used.
8. Copy the shared library to the DB2 function directory on the database server.
9. If this is not the first time that the source code file containing the routine implementations was built, stop and restart the database to ensure the new version of the shared library is used by DB2. You can do this by issuing the db2stop command followed by the db2start command.

Once you have successfully built and deployed the routine library, you should complete the steps associated with the task of creating C and C++ routines. Creating C and C++ routines includes a step for executing the CREATE statement for each routine that was implemented in the source code file. This step must also be completed before you will be able to invoke the routines.

**Example:**

The following example demonstrates the re-building of an embedded SQL C++ source code file named myfile.sqC containing routine implementations. The routines are being built on an AIX operating system using the default supported IBM VisualAge C++ compiler to generate a 32-bit routine library.

1. Open a DB2 Command Window.
2. Navigate to the directory that contains your source code file.
3. Establish a connection with the database in which the routines will be created.
   ```
   db2 connect to <database-name>
   ```
4. Precompile the source code file using the PREPARE command.
   ```
   db2 prep myfile.sqC bindfile
   ```
   The precompiler will generate output indicating if the precompilation proceeded successfully or if there were any errors. This step generates bindfile named myfile.bnd that can be used to generate a package in the next step.
5. Bind the package that was generated to the database using the BIND command.
   ```
   db2 bind myfile.bnd
   ```
   The bind utility will generate output indicating if the bind proceeded successfully or if there were any errors.
6. Compile the source code file using the recommended compile and link options:
   ```
   xlC_r -qstaticinline -I$HOME/sqllib/include -c $myfile.C
   ```
   The compiler will generate output if there are any errors. This step generates an export file named myfile.exp.
7. Link the source code file to generate a shared library.
   ```
   xlC_r -qmkshrobj -o $1 $1.o -L$ HOME/sqllib/include/lib32 -lDB2
   ```
   The linker will generate output if there are any errors. This step generates a shared library file name myfile.
8. Copy the shared library to the DB2 function directory on the database server.
   ```
   rm -f ~HOME/sqllib/function/myfile
   cp myfile $HOME/sqllib/function/myfile
   ```
   This step ensures that the routine library is in the default directory where DB2 looks for routine libraries. Refer to the topic on creating C and C++ routines for more on deploying routine libraries.
9. Stop and restart the database as this is a re-building of a previously built routine source code file.
   ```
   db2stop
   db2start
   ```

Building C and C++ routines is generally most easily done using the operating specific sample build scripts which also can be used as a reference for how to build routines from the command line.

**Related concepts:**

- "Binding embedded SQL packages to a database" on page 190
- "Precompilation of embedded SQL applications with the PRECOMPILE command" on page 182
- "Rebinding existing packages with the REBIND command" on page 194

**Related reference:**

- "AIX C routine compile and link options" on page 338
- "AIX C++ routine compile and link options" on page 339
- "AIX IBM COBOL routine compile and link options" on page 354
- "AIX Micro Focus COBOL routine compile and link options" on page 355
- "Windows C and C++ routine compile and link options" on page 348
- "Windows IBM COBOL routine compile and link options" on page 358
- "Windows Micro Focus COBOL routine compile and link options" on page 359
- "BIND command" in *Command Reference*
- "PRECOMPILE command" in *Command Reference*

## Compile and link options for C and C++ routines

**AIX C routine compile and link options:** The following are the compile and link options recommended by DB2 for building C routines (stored procedures and user-defined functions) with the AIX IBM C compiler, as demonstrated in the `bldrtn` build script.

**Compile and link options for bldrtn:**

Compile options:

**`xlc_r`**
Use the multi-threaded version of the IBM C compiler, needed as the routines can run in the same process as other routines (THREADSAFE) or in the engine itself (NOT FENCED).

**`$EXTRA_CFLAG`**
Contains "-q64" for an instance where 64-bit support is enabled; otherwise, it contains no value.

**`-I$DB2PATH/include`**
Specify the location of the DB2 include files. For example: `$HOME/sqllib/include`.

**`-c`**
Perform compile only; no link. Compile and link are separate steps.

Link options:

**xlc_r** Use the multi-threaded version of the compiler as a front end for the linker.

**$EXTRA_CFLAG**
Contains ″-q64″ for an instance where 64-bit support is enabled; otherwise, it contains no value.

**-qmkshrobj**
Create the shared library.

**-o $1** Specify the output file name.

**$1.o** Specify the object file.

**-ldb2** Link with the DB2 library.

**-L$DB2PATH/$LIB**
Specify the location of the DB2 runtime shared libraries. For example: `$HOME/sqllib/$LIB`. If you do not specify the `-L` option, the compiler assumes the following path: `/usr/lib:/lib`.

**-bE:$1.exp**
Specify an export file. The export file contains a list of the routines.

Refer to your compiler documentation for additional compiler options.

**Related tasks:**

- "Building routines in C or C++ using the sample build script (UNIX)" on page 330

**Related reference:**

- "AIX C embedded SQL and DB2 API applications compile and link options" on page 212

**Related samples:**

- "bldrtn -- Builds AIX C routines (stored procedures and UDFs) (C)"

**AIX C++ routine compile and link options:** The following are the compile and link options recommended by DB2 for building C++ routines (stored procedures and user-defined functions) with the AIX IBM XL C/C++ compiler, as demonstrated in the `bldrtn` build script.

**Compile and link options for bldrtn:**

Compile options:

**xlC_r** The multi-threaded version of the IBM XL C/C++ compiler, needed as the routines can run in the same process as other routines (THREADSAFE) or in the engine itself (NOT FENCED).

**$EXTRA_CFLAG**
Contains ″-q64″ for an instance where 64-bit support is enabled; otherwise, it contains no value.

**-I$DB2PATH/include**
Specify the location of the DB2 include files. For example: `$HOME/sqllib/include`.

**-c** Perform compile only; no link. Compile and link are separate steps.

Link options:

**xlC_r** Use the multi-threaded version of the compiler as a front-end for the linker.

**$EXTRA_CFLAG**
Contains ″-q64″ for an instance where 64-bit support is enabled; otherwise, it contains no value.

**-qmkshrobj**
Create a shared library.

**-o $1** Specify the output as a shared library file.

**$1.o** Specify the program object file.

**-L$DB2PATH/$LIB**
Specify the location of the DB2 runtime shared libraries. For example: `$HOME/sqllib/$LIB`. If you do not specify the `-L` option, the compiler assumes the following path: `/usr/lib:/lib`.

**-ldb2** Link with the DB2 library.

**-bE:$1.exp**
Specify an export file. The export file contains a list of the routines.

Refer to your compiler documentation for additional compiler options.

**Related tasks:**

- "Building routines in C or C++ using the sample build script (UNIX)" on page 330
- "Building embedded SQL stored procedures in C or C++ with configuration files" on page 348
- "Building user-defined functions in C or C++ with configuration files (AIX)" on page 350

**Related reference:**

- "AIX C++ embedded SQL and DB2 administrative API applications compile and link options" on page 213

**Related samples:**

- "bldrtn -- Builds AIX C++ routines (stored procedures and UDFs) (C++)"

**HP-UX C routine compile and link options:** The following are the compile and link options recommended by DB2 for building C routines (stored procedures and user-defined functions) with the HP-UX C compiler, as demonstrated in the `bldrtn` build script.

**Compile and link options for bldrtn:**

Compile options:

**`cc`** The C compiler.

**`$EXTRA_CFLAG`**
If the HP-UX platform is IA64 and 64-bit support is enabled, this flag contains the value **`+DD64`**; if 32-bit support is enabled, it contains the value **`+DD32`**. If the HP-UX platform is PA-RISC and 64-bit support is enabled, it contains the value **`+DA2.0W`**. For 32-bit support on a PA-RISC platform, this flag contains the value **`+DA2.0N`**.
- **`+DD64`** Must be used to generate 64-bit code for HP-UX on IA64.
- **`+DD32`** Must be used to generate 32-bit code for HP-UX on IA64.
- **`+DA2.0W`** Must be used to generate 64-bit code for HP-UX on PA-RISC.
- **`+DA2.0N`** Must be used to generate 32-bit code for HP-UX on PA-RISC.

**`+u1`** Allow unaligned data access. Use only if your application uses unaligned data.

**`+z`** Generate position-independent code.

**`-Ae`** Enables HP ANSI extended mode.

**`-I$DB2PATH/include`**
Specify the location of the DB2 include files. For example: `-I$DB2PATH/include`.

**`-D_POSIX_C_SOURCE=199506L`**
POSIX thread library option that ensures _REENTRANT is defined, needed as the routines can run in the same process as other routines (THREADSAFE) or in the engine itself (NOT FENCED).

**`-c`** Perform compile only; no link. Compile and link are separate steps.

Link options:

**`ld`** Use the linker to link.

**`-b`** Create a shared library rather than a normal executable.

**`-o $1`** Specify the output as a shared library file.

**`$1.o`** Specify the program object file.

**`$EXTRA_LFLAG`**
Specify the runtime path. If set, for 32-bit it contains the value `"+b$HOME/sqllib/lib32"`, and for 64-bit: `"+b$HOME/sqllib/lib64"`. If not set, it contains no value.

**`-L$DB2PATH/$LIB`**
Specify the location of the DB2 runtime shared libraries. For 32-bit: `$HOME/sqllib/lib32`; for 64-bit: `$HOME/sqllib/lib64`.

**`-ldb2`** Link with the DB2 library.

**`-lpthread`**
Link with the POSIX thread library.

Refer to your compiler documentation for additional compiler options.

**Related tasks:**
- “Building routines in C or C++ using the sample build script (UNIX)” on page 330

**Related samples:**
- “bldrtn -- Builds HP-UX C routines (stored procedures and UDFs) (C)”

**HP-UX C++ routine compile and link options:** The following are the compile and link options recommended by DB2 for building C++ routines (stored procedures and user-defined functions) with the HP-UX C++ compiler, as demonstrated in the `bldrtn` build script.

**Compile and link options for bldrtn:**

Compile options:

**aCC** The HP aC++ compiler.

**$EXTRA_CFLAG**
: If the HP-UX platform is IA64 and 64-bit support is enabled, this flag contains the value **+DD64**; if 32-bit support is enabled, it contains the value **+DD32**. If the HP-UX platform is PA-RISC and 64-bit support is enabled, it contains the value **+DA2.0W**. For 32-bit support on a PA-RISC platform, this flag contains the value **+DA2.0N**.
  - **+DD64** Must be used to generate 64-bit code for HP-UX on IA64.
  - **+DD32** Must be used to generate 32-bit code for HP-UX on IA64.
  - **+DA2.0W** Must be used to generate 64-bit code for HP-UX on PA-RISC.
  - **+DA2.0N** Must be used to generate 32-bit code for HP-UX on PA-RISC.

**+u1** Allows unaligned data access.

**+z** Generate position-independent code.

**-ext** Allow various C++ extensions including ″long long″ support.

**-mt** Allows threads support for the HP aC++ compiler, needed as the routines can run in the same process as other routines (THREADSAFE) or in the engine itself (NOT FENCED).

**-I$DB2PATH/include**
: Specify the location of the DB2 include files. For example: `$DB2PATH/include`

**-c** Perform compile only; no link. Compile and link are separate steps.

Link options:

**aCC** Use the HP aC++ compiler as a front end for the linker.

**$EXTRA_CFLAG**
: If the HP-UX platform is IA64 and 64-bit support is enabled, this flag contains the value **+DD64**; if 32-bit support is enabled, it contains the value **+DD32**. If the HP-UX platform is PA-RISC and 64-bit support is enabled, it contains the value **+DA2.0W**. For 32-bit support on a PA-RISC platform, this flag contains the value **+DA2.0N**.
  - **+DD64** Must be used to generate 64-bit code for HP-UX on IA64.
  - **+DD32** Must be used to generate 32-bit code for HP-UX on IA64.
  - **+DA2.0W** Must be used to generate 64-bit code for HP-UX on PA-RISC.
  - **+DA2.0N** Must be used to generate 32-bit code for HP-UX on PA-RISC.

**-mt** Allows threads support for the HP aC++ compiler, needed as the routines can run in the same process as other routines (THREADSAFE) or in the engine itself (NOT FENCED).

**-b** Create a shared library rather than a normal executable.

**-o $1** Specify the executable.

**$1.o** Specify the program object file.

**$EXTRA_LFLAG**
: Specify the runtime path. If set, for 32-bit it contains the value `-Wl,+b$HOME/sqllib/lib32`, and for 64-bit: `-Wl,+b$HOME/sqllib/lib64`. If not set, it contains no value.

**-L$DB2PATH/$LIB**
: Specify the location of the DB2 runtime shared libraries. For 32-bit: ″`$HOME/sqllib/lib32`″; for 64-bit: ″`$HOME/sqllib/lib64`″.

**-ldb2** Link with the DB2 library.

Refer to your compiler documentation for additional compiler options.

**Related concepts:**

- “Building applications and routines written in C and C++” on page 203

**Related tasks:**

- "Building routines in C or C++ using the sample build script (UNIX)" on page 330

**Related samples:**

- "bldrtn -- Builds HP-UX C++ routines (stored procedures and UDFs) (C++)"

**Linux C routine compile and link options:** The following are the compile and link options recommended by DB2 for building C routines (stored procedures and user-defined functions) with the Linux C compiler, as demonstrated in the `bldrtn` build script.

**Compile and link options for bldrtn:**

Compile options:

**`$CC`** The gcc or xlc_r compiler

**`$EXTRA_C_FLAGS`**
Contains one of the following:
- `-m31` on Linux for zSeries only, to build a 32-bit library;
- `-m32` on Linux for x86, x86_64 and POWER, to build a 32-bit library;
- `-m64` on Linux for zSeries, POWER, x86_64, to build a 64-bit library; or
- No value on Linux for IA64, to build a 64-bit library.

**`-I$DB2PATH/include`**
Specify the location of the DB2 include files.

**`-c`** Perform compile only; no link. This script file has separate compile and link steps.

**`-D_REENTRANT`**
Defines _REENTRANT, needed as the routines can run in the same process as other routines (THREADSAFE) or in the engine itself (NOT FENCED).

Link options:

**$CC** The gcc or xlc_r compiler; use the compiler as a front end for the linker.

**$LINK_FLAGS**
Contains the value "$EXTRA_C_FLAGS $SHARED_LIB_FLAG"

**$EXTRA_C_FLAGS**
Contains one of the following:
- `-m31` on Linux for zSeries only, to build a 32-bit library;
- `-m32` on Linux for x86, x86_64 and POWER, to build a 32-bit library;
- `-m64` on Linux for zSeries, POWER, x86_64, to build a 64-bit library; or
- No value on Linux for IA64, to build a 64-bit library.

**$SHARED_LIB_FLAG**
Contains `-shared` for gcc compiler or `-qmkshrobj` for xlc_r compiler.

**-o $1** Specify the executable.

**$1.o** Include the program object file.

**$EXTRA_LFLAG**
Specify the location of the DB2 shared libraries at run-time. For 32-bit it contains the value "`-Wl,-rpath,$DB2PATH/lib32`". For 64-bit it contains the value "`-Wl,-rpath,$DB2PATH/lib64`".

**-L$DB2PATH/$LIB**
Specify the location of the DB2 static and shared libraries at link-time. For example, for 32-bit: `$HOME/sqllib/lib32`, and for 64-bit: `$HOME/sqllib/lib64`.

**-ldb2** Link with the DB2 library.

**-lpthread**
Link with the POSIX thread library.

Refer to your compiler documentation for additional compiler options.

**Related tasks:**
- "Building routines in C or C++ using the sample build script (UNIX)" on page 330

**Related samples:**
- "bldrtn -- Builds Linux C routines (stored procedures or UDFs) (C)"

**Linux C++ routine compile and link options:** These are the compile and link options recommended by DB2 for building C++ routines (stored procedures and user-defined functions) with the Linux C++ compiler, as demonstrated in the `bldrtn` build script.

**Compile and link options for bldrtn:**

Compile options:

**`g++`** The GNU/Linux C++ compiler.

**`$EXTRA_C_FLAGS`**
Contains one of the following:
- `-m31` on Linux for zSeries only, to build a 32-bit library;
- `-m32` on Linux for x86, x86_64 and POWER, to build a 32-bit library;
- `-m64` on Linux for zSeries, POWER, x86_64, to build a 64-bit library; or
- No value on Linux for IA64, to build a 64-bit library.

**`-fpic`** Generate position independent code.

**`-I$DB2PATH/include`**
Specify the location of the DB2 include files.

**`-c`** Perform compile only; no link. This script file has separate compile and link steps.

**`-D_REENTRANT`**
Defines _REENTRANT, needed as the routines can run in the same process as other routines (THREADSAFE) or in the engine itself (NOT FENCED).

Link options:

**`g++`** Use the compiler as a front end for the linker.

**`$EXTRA_C_FLAGS`**
Contains one of the following:
- `-m31` on Linux for zSeries only, to build a 32-bit library;
- `-m32` on Linux for x86, x86_64 and POWER, to build a 32-bit library;
- `-m64` on Linux for zSeries, POWER, x86_64, to build a 64-bit library; or
- No value on Linux for IA64, to build a 64-bit library.

**`-shared`**
Generate a shared library.

**`-o $1`** Specify the executable.

**`$1.o`** Include the program object file.

**`$EXTRA_LFLAG`**
Specify the location of the DB2 shared libraries at run-time. For 32-bit it contains the value ″-Wl,-rpath,$DB2PATH/lib32″. For 64-bit it contains the value ″-Wl,-rpath,$DB2PATH/lib64″.

**`-L$DB2PATH/$LIB`**
Specify the location of the DB2 static and shared libraries at link-time. For example, for 32-bit: `$HOME/sqllib/lib32`, and for 64-bit: `$HOME/sqllib/lib64`.

**`-ldb2`** Link with the DB2 library.

**`-lpthread`**
Link with the POSIX thread library.

Refer to your compiler documentation for additional compiler options.

**Related concepts:**
- "Building applications and routines written in C and C++" on page 203

**Related samples:**
- "bldrtn -- Builds Linux C++ routines (stored procedures and UDFs) (C++)"

**Solaris C routine compile and link options:** These are the compile and link options recommended by DB2 for building C routines (stored procedures and user-defined functions) with the Forte C compiler, as demonstrated in the `bldrtn` build script.

**Compile and link options for bldrtn:**

Compile options:

**`cc`** The C compiler.

**`-xarch=$CFLAG_ARCH`**
This option ensures that the compiler will produce valid executables when linking with `libdb2.so`. The value for $CFLAG_ARCH is set to either ″v8plusa″ for 32-bit, or ″v9″ for 64-bit.

**`-mt`** Allow multi-threaded support, needed as the routines can run in the same process as other routines (THREADSAFE) or in the engine itself (NOT FENCED).

**`-DUSE_UI_THREADS`**
Allows Sun's ″UNIX International″ threads APIs.

**`-Kpic`** Generate position-independent code for shared libraries.

**`-I$DB2PATH/include`**
Specify the location of the DB2 include files.

**`-c`** Perform compile only; no link. This script has separate compile and link steps.

Link options:

**`cc`** Use the compiler as a front end for the linker.

**`-xarch=$CFLAG_ARCH`**
This option ensures that the compiler will produce valid executables when linking with `libdb2.so`. The value for $CFLAG_ARCH is set to either ″v8plusa″ for 32-bit, or ″v9″ for 64-bit.

**`-mt`** This is required because the DB2 library is linked with `-mt`.

**`-G`** Generate a shared library.

**`-o $1`** Specify the executable.

**`$1.o`** Include the program object file.

**`-L$DB2PATH/$LIB`**
Specify the location of the DB2 static and shared libraries at link-time. For example, for 32-bit: `$HOME/sqllib/lib32`, and for 64-bit: `$HOME/sqllib/lib64`.

**`$EXTRA_LFLAG`**
Specify the location of the DB2 shared libraries at run-time. For 32-bit it contains the value ″-R$DB2PATH/lib32″, and for 64-bit it contains the value ″-R$DB2PATH/lib64″.

**`-ldb2`** Link with the DB2 library.

Refer to your compiler documentation for additional compiler options.

**Related tasks:**

- "Building routines in C or C++ using the sample build script (UNIX)" on page 330

**Related samples:**

- "bldrtn -- Builds Solaris C routines (stored procedures or UDFs) (C)"

**Solaris C++ routine compile and link options:** These are the compile and link options recommended by DB2 for building C++ routines (stored procedures and user-defined functions) with the Forte C++ compiler, as demonstrated in the `bldrtn` build script.

**Compile and link options for bldrtn:**

Compile options:

**`CC`** The C++ compiler.

**`-xarch=$CFLAG_ARCH`**
This option ensures that the compiler will produce valid executables when linking with `libdb2.so`. The value for $CFLAG_ARCH is set to either ″v8plusa″ for 32-bit, or ″v9″ for 64-bit.

**`-mt`** Allow multi-threaded support, needed as the routines can run in the same process as other routines (THREADSAFE) or in the engine itself (NOT FENCED).

**`-DUSE_UI_THREADS`**
Allows Sun's ″UNIX International″ threads APIs.

**`-Kpic`** Generate position-independent code for shared libraries.

**`-I$DB2PATH/include`**
Specify the location of the DB2 include files.

**`-c`** Perform compile only; no link. This script has separate compile and link steps.

Link options:

**`CC`** Use the compiler as a front end for the linker.

**`-xarch=$CFLAG_ARCH`**
This option ensures that the compiler will produce valid executables when linking with `libdb2.so`. The value for $CFLAG_ARCH is set to either ″v8plusa″ for 32-bit, or ″v9″ for 64-bit.

**`-mt`** This is required because the DB2 library is linked with `-mt`.

**`-G`** Generate a shared library.

**`-o $1`** Specify the executable.

**`$1.o`** Include the program object file.

**`-L$DB2PATH/$LIB`**
Specify the location of the DB2 static and shared libraries at link-time. For example, for 32-bit: `$HOME/sqllib/lib32`, and for 64-bit: `$HOME/sqllib/lib64`.

**`$EXTRA_LFLAG`**
Specify the location of the DB2 shared libraries at run-time. For 32-bit it contains the value ″-R$DB2PATH/lib32″, and for 64-bit it contains the value ″-R$DB2PATH/lib64″.

**`-ldb2`** Link with the DB2 library.

Refer to your compiler documentation for additional compiler options.

**Related concepts:**

- "Building applications and routines written in C and C++" on page 203

**Related samples:**

- "bldrtn -- Builds Solaris C++ routines (stored procedures or UDFs) (C++)"

**Windows C and C++ routine compile and link options:** The following are the compile and link options recommended by DB2 for building C and C++ routines (stored procedures and user-defined functions) on Windows with the Microsoft Visual C++ compiler, as demonstrated in the `bldrtn.bat` batch file.

**Compile and link options for bldrtn:**

| | |
|---|---|
| Compile options: | |
| **`%BLDCOMP%`** | Variable for the compiler. The default is `cl`, the Microsoft Visual C++ compiler. It can be also set to `icl`, the Intel C++ Compiler for 32-bit and 64-bit applications, or `ecl`, the Intel C++ Compiler for Itanium 64-bit applications. |
| **`-Zi`** | Enable debugging information |
| **`-Od`** | Disable optimization. |
| **`-c`** | Perform compile only; no link. Compile and link are separate steps. |
| **`-W2`** | Output warning, error, and severe and unrecoverable error messages. |
| **`-DWIN32`** | Compiler option necessary for Windows operating systems. |
| **`-MD`** | Link using MSVCRT.LIB |
| Link options: | |
| **`link`** | Use the linker to link. |
| **`-debug`** | Include debugging information. |
| **`-out:%1.dll`** | Build a .DLL file. |
| **`%1.obj`** | Include the object file. |
| **`db2api.lib`** | Link with the DB2 library. |
| **`-def:%1.def`** | Module definition file. |
| Refer to your compiler documentation for additional compiler options. | |

**Related tasks:**
- "Building C/C++ routines on Windows" on page 333

**Related samples:**
- "bldrtn.bat -- Builds C++ routines (stored procedures and UDFs) on Windows"
- "bldrtn.bat -- Builds C routines (stored procedures and UDFs) on Windows"

## Building embedded SQL stored procedures in C or C++ with configuration files

The configuration file, `stp.icc`, in `sqllib/samples/c` and `sqllib/samples/cpp`, allows you to build DB2 embedded SQL stored procedures in C and C++ on AIX.

**Procedure:**

To use the configuration file to build the embedded SQL stored procedure shared library `spserver` from the source file `spserver.sqc`, do the following:
1. Set the STP environment variable to the program name by entering:

- For bash or Korn shell:
  ```
  export STP=spserver
  ```
- For C shell:
  ```
  setenv STP spserver
  ```

2. If you have an `stp.ics` file in your working directory, produced by building a different program with the `stp.icc` file, delete the `stp.ics` file with this command:
   ```
   rm stp.ics
   ```
   An existing `stp.ics` file produced for the same program you are going to build again does not have to be deleted.
3. Compile the sample program by entering:
   ```
   vacbld stp.icc
   ```
   **Note:** The `vacbld` command is provided by VisualAge C++.

The stored procedure shared library is copied to the server in the path `sqllib/function`.

Next, catalog the stored procedures in the shared library by running the `spcat` script on the server:

```
spcat
```

This script connects to the sample database, uncatalogs the stored procedures if they were previously cataloged by calling `spdrop.db2`, then catalogs them by calling `spcreate.db2`, and finally disconnects from the database. You can also call the `spdrop.db2` and `spcreate.db2` scripts individually.

Then, stop and restart the database to allow the new shared library to be recognized. If necessary, set the file mode for the shared library so the DB2 instance can access it.

Once you build the stored procedure shared library, `spserver`, you can build the client application, `spclient`, that calls the stored procedures in it. You can build `spclient` using the configuration file, `emb.icc`.

To call the stored procedures, run the sample client application by entering:
`spclient` *database userid password*

where

**database**
Is the name of the database to which you want to connect. The name could be `sample`, or its remote alias, or some other name.

**userid** Is a valid user ID.

**password**
Is a valid password.

The client application accesses the shared library, `spserver`, and executes a number of stored procedure functions on the server database. The output is returned to the client application.

**Related tasks:**

- "Building embedded SQL and DB2 API applications in C or C++ with configuration files (AIX)" on page 208
- "Building user-defined functions in C or C++ with configuration files (AIX)" on page 350
- "Setting up the embedded SQL development environment" on page 11

## Building user-defined functions in C or C++ with configuration files (AIX)

The configuration file, `udf.icc`, in `sqllib/samples/c` and `sqllib/samples/cpp`, allows you to build user-defined functions in C and C++ on AIX.

**Procedure:**

To use the configuration file to build the user-defined function program `udfsrv` from the source file `udfsrv.c`, do the following:

1. Set the UDF environment variable to the program name by entering:
   - For bash or Korn shell:
     ```
     export UDF=udfsrv
     ```
   - For C shell:
     ```
     setenv UDF udfsrv
     ```
2. If you have a `udf.ics` file in your working directory, produced by building a different program with the `udf.icc` file, delete the `udf.ics` file with this command:
   ```
   rm udf.ics
   ```

   An existing `udf.ics` file produced for the same program you are going to build again does not have to be deleted.
3. Compile the sample program by entering:
   ```
   vacbld udf.icc
   ```

   **Note:** The `vacbld` command is provided by VisualAge C++.

The UDF library is copied to the server in the path `sqllib/function`.

If necessary, set the file mode for the user-defined function so the DB2 instance can run it.

Once you build `udfsrv`, you can build the client application, `udfcli`, that calls it. DB2 CLI and embedded SQL versions of this program are provided.

You can build the DB2 CLI `udfcli` program from the source file `udfcli.c`, in `sqllib/samples/cli`, by using the configuration file `cli.icc`.

You can build the embedded SQL `udfcli` program from the source file `udfcli.sqc`, in `sqllib/samples/c`, by using the configuration file `emb.icc`.

To call the UDF, run the sample calling application by entering the executable name:

```
udfcli
```

The calling application calls the `ScalarUDF` function from the `udfsrv` library.

**Related tasks:**

- "Building embedded SQL and DB2 API applications in C or C++ with configuration files (AIX)" on page 208
- "Building embedded SQL stored procedures in C or C++ with configuration files" on page 348
- "Setting up the embedded SQL development environment" on page 11

### Rebuilding DB2 routine shared libraries

DB2 will cache the shared libraries used for stored procedures and user-defined functions once loaded. If you are developing a routine, you might want to test loading the same shared library a number of times, and this caching can prevent you from picking up the latest version of a shared library. The way to avoid caching problems depends on the type of routine:

1. **Fenced, not threadsafe routines.** The database manager configuration keyword KEEPFENCED has a default value of YES. This keeps the fenced mode process alive. This default setting can interfere with reloading the library. It is best to change the value of this keyword to NO while developing fenced, not threadsafe routines, and then change it back to YES when you are ready to load the final version of your shared library. For more information, see Updating the database manager configuration file .
2. **Trusted or threadsafe routines.** Except for SQL routines (including SQL procedures), the only way to ensure that an updated version of a DB2 routine library is picked up when that library is used for trusted, or threadsafe routines, is to recycle the DB2 instance by entering db2stop followed by db2start on the command line. This is not needed for an SQL routine because when it is recreated, the compiler uses a new unique library name to prevent possible conflicts.

For routines other than SQL routines, you can also avoid caching problems by creating the new version of the routine with a differently named library (for example foo.a becomes foo.1.a), and then using either the ALTER PROCEDURE or ALTER FUNCTION SQL statement with the new library.

**Related tasks:**

- "Updating the database manager configuration file" in *Developing SQL and External Routines*

**Related reference:**

- "ALTER FUNCTION statement" in *SQL Reference, Volume 2*
- "ALTER PROCEDURE statement" in *SQL Reference, Volume 2*

## COBOL procedures

### COBOL procedures

COBOL procedures are to be written in a similar manner as COBOL subprograms.

**Handling parameters in a COBOL procedure**

Each parameter to be accepted or passed by a procedure must be declared in the LINKAGE SECTION. For example, this code fragment comes from a procedure that accepts two IN parameters (one CHAR(15) and one INT), and passes an OUT parameter (an INT):

```
LINKAGE SECTION.
01  IN-SPERSON    PIC X(15).
01  IN-SQTY       PIC S9(9)  USAGE COMP-5.
01  OUT-SALESSUM  PIC S9(9)  USAGE COMP-5.
```

Ensure that the COBOL data types you declare map correctly to SQL data types. For a detailed list of data type mappings between SQL and COBOL, see ″Supported SQL Data Types in COBOL″.

Each parameter must then be listed in the PROCEDURE DIVISION. The following example shows a PROCEDURE DIVISION that corresponds to the parameter definitions from the previous LINKAGE SECTION example.

```
PROCEDURE DIVISION USING IN-SPERSON
                         IN-SQTY
                         OUT-SALESSUM.
```

**Exiting a COBOL procedure**

To properly exit the procedure use the following commands:

```
MOVE SQLZ-HOLD-PROC TO RETURN-CODE.
GOBACK.
```

With these commands, the procedure returns correctly to the client application. This is especially important when the procedure is called by a local COBOL client application.

When building a COBOL procedure, it is strongly recommended that you use the build script written for your operating system and compiler. Build scripts for Micro Focus COBOL are found in the sqllib/samples/cobol_mf directory. Build scripts for IBM COBOL are found in the sqllib/samples/cobol directory.

The following is an example of a COBOL procedure that accepts two input parameters, and then returns an output parameter and a result set:

```
IDENTIFICATION DIVISION.
PROGRAM-ID.    "NEWSALE".
DATA DIVISION.

WORKING-STORAGE SECTION.
01  INSERT-STMT.
    05  FILLER   PIC X(24) VALUE "INSERT INTO SALES (SALES".
    05  FILLER   PIC X(24) VALUE "_PERSON,SALES) VALUES ('".
    05  SPERSON  PIC X(16).
    05  FILLER   PIC X(2) VALUE "',".
    05  SQTY     PIC S9(9).
    05  FILLER   PIC X(1) VALUE ")".
    EXEC SQL BEGIN DECLARE SECTION END-EXEC.
01  INS-SMT-INF.
    05  INS-STMT.
    49  INS-LEN   PIC S9(4) USAGE COMP.
    49  INS-TEXT  PIC X(100).
01  SALESSUM      PIC S9(9)  USAGE COMP-5.
    EXEC SQL END DECLARE SECTION END-EXEC.
    EXEC SQL INCLUDE SQLCA END-EXEC.

LINKAGE SECTION.
01  IN-SPERSON    PIC X(15).
01  IN-SQTY       PIC S9(9)  USAGE COMP-5.
01  OUT-SALESSUM  PIC S9(9)  USAGE COMP-5.

PROCEDURE DIVISION USING IN-SPERSON
                         IN-SQTY
                         OUT-SALESSUM.
MAINLINE.
```

```
           MOVE 0 TO SQLCODE.
           PERFORM INSERT-ROW.
           IF SQLCODE IS NOT EQUAL TO 0
              GOBACK
           END-IF.
           PERFORM SELECT-ROWS.
           PERFORM GET-SUM.
           GOBACK.
       INSERT-ROW.
           MOVE IN-SPERSON TO SPERSON.
           MOVE IN-SQTY TO SQTY.
           MOVE           INSERT-STMT TO INS-TEXT.
           MOVE LENGTH OF INSERT-STMT TO INS-LEN.
           EXEC SQL EXECUTE IMMEDIATE :INS-STMT END-EXEC.
       GET-SUM.
           EXEC SQL
              SELECT SUM(SALES) INTO :SALESSUM FROM SALES
           END-EXEC.
           MOVE SALESSUM TO OUT-SALESSUM.
       SELECT-ROWS.
           EXEC SQL
              DECLARE CUR CURSOR WITH RETURN FOR SELECT * FROM SALES
           END-EXEC.
           IF SQLCODE = 0
              EXEC SQL OPEN CUR END-EXEC
           END-IF.
```

The corresponding CREATE PROCEDURE statement for this procedure is as follows:

```
CREATE PROCEDURE NEWSALE ( IN SALESPERSON CHAR(15),
                           IN SALESQTY INT,
                           OUT SALESSUM INT)
  RESULT SETS 1
  EXTERNAL NAME 'NEWSALE!NEWSALE'
  FENCED
  LANGUAGE COBOL
  PARAMETER STYLE SQL
  MODIFIES SQL DATA
```

The preceding statement assumes that the COBOL function exists in a library called `NEWSALE`.

**Note:** When registering a COBOL procedure on Windows operating systems, take the following precaution when identifying a stored procedure body in the CREATE statement's EXTERNAL NAME clause. If you use an absolute path id to identify the procedure body, you must append the `.dll` extension. For example:

```
CREATE PROCEDURE NEWSALE ( IN SALESPERSON CHAR(15),
                           IN SALESQTY INT,
                           OUT SALESSUM INT)
  RESULT SETS 1
  EXTERNAL NAME 'NEWSALE!NEWSALE'
  FENCED
  LANGUAGE COBOL
  PARAMETER STYLE SQL
  MODIFIES SQL DATA
  EXTERNAL NAME 'd:\mylib\NEWSALE.dll'
```

**Related concepts:**

- "Embedded SQL statements in COBOL applications" on page 6
- "External routines" on page 247

**Related tasks:**
- "Building IBM COBOL routines on AIX" on page 360
- "Building IBM COBOL routines on Windows" on page 362
- "Building UNIX Micro Focus COBOL routines" on page 361
- "Building Micro Focus COBOL routines on Windows" on page 364

**Related reference:**
- "Supported SQL data types in COBOL embedded SQL applications" on page 60
- "CREATE PROCEDURE (External) statement" in *SQL Reference, Volume 2*
- "Passing arguments to C, C++, OLE, or COBOL routines" on page 311
- "Support for external procedure development in COBOL" on page 354

## Support for external procedure development in COBOL

To develop external procedures in COBOL you must use the supported COBOL development software.

All of the development software supported for database application development in COBOL can also be used for external procedure development in COBOL.

**Related concepts:**
- "COBOL procedures" on page 351
- "Supported APIs and programming languages for external routine development" on page 260

**Related reference:**
- "Support for database application development in COBOL" in *Getting Started with Database Application Development*

## Building COBOL routines

### Compile and link options for COBOL routines

**AIX IBM COBOL routine compile and link options:** The following are the compile and link options recommended by DB2 for building COBOL routines (stored procedures) with the IBM COBOL for AIX compiler on AIX, as demonstrated in the `bldrtn` build script.

**Compile and link options for bldrtn:**

Compile Options:

**`cob2`** The IBM COBOL for AIX compiler.

**`-qpgmname\(mixed\)`**
Instructs the compiler to permit CALLs to library entry points with mixed-case names.

**`-qlib`** Instructs the compiler to process COPY statements.

**`-c`** Perform compile only; no link. Compile and link are separate steps.

**`-I$DB2PATH/include/cobol_a`**
Specify the location of the DB2 include files. For example: `$HOME/sqllib/include/cobol_a`.

Link Options:

**cob2** Use the compiler to link edit.

**-o $1** Specify the output as a shared library file.

**$1.o** Specify the stored procedure object file.

**checkerr.o**
Include the utility object file for error-checking.

**-bnoentry**
Do not specify the default entry point to the shared library.

**-bE:$1.exp**
Specify an export file. The export file contains a list of the stored procedures.

**-L$DB2PATH/$LIB**
Specify the location of the DB2 runtime shared libraries. For example: `$HOME/sqllib/lib32`.

**-ldb2** Link with the database manager library.

Refer to your compiler documentation for additional compiler options.

**Related tasks:**
- "Building IBM COBOL routines on AIX" on page 360
- "Configuring the IBM COBOL compiler on AIX" on page 231

**Related reference:**
- "AIX IBM COBOL application compile and link options" on page 234
- "Include files for COBOL embedded SQL applications" on page 45

**Related samples:**
- "bldrtn -- Builds AIX COBOL routines (stored procedures)"

**AIX Micro Focus COBOL routine compile and link options:** The following are the compile and link options recommended by DB2 for building COBOL routines (stored procedures) with the Micro Focus COBOL compiler on AIX, as demonstrated in the `bldrtn` build script. Note that the DB2 MicroFocus COBOL include files are found by setting up the COBCPY environment variable, so no -I flag is needed in the compile step. Refer to the bldapp script for an example.

**Compile and link options for bldrtn:**

Compile options:

**cob** The MicroFocus COBOL compiler.

**-c** Perform compile only; no link. Compile and link are separate steps.

**$EXTRA_COBOL_FLAG="-C MFSYNC"**
Enables 64-bit support.

**-x** Compile to an object module when used with the `-c` option.

Link options:

**cob** Use the compiler as a front-end for the linker.

**-x** Produce a shared library.

**-o $1** Specify the executable program.

**$1.o** Specify the program object file.

**-L$DB2PATH/$LIB**
Specify the location of the DB2 runtime shared libraries. For example: `$HOME/sqllib/lib32`.

**-ldb2** Link to the DB2 library.

**-ldb2gmf**
Link to the DB2 exception-handler library for Micro Focus COBOL.

Refer to your compiler documentation for additional compiler options.

**Related tasks:**
- "Building UNIX Micro Focus COBOL routines" on page 361
- "Configuring the Micro Focus COBOL compiler on AIX" on page 232

**Related reference:**
- "AIX Micro Focus COBOL application compile and link options" on page 234

**Related samples:**
- "bldrtn -- Builds AIX Micro Focus COBOL routines (stored procedures)"

**HP-UX Micro Focus COBOL routine compile and link options:** The following are the compile and link options recommended by DB2 for building COBOL routines (stored procedures) with the Micro Focus COBOL compiler on HP-UX, as demonstrated in the `bldrtn` build script.

**Compile and link options for bldrtn:**

Compile options:
**cob** The COBOL compiler.
**$EXTRA_COBOL_FLAG**
Contains "-C MFSYNC" if the HP-UX platform is IA64 and 64-bit support is enabled.

Link options:
**-y** Specify that the desired output is a shared library.
**-o $1** Specify the executable.
**-L$DB2PATH/$LIB**
Specify the location of the DB2 runtime shared libraries.
**-ldb2** Link to the DB2 shared library.
**-ldb2gmf**
Link to the DB2 exception-handler library for Micro Focus COBOL.

Refer to your compiler documentation for additional compiler options.

**Related tasks:**
- "Building UNIX Micro Focus COBOL routines" on page 361
- "Configuring the Micro Focus COBOL compiler on HP-UX" on page 232

**Related samples:**
- "bldrtn -- Builds HP-UX Micro Focus COBOL routines (stored procedures)"

**Solaris Micro Focus COBOL routine compile and link options:** The following are the compile and link options recommended by DB2 for building COBOL routines (stored procedures) with the Micro Focus COBOL compiler on Solaris, as demonstrated in the `bldrtn` build script.

**Compile and link options for bldrtn:**

Compile options:

| | |
|---|---|
| **`cob`** | The COBOL compiler. |
| **`-cx`** | Compile to object module. |
| **`$EXTRA_COBOL_FLAG`** | For 64-bit support, contains the value ″-C MFSYNC″; otherwise it contains no value. |

Link options:

| | |
|---|---|
| **`cob`** | Use the compiler as a front-end for the linker. |
| **`-y`** | Create a self-contained standalone shared library. |
| **`-o $1`** | Specify the executable program. |
| **`$1.o`** | Specify the program object file. |
| **`-L$DB2PATH/$LIB`** | Specify the location of the DB2 runtime shared libraries. For example: `$HOME/sqllib/lib64`. |
| **`-ldb2`** | Link to the DB2 library. |
| **`-ldb2gmf`** | Link to the DB2 exception-handler library for Micro Focus COBOL. |

Refer to your compiler documentation for additional compiler options.

**Related tasks:**
- "Building UNIX Micro Focus COBOL routines" on page 361
- "Configuring the Micro Focus COBOL compiler on Solaris" on page 233

**Related samples:**
- "bldrtn -- Builds Solaris Micro Focus COBOL routines (stored procedures)"

**Linux Micro Focus COBOL routine compile and link options:** The following are the compile and link options recommended by DB2 for building COBOL routines (stored procedures) with the Micro Focus COBOL compiler on Linux, as demonstrated in the `bldrtn` build script.

**Compile and link options for bldrtn:**

Compile and Link options:

**`cob`** The COBOL compiler

**`$EXTRA_COBOL_FLAG`**
: For 64-bit support, contains the value ″-C MFSYNC″; otherwise it contains no value.

**`-y`** Specify to compile to self-contained callable shared object

**`-o $1`** Specify the executable.

**`$1.cbl`** Specify the source file

**`-L$DB2PATH/$LIB`**
: Specify the location of the DB2 runtime shared libraries.

**`-ldb2`** Link to the DB2 library.

**`-ldb2gmf`**
: Link to the DB2 exception-handler library for Micro Focus COBOL.

Refer to your compiler documentation for additional compiler options.

**Related tasks:**

- "Building UNIX Micro Focus COBOL routines" on page 361
- "Configuring the Micro Focus COBOL compiler on Linux" on page 230

**Related samples:**

- "bldrtn -- Builds Linux Micro Focus COBOL routines (stored procedures)"
- "embprep -- To prep and bind C/C++ and Micro Focus COBOL embedded SQL programs (C)"

**Windows IBM COBOL routine compile and link options:** The following are the compile and link options recommended by DB2 for building COBOL routines (stored procedures and user-defined functions) on Windows with the IBM VisualAge COBOL compiler, as demonstrated in the `bldrtn.bat` batch file.

**Compile and link options for bldrtn:**

Compile options:

**`cob2`** The IBM VisualAge COBOL compiler.

**`-qpgmname(mixed)`**
: Instructs the compiler to permit CALLs to library entry points with mixed-case names.

**`-c`** Perform compile only; no link. This batch file has separate compile and link steps.

**`-qlib`** Instructs the compiler to process COPY statements.

**`-I`***path* Specify the location of the DB2 include files. For example: `-I"%DB2PATH%\include\cobol_a"`.

**`%EXTRA_COMPFLAG%`**
: If `"set IBMCOB_PRECOMP=true"` is uncommented, the IBM COBOL precompiler is used to precompile the embedded SQL. It is invoked with one of the following formulations, depending on the input parameters:

  **`-q"SQL('database sample CALL_RESOLUTION DEFERRED')"`**
  : precompile using the default sample database, and defer call resolution.

  **`-q"SQL('database %2 CALL_RESOLUTION DEFERRED')"`**
  : precompile using a database specified by the user, and defer call resolution.

Link options:

| | |
|---|---|
| **ilink** | Use the IBM VisualAge COBOL linker. |
| **/free** | Free format. |
| **/nol** | No logo. |
| **/dll** | Create the DLL with the source program name. |
| **db2api.lib** | Link with the DB2 library. |
| **%1.exp** | Include the export file. |
| **%1.obj** | Include the program object file. |
| **iwzrwin3.obj** | Include the object file provided by IBM VisualAge COBOL. |

Refer to your compiler documentation for additional compiler options.

**Related tasks:**

- "Building IBM COBOL routines on Windows" on page 362
- "Configuring the IBM COBOL compiler on Windows" on page 228

**Related samples:**

- "bldrtn.bat -- Builds Windows VisualAge COBOL routines (stored procedures)"

**Windows Micro Focus COBOL routine compile and link options:** The following are the compile and link options recommended by DB2 for building COBOL routines (stored procedures and user-defined functions) on Windows with the Micro Focus COBOL compiler, as demonstrated in the `bldrtn.bat` batch file.

**Compile and link options for bldrtn:**

Compile options:

| | |
|---|---|
| **cobol** | The Micro Focus COBOL compiler. |
| **/case** | Prevent external symbols being converted to uppercase. |

Link options:

| | |
|---|---|
| **cbllink** | Use the Micro Focus COBOL linker to link edit. |
| **/d** | Create a .dll file. |
| **db2api.lib** | Link with the DB2 API library. |

Refer to your compiler documentation for additional compiler options.

**Related tasks:**

- "Building Micro Focus COBOL routines on Windows" on page 364
- "Configuring the Micro Focus COBOL compiler on Windows" on page 229

**Related samples:**

- "bldrtn.bat -- Builds Windows Micro Focus Cobol routines (stored procedures)"

## Building IBM COBOL routines on AIX

DB2 provides build scripts for compiling and linking COBOL embedded SQL and DB2 administrative API programs. These are located in the `sqllib/samples/cobol` directory, along with sample programs that can be built with these files.

The script, `bldrtn`, in `sqllib/samples/cobol`, contains the commands to build routines (stored procedures). The script compiles the routines into a shared library that can be called by a client application.

The first parameter, `$1`, specifies the name of your source file. The second parameter, `$2`, specifies the name of the database to which you want to connect. Since the shared library must be built on the same instance where the database resides, there are no parameters for user ID and password.

Only the first parameter, source file name, is required. The script uses the source file name, `$1`, for the shared library name. Database name is optional. If no database name is supplied, the program uses the default `sample` database.

**Procedure:**

To build the sample program `outsrv` from the source file `outsrv.sqb`, connecting to the sample database, enter:

```
bldrtn outsrv
```

If connecting to another database, also include the database name:

```
bldrtn outsrv database
```

The script file copies the shared library to the server in the path `sqllib/function`.

Once you build the routine shared library, `outsrv`, you can build the client application, `outcli`, that calls the routine within the library. You can build `outcli` using the script file `bldapp`.

To call the routine, run the sample client application by entering:

```
outcli database userid password
```

where

**database**
: Is the name of the database to which you want to connect. The name could be `sample`, or its remote alias, or some other name.

**userid**
: Is a valid user ID.

**password**
: Is a valid password for the user ID.

The client application accesses the shared library, `outsrv`, and executes the routine of the same name on the server database, and then returns the output to the client application.

**Related concepts:**

- "Building embedded SQL applications using the sample build script" on page 200

**Related tasks:**

- "Building IBM COBOL applications on AIX" on page 222

**Related reference:**
- "AIX IBM COBOL routine compile and link options" on page 354
- "COBOL samples" on page 373

**Related samples:**
- "bldrtn -- Builds AIX COBOL routines (stored procedures)"
- "embprep -- To prep and bind a COBOL embedded SQL sample on AIX"
- "outcli.sqb -- Call stored procedures using the SQLDA structure (IBM COBOL)"
- "outsrv.sqb -- Demonstrates stored procedures using the SQLDA structure (IBM COBOL)"

## Building UNIX Micro Focus COBOL routines

DB2 provides build scripts for compiling and linking Micro Focus COBOL embedded SQL and DB2 API programs. These are located in the `sqllib/samples/cobol_mf` directory, along with sample programs that can be built with these files.

The script, `bldrtn`, contains the commands to build routines (stored procedures). The script compiles the routine source file into a shared library that can be called by a client application.

The first parameter, $1, specifies the name of your source file. The script uses the source file name for the shared library name. The second parameter, $2, specifies the name of the database to which you want to connect. Since the shared library must be built in the same instance where the database resides, there are no parameters for user ID and password.

Only the first parameter, source file name, is required. Database name is optional. If no database name is supplied, the program uses the default `sample` database.

**Settings:**

Before building Micro Focus COBOL routines, you must run the following commands:

```
db2stop
db2set DB2LIBPATH=$LD_LIBRARY_PATH
db2set DB2ENVLIST="COBDIR LD_LIBRARY_PATH"
db2set
db2start
```

Ensure that `db2stop` stops the database. The last `db2set` command is issued to check your settings: make sure `DB2LIBPATH` and `DB2ENVLIST` are set correctly.

**Procedure:**

To build the sample program `outsrv` from the source file `outsrv.sqb`, if connecting to the sample database, enter:

```
bldrtn outsrv
```

If connecting to another database, also enter the database name:

```
bldrtn outsrv database
```

The script file copies the shared library to the server in the path `sqllib/function`.

Once you build the stored procedure `outsrv`, you can build the client application `outcli` that calls it. You can build `outcli` using the script file, `bldapp`.

To call the stored procedure, run the sample client application by entering:

```
outcli database userid password
```

where

**database**
: Is the name of the database to which you want to connect. The name could be `sample`, or its alias, or another name.

**userid**
: Is a valid user ID.

**password**
: Is a valid password for the user ID.

The client application accesses the shared library, `outsrv`, and executes the stored procedure function of the same name on the server database. The output is then returned to the client application.

**Related tasks:**

- "Building UNIX Micro Focus COBOL applications" on page 223

**Related reference:**

- "AIX Micro Focus COBOL routine compile and link options" on page 355
- "HP-UX Micro Focus COBOL routine compile and link options" on page 356
- "Linux Micro Focus COBOL routine compile and link options" on page 357
- "Solaris Micro Focus COBOL routine compile and link options" on page 357

**Related samples:**

- "bldrtn -- Builds AIX Micro Focus COBOL routines (stored procedures)"
- "bldrtn -- Builds HP-UX Micro Focus COBOL routines (stored procedures)"
- "bldrtn -- Builds Linux Micro Focus COBOL routines (stored procedures)"
- "bldrtn -- Builds Solaris Micro Focus COBOL routines (stored procedures)"
- "outcli.sqb -- Call stored procedures using the SQLDA structure (MF COBOL)"
- "outsrv.sqb -- Demonstrates stored procedures using the SQLDA structure (MF COBOL)"
- "embprep -- To prep and bind C/C++ and Micro Focus COBOL embedded SQL programs (C)"

## Building IBM COBOL routines on Windows

DB2 provides build scripts for compiling and linking DB2 API and embedded SQL programs in IBM COBOL. These are located in the `sqllib\samples\cobol` directory, along with sample programs that can be built with these files.

DB2 supports two precompilers for building IBM COBOL applications on Windows, the DB2 precompiler and the IBM COBOL precompiler. The default is the DB2 precompiler. The IBM COBOL precompiler can be selected by uncommenting the appropriate line in the batch file you are using. Precompilation with IBM COBOL is done by the compiler itself, using specific precompile options.

The batch file, `bldrtn.bat`, contains the commands to build embedded SQL routines (stored procedures). The batch file compiles the routines into a DLL on the server. It takes two parameters, represented inside the batch file by the variables `%1` and `%2`.

The first parameter, `%1`, specifies the name of your source file. The batch file uses the source file name, `%1`, for the DLL name. The second parameter, `%2`, specifies the name of the database to which you want to connect. Since the stored procedure must be built on the same instance where the database resides, there are no parameters for user ID and password.

Only the first parameter, source file name, is required. Database name is optional. If no database name is supplied, the program uses the default `sample` database.

If using the default DB2 precompiler, `bldrtn.bat` passes the parameters to the precompile and bind file, `embprep.bat`.

If using the IBM COBOL precompiler, `bldrtn.bat` copies the `.sqb` source file to a `.cbl` source file. The compiler performs the precompile on the `.cbl` source file with specific precompile options.

**Procedure:**

To build the sample program `outsrv` from the source file `outsrv.sqb`, connecting to the sample database, enter:

```
bldrtn outsrv
```

If connecting to another database, also include the database name:

```
bldrtn outsrv database
```

The batch file copies the DLL to the server in the path `sqllib\function`.

Once you build the DLL `outsrv`, you can build the client application `outcli` that calls the routine within the DLL (which has the same name as the DLL). You can build `outcli` using the batch file `bldapp.bat`.

To call the `outsrv` routine, run the sample client application by entering:

```
outcli database userid password
```

where

**database**
Is the name of the database to which you want to connect. The name could be `sample`, or its remote alias, or some other name.

**userid** Is a valid user ID.

**password**
Is a valid password for the user ID.

The client application accesses the DLL, `outsrv`, and executes the routine of the same name on the server database, and then returns the output to the client application.

**Related concepts:**

- "Building embedded SQL applications using the sample build script" on page 200

**Related reference:**
- "COBOL samples" on page 373
- "Windows IBM COBOL routine compile and link options" on page 358

**Related samples:**
- "bldrtn.bat -- Builds Windows VisualAge COBOL routines (stored procedures)"
- "embprep.bat -- To prep and bind a COBOL embedded SQL program on Windows"
- "outcli.sqb -- Call stored procedures using the SQLDA structure (IBM COBOL)"
- "outsrv.sqb -- Demonstrates stored procedures using the SQLDA structure (IBM COBOL)"

## Building Micro Focus COBOL routines on Windows

DB2 provides build scripts for compiling and linking DB2 API and embedded SQL programs in Micro Focus COBOL. These are located in the `sqllib\samples\cobol_mf` directory, along with sample programs that can be built with these files.

The batch file `bldrtn.bat` contains the commands to build embedded SQL routines (stored procedures). The batch file compiles the routines into a DLL on the server. The batch file takes two parameters, represented inside the batch file by the variables `%1` and `%2`.

The first parameter, `%1`, specifies the name of your source file. The batch file uses the source file name, `%1`, for the DLL name. The second parameter, `%2`, specifies the name of the database to which you want to connect. Since the stored procedure must be built on the same instance where the database resides, there are no parameters for user ID and password.

Only the first parameter, source file name, is required. Database name is optional. If no database name is supplied, the program uses the default `sample` database.

**Procedure:**

To build the sample program `outsrv` from the source file `outsrv.sqb`, if connecting to the sample database, enter:

```
bldrtn outsrv
```

If connecting to another database, also enter the database name:

```
bldrtn outsrv database
```

The script file copies the DLL to the server in the path `sqllib/function`.

Once you build the DLL, `outsrv`, you can build the client application, `outcli`, that calls the routine within the DLL (which has the same name as the DLL). You can build `outcli` using the batch file, `bldapp.bat`.

To call the `outsrv` routine, run the sample client application by entering:

```
outcli database userid password
```

where

**database**
: Is the name of the database to which you want to connect. The name could be `sample`, or its alias, or another name.

**userid** Is a valid user ID.

**password**
Is a valid password for the user ID.

The client application accesses the DLL, `outsrv`, which executes the routine of the same name on the server database. The output is then returned to the client application.

**Related concepts:**

- "Building embedded SQL applications using the sample build script" on page 200

**Related reference:**

- "COBOL samples" on page 373
- "Windows Micro Focus COBOL routine compile and link options" on page 359

**Related samples:**

- "bldrtn.bat -- Builds Windows Micro Focus Cobol routines (stored procedures)"
- "outcli.sqb -- Call stored procedures using the SQLDA structure (MF COBOL)"
- "outsrv.sqb -- Demonstrates stored procedures using the SQLDA structure (MF COBOL)"
- "embprep.bat -- Prep and binds a C/C++ or Micro Focus COBOL embedded SQL program on Windows"

## Part 3. Appendixes

# Appendix A. Embedded SQL samples

## C samples

UNIX directory: `sqllib/samples/c`. Windows directory: `sqllib\samples\c`.

File extensions: `.c` (no embedded SQL); `.sqc` (embedded SQL)

*Table 29. C sample program files*

| Type of sample | Sample program name | Program description |
|---|---|---|
| Client level | `cli_info.c` | How to get and set client level information. |
| | `clisnap.c` | How to capture a snapshot at the client level. |
| | `clisnapnew.c` | How to get a snapshot at the client level (using API). |
| Instance level | `inattach.c` | How to attach to/detach from an instance. |
| | `inauth.sqc` | How to display authorities at instance level. |
| | `ininfo.c` | How to get and set instance level information. |
| | `insnap.c` | How to capture a snapshot at the instance level. |
| | `insnapnew.c` | How to get a snapshot at instance level (using API). |
| | `instart.c` | How to stop and start the current local instance. |

Table 29. C sample program files (continued)

| Type of sample | Sample program name | Program description |
|---|---|---|
| Database level | `autostore.c` | How to use automatic storage capability for a database. |
| | `dbauth.sqc` | How to grant/display/revoke authorities at the database level |
| | `dbcfg.sqc` | How to configure database and database manager parameters. |
| | `dbconn.sqc` | How to connect and disconnect from a database. |
| | `dbcreate.c` | How to create and drop databases. |
| | `dbhistfile.sqc` | How to read and update a database recovery history file entry. |
| | `dbinfo.c` | How to get and set information at a database level. |
| | `dbinline.sqc` | How to use inline SQL procedure language. |
| | `dbinspec.sqc` | How to check architectural integrity with the DB2 API db2Inspect. |
| | `dblogconn.sqc` | How to read database log files asynchronously with a database connection. |
| | `dblognoconn.sqc` | How to read database log files asynchronously with no database connection. |
| | `dbmcon.sqc` | How to connect to and disconnect from multiple databases. |
| | `dbmcon1.h` | Header file for dbmcon1.sqc. |
| | `dbmcon1.sqc` | Support file for dbmcon.sqc. |
| | `dbmcon2.h` | Header file for dbmcon2.sqc. |
| | `dbmcon2.sqc` | Support file for dbmcon.sqc. |
| | `dbmigrat.c` | How to migrate a database. |
| | `dbpkg.sqc` | How to work with packages. |
| | `dbrec.sqc` | How to use the db2GetRecommendations API. |
| | `dbrecov.sqc` | How to recover a database. |
| | `dbredirect.sqc` | How to perform Redirected Restore of a database. |
| | `dbrestore.sqc` | How to restore a database from a backup. |
| | `dbrollfwd.sqc` | How to perform rollforward after a restore of a database. |
| | `dbsample.sqc` | How to create the sample database including Host and AS/400 tables and views. |
| | `dbsnap.c` | How to capture a snapshot at the database level. |
| | `dbsnapnew.c` | How to get a snapshot at database level (using API). |
| | `dbthrds.sqc` | How to use multiple context APIs on UNIX. |
| | `dbthrds.sqc` | How to use multiple context APIs on Windows. |

*Table 29. C sample program files (continued)*

| Type of sample | Sample program name | Program description |
|---|---|---|
| Table space level | `tscreate.sqc` | How to create and drop buffer pools and table spaces. |
| | `tsinfo.sqc` | How to get information at the table space level. |

Table 29. C sample program files (continued)

| Type of sample | Sample program name | Program description |
|---|---|---|
| Table level | getmessage.sqc | How to get error message in the required locale with token replacement. |
| | largerid.sqc | How to enable Large RIDs support on both new tables/tablespaces and existing tables/tablespaces. |
| | setintegrity.sqc | How to perform online SET INTEGRITY on a table. |
| | tbast.sqc | How to use a staging table for updating a deferred Automatic Summary Table. |
| | tbcompress.sqc | How to create tables with null and default value compression options. |
| | tbconstr.sqc | How to work with table constraints. |
| | tbcreate.sqc | How to create, alter and drop tables. |
| | tbident.sqc | How to use identity columns. |
| | tbinfo.sqc | How to get and set information at a table level. |
| | tbintrig.sqc | How to use an 'INSTEAD OF' trigger on a view. |
| | tbload.sqc | How to load into a partitioned database. |
| | tbloadcursor.sqc | How to load data returned from a SELECT statement into a table using CURSOR method or REMOTEFETCH media method. |
| | tbmerge.sqc | How to use the MERGE statement. |
| | tbmod.sqc | How to modify information in a table. |
| | tbmove.sqc | How to move a table data. |
| | tbonlineinx.sqc | How to create and reorganize indexes on a table. |
| | tbpriv.sqc | How to grant/display/revoke table level privileges. |
| | tbread.sqc | How to read information in a table. |
| | tbreorg.sqc | How to reorganize a table. |
| | tbrowcompress.sqc | How to perform row compression on a table. |
| | tbrunstats.sqc | How to perform runstats on a table. |
| | tbsavept.sqc | How to use external savepoints. |
| | tbsel.sqc | How to select from each of: insert, update, delete. |
| | tbselcreate.db2 | How to create the tables for the tbsel program. |
| | tbseldrop.db2 | How to drop the tables for the tbsel program. |
| | tbtemp.sqc | How to use a declared temporary table. |
| | tbtrig.sqc | How to use a trigger on a table. |
| | tbumqt.sqc | How to use user materialized query tables (summary tables). |

*Table 29. C sample program files (continued)*

| Type of sample | Sample program name | Program description |
|---|---|---|
| Data type level | `dtformat.sqc` | How to use load and import data format extensions. |
| | `dtlob.sqc` | How to read and write LOB data. |
| | `dtudt.sqc` | How to create, use, and drop user-defined distinct types. |
| DB2 function level | `fnuse.sqc` | How to use SQL functions. |
| Stored procedure level | `spcat` | Stored procedure catalog script for the spserver program. This script calls spdrop.db2 and spcreate.db2. |
| | `spcreate.db2` | CLP script to issue CREATE PROCEDURE statements. |
| | `spdrop.db2` | CLP script to drop stored procedures from the catalog. |
| | `spclient.sqc` | Client program used to call the server routines declared in spserver.sqc. |
| | `spserver.sqc` | Stored procedure routines built and run on the server. |
| UDF level | `udfcli.sqc` | Client application which calls the user-defined function in udfsrv.c, udfsrv.C. |
| | `udfsrv.c` | User-defined function ScalarUDF called by udfcli.sqc |
| | `udfemcli.sqc` | Client application which calls the embedded SQL user defined function library udfemsrv. |
| | `udfemsrv.sqc` | Embedded SQL User-defined function library called by udfemcli. |
| Other | `evm.sqc` | How to create and parse file, pipe, and table event monitors. |
| | `utilrecov.c` | Utilities for the backup, restore and log file samples. |
| | `utilsnap.c` | Utilities for the snapshot monitor samples. |

**Related concepts:**

- "Building embedded SQL applications using the sample build script" on page 200
- "Error-checking utilities" on page 202
- "Sample files" in *Samples Topics*

## COBOL samples

UNIX directories. IBM COBOL: `sqllib/samples/cobol`; Micro Focus COBOL: `sqllib/samples/cobol_mf`.

Windows directories. IBM COBOL: `sqllib\samples\cobol`; Micro Focus COBOL: `sqllib\samples\cobol_mf`.

**Note:** The COBOL samples are not structured in the DB2 level design used for the C, CLI, C++, C#, Java, Perl, PHP, Visual Basic ADO, and Visual Basic .NET samples.

*Table 30. COBOL DB2 API sample programs with no embedded SQL*

| Sample program | Included APIs |
|---|---|
| `checkerr.cbl` | • sqlaintp - Get Error Message<br>• sqlogstt - Get SQLSTATE Message |
| `client.cbl` | • sqleqryc - Query Client<br>• sqlesetc - Set Client |
| `d_dbconf.cbl` | • sqleatin - Attach<br>• sqledtin - Detach<br>• sqlfddb - Get Database Configuration Defaults |
| `d_dbmcon.cbl` | • sqleatin - Attach<br>• sqledtin - Detach<br>• sqlfdsys - Get Database Manager Configuration Defaults |
| `db_udcs.cbl` | • sqleatin - Attach<br>• sqlecrea - Create Database<br>• sqledrpd - Drop Database |
| `dbcat.cbl` | • sqlecadb - Catalog Database<br>• db2DbDirCloseScan - Close Database Directory Scan<br>• db2DbDirGetNextEntry - Get Next Database Directory Entry<br>• db2DbDirOpenScan - Open Database Directory Scan<br>• sqleuncd - Uncatalog Database |
| `dbcmt.cbl` | • sqledcgd - Change Database Comment<br>• db2DbDirCloseScan - Close Database Directory Scan<br>• db2DbDirGetNextEntry - Get Next Database Directory Entry<br>• db2DbDirOpenScan - Open Database Directory Scan<br>• sqleisig - Install Signal Handler |
| `dbconf.cbl` | • sqleatin - Attach<br>• sqlecrea - Create Database<br>• sqledrpd - Drop Database<br>• sqlfrdb - Reset Database Configuration<br>• sqlfudb - Update Database Configuration<br>• sqlfxdb - Get Database Configuration |
| `dbinst.cbl` | • sqleatcp - Attach and Change Password<br>• sqleatin - Attach<br>• sqledtin - Detach<br>• sqlegins - Get Instance |
| `dbmconf.cbl` | • sqleatin - Attach<br>• sqledtin - Detach<br>• sqlfrsys - Reset Database Manager Configuration<br>• sqlfusys - Update Database Manager Configuration<br>• sqlfxsys - Get Database Manager Configuration |

*Table 30. COBOL DB2 API sample programs with no embedded SQL (continued)*

| Sample program | Included APIs |
|---|---|
| dbsnap.cbl | • sqleatin - Attach<br>• sqlmonss - Get Snapshot |
| dbstart.cbl | • sqlepstart - Start Database Manager |
| dbstop.cbl | • sqlefrce - Force Application<br>• sqlepstp - Stop Database Manager |
| dcscat.cbl | • sqlegdad - Catalog DCS Database<br>• sqlegdcl - Close DCS Directory Scan<br>• sqlegdel - Uncatalog DCS Database<br>• sqlegdge - Get DCS Directory Entry for Database<br>• sqlegdgt - Get DCS Directory Entries<br>• sqlegdsc - Open DCS Directory Scan |
| ebcdicdb.cbl | • sqleatin - Attach<br>• sqlecrea - Create Database<br>• sqledrpd - Drop Database |
| migrate.cbl | • sqlemgdb - Migrate Database |
| monreset.cbl | • sqleatin - Attach<br>• sqlmrset - Reset Monitor |
| monsz.cbl | • sqleatin - Attach<br>• sqlmonss - Get Snapshot<br>• sqlmonsz - Estimate Size Required for sqlmonss() Output Buffer |
| nodecat.cbl | • sqlectnd - Catalog Node<br>• sqlencls - Close Node Directory Scan<br>• sqlengne - Get Next Node Directory Entry<br>• sqlenops - Open Node Directory Scan<br>• sqleuncn - Uncatalog Node |
| restart.cbl | • sqlerstd - Restart Database |
| setact.cbl | • sqlesact - Set Accounting String |
| sws.cbl | • sqleatin - Attach<br>• sqlmon - Get/Update Monitor Switches |

*Table 31. COBOL DB2 API embedded SQL sample programs*

| Sample program | Included APIs |
|---|---|
| dbauth.sqb | • sqluadau - Get Authorizations |
| dbstat.sqb | • db2Reorg - Reorganize Table<br>• db2Runstats - Run Statistics |
| expsamp.sqb | • db2Export - Export<br>• sqluimpr - Import |
| impexp.sqb | • db2Export - Export<br>• sqluimpr - Import |

*Table 31. COBOL DB2 API embedded SQL sample programs (continued)*

| Sample program | Included APIs |
|---|---|
| `loadqry.sqb` | • db2LoadQuery - Load Query |
| `rebind.sqb` | • sqlarbnd - Rebind |
| `tabscont.sqb` | • sqlbctcq - Close Tablespace Container Query<br>• sqlbftcq - Fetch Tablespace Container Query<br>• sqlbotcq - Open Tablespace Container Query<br>• sqlbtcq - Tablespace Container Query<br>• sqlefmem - Free Memory |
| `tabspace.sqb` | • sqlbctsq - Close Tablespace Query<br>• sqlbftpq - Fetch Tablespace Query<br>• sqlbgtss - Get Tablespace Statistics<br>• sqlbmtsq - Tablespace Query<br>• sqlbotsq - Open Tablespace Query<br>• sqlbstpq - Single Tablespace Query<br>• sqlefmem - Free Memory |
| `tload.sqb` | • db2Export - Export<br>• sqluload - Load<br>• sqluvqdp - Quiesce Tablespaces for Table |
| `tspace.sqb` | • sqlbctcq - Close Tablespace Container Query<br>• sqlbctsq - Close Tablespace Query<br>• sqlbftcq - Fetch Tablespace Container Query<br>• sqlbftpq - Fetch Tablespace Query<br>• sqlbgtss - Get Tablespace Statistics<br>• sqlbmtsq - Tablespace Query<br>• sqlbotcq - Open Tablespace Container Query<br>• sqlbotsq - Open Tablespace Query<br>• sqlbstpq - Single Tablespace Query<br>• sqlbstsc - Set Tablespace Containers<br>• sqlbtcq - Tablespace Container Query<br>• sqlefmem - Free Memory |

*Table 32. COBOL Embedded SQL sample programs with No DB2 APIs*

| Sample program name | Program description |
|---|---|
| `advsql.sqb` | Demonstrates the use of advanced SQL expressions like CASE, CAST, and scalar full selects. |
| `cursor.sqb` | Demonstrates the use of a cursor using static SQL. |
| `delet.sqb` | Demonstrates static SQL to delete items from a database. |
| `dynamic.sqb` | Demonstrates the use of a cursor using dynamic SQL. |
| `joinsql.sqb` | Demonstrates using advanced SQL join expressions. |
| `lobeval.sqb` | Demonstrates the use of LOB locators and defers the evaluation of the actual LOB data. |
| `lobfile.sqb` | Demonstrates the use of LOB file handles. |
| `lobloc.sqb` | Demonstrates the use of LOB locators. |

Table 32. COBOL Embedded SQL sample programs with No DB2 APIs (continued)

| Sample program name | Program description |
|---|---|
| openftch.sqb | Demonstrates fetching, updating, and deleting rows using static SQL. |
| static.sqb | Demonstrates static SQL to retrieve information. |
| tabsql.sqb | Demonstrates the use of advanced SQL table expressions. |
| trigsql.sqb | Demonstrates using advanced SQL triggers and constraints. |
| updat.sqb | Demonstrates static SQL to update a database. |
| varinp.sqb | Demonstrates variable input to Embedded Dynamic SQL statement calls using parameter markers. |

**Related concepts:**

- “Building embedded SQL applications using the sample build script” on page 200
- “Error-checking utilities” on page 202
- “Sample files” in *Samples Topics*

## REXX samples

AIX directory: `sqllib/samples/rexx`. Windows directory: `sqllib\samples\rexx`.

Table 33. REXX sample program files.

| Sample file name | File description |
|---|---|
| blobfile.cmd | Demonstrates Binary Large Object (BLOB) manipulation. |
| chgisl.cmd | Demonstrates the CHANGE ISOLATION LEVEL API. |
| client.cmd | Demonstrates the SET CLIENT and QUERY CLIENT APIs. |
| d_dbconf.cmd | Demonstrates the API: GET DATABASE CONFIGURATION DEFAULTS |
| d_dbmcon.cmd | Demonstrates the API: GET DATABASE MANAGER CONFIGURATION DEFAULTS |
| db_udcs.cmd | Demonstrates the CREATE DATABASE and DROP DATABASE APIs to simulate the collating behavior of a DB2 for MVS/ESA CCSID 500 (EBCDIC International) collating sequence |
| dbauth.cmd | Demonstrates the GET AUTHORIZATIONS API |
| dbcat.cmd | Demonstrates the following APIs:<br>CATALOG DATABASE<br>CLOSE DATABASE DIRECTORY SCAN<br>GET NEXT DATABASE DIRECTORY ENTRY<br>OPEN DATABASE DIRECTORY SCAN<br>UNCATALOG DATABASE |
| dbcmt.cmd | Demonstrates the following APIs:<br>CHANGE DATABASE COMMENT<br>GET ERROR MESSAGE<br>INSTALL SIGNAL HANDLER |
| dbconf.cmd | Demonstrates the following APIs:<br>CREATE DATABASE<br>DROP DATABASE<br>GET DATABASE CONFIGURATION<br>RESET DATABASE CONFIGURATION<br>UPDATE DATABASE CONFIGURATION |

*Table 33. REXX sample program files. (continued)*

| Sample file name | File description |
|---|---|
| `dbinst.cmd` | Demonstrates the following APIs:<br>`ATTACH TO INSTANCE`<br>`DETACH FROM INSTANCE`<br>`GET INSTANCE` |
| `dbmconf.cmd` | Demonstrates the following APIs:<br>`GET DATABASE MANAGER CONFIGURATION`<br>`RESET DATABASE MANAGER CONFIGURATION`<br>`UPDATE DATABASE MANAGER CONFIGURATION` |
| `dbstart.cmd` | Demonstrates the START DATABASE MANAGER API |
| `dbstat.cmd` | Demonstrates the following APIs:<br>`REORGANIZE TABLE`<br>`RUN STATISTICS` |
| `dbstop.cmd` | Demonstrates the following APIs:<br>`FORCE USERS`<br>`STOP DATABASE MANAGER` |
| `dcscat.cmd` | Demonstrates the following APIs:<br>`ADD DCS DIRECTORY ENTRY`<br>`CLOSE DCS DIRECTORY SCAN`<br>`GET DCS DIRECTORY ENTRY FOR DATABASE`<br>`GET DCS DIRECTORY ENTRIES`<br>`OPEN DCS DIRECTORY SCAN`<br>`UNCATALOG DCS DIRECTORY ENTRY` |
| `dynamic.cmd` | Demonstrates the use of a ″CURSOR″ using dynamic SQL |
| `ebcdicdb.cmd` | Demonstrates the CREATE DATABASE and DROP DATABASE APIs to simulate the collating behavior of a DB2 for MVS/ESA CCSID 037 (EBCDIC US English) collating sequence |
| `impexp.cmd` | Demonstrates the EXPORT and IMPORT APIs |
| `lobeval.cmd` | Demonstrates deferring the evaluation of a LOB within a database |
| `lobfile.cmd` | Demonstrates the use of LOB file handles |
| `lobloc.cmd` | Demonstrates the use of LOB locators |
| `lobval.cmd` | Demonstrates the use of LOBs |
| `migrate.cmd` | Demonstrates the MIGRATE DATABASE API |
| `nodecat.cmd` | Demonstrates the following APIs:<br>`CATALOG NODE`<br>`CLOSE NODE DIRECTORY SCAN`<br>`GET NEXT NODE DIRECTORY ENTRY`<br>`OPEN NODE DIRECTORY SCAN`<br>`UNCATALOG NODE` |
| `quitab.cmd` | Demonstrates the API: QUIESCE TABLESPACES FOR TABLE |
| `rechist.cmd` | Demonstrates the following APIs:<br>`CLOSE RECOVERY HISTORY FILE SCAN`<br>`GET NEXT RECOVERY HISTORY FILE ENTRY`<br>`OPEN RECOVER HISTORY FILE SCAN`<br>`PRUNE RECOVERY HISTORY FILE ENTRY`<br>`UPDATE RECOVERY HISTORY FILE ENTRY` |
| `restart.cmd` | Demonstrates the RESTART DATABASE API |
| `sqlecsrx.cmd` | An example of a collating sequence |
| `updat.cmd` | Uses dynamic SQL to update a database |

**Related concepts:**
- "Sample files" in *Samples Topics*

**Related tasks:**
- "Building Object REXX applications on Windows" on page 241

# Appendix B. DB2 Database technical information

## Overview of the DB2 technical information

DB2 technical information is available through the following tools and methods:

- DB2 Information Center
  - Topics
  - Help for DB2 tools
  - Sample programs
  - Tutorials
- DB2 books
  - PDF files (downloadable)
  - PDF files (from the DB2 PDF CD)
  - printed books
- Command line help
  - Command help
  - Message help
- Sample programs

IBM periodically makes documentation updates available. If you access the online version on the DB2 Information Center at ibm.com®, you do not need to install documentation updates because this version is kept up-to-date by IBM. If you have installed the DB2 Information Center, it is recommended that you install the documentation updates. Documentation updates allow you to update the information that you installed from the *DB2 Information Center CD* or downloaded from Passport Advantage as new information becomes available.

**Note:** The DB2 Information Center topics are updated more frequently than either the PDF or the hard-copy books. To get the most current information, install the documentation updates as they become available, or refer to the DB2 Information Center at ibm.com.

You can access additional DB2 technical information such as technotes, white papers, and Redbooks™ online at ibm.com. Access the DB2 Information Management software library site at http://www.ibm.com/software/data/sw-library/.

### Documentation feedback

We value your feedback on the DB2 documentation. If you have suggestions for how we can improve the DB2 documentation, send an e-mail to db2docs@ca.ibm.com. The DB2 documentation team reads all of your feedback, but cannot respond to you directly. Provide specific examples wherever possible so that we can better understand your concerns. If you are providing feedback on a specific topic or help file, include the topic title and URL.

Do not use this e-mail address to contact DB2 Customer Support. If you have a DB2 technical issue that the documentation does not resolve, contact your local IBM service center for assistance.

**Related concepts:**

- "Features of the DB2 Information Center" in *Online DB2 Information Center*
- "Sample files" in *Samples Topics*

**Related tasks:**

- "Invoking command help from the command line processor" in *Command Reference*
- "Invoking message help from the command line processor" in *Command Reference*
- "Updating the DB2 Information Center installed on your computer or intranet server" on page 387

**Related reference:**

- "DB2 technical library in hardcopy or PDF format" on page 382

## DB2 technical library in hardcopy or PDF format

The following tables describe the DB2 library available from the IBM Publications Center at www.ibm.com/shop/publications/order. DB2 Version 9 manuals in PDF format can be downloaded from www.ibm.com/software/data/db2/udb/support/manualsv9.html.

Although the tables identify books available in print, the books might not be available in your country or region.

The information in these books is fundamental to all DB2 users; you will find this information useful whether you are a programmer, a database administrator, or someone who works with DB2 Connect or other DB2 products.

*Table 34. DB2 technical information*

| Name | Form Number | Available in print |
|---|---|---|
| *Administration Guide: Implementation* | SC10-4221 | Yes |
| *Administration Guide: Planning* | SC10-4223 | Yes |
| *Administrative API Reference* | SC10-4231 | Yes |
| *Administrative SQL Routines and Views* | SC10-4293 | No |
| *Call Level Interface Guide and Reference, Volume 1* | SC10-4224 | Yes |
| *Call Level Interface Guide and Reference, Volume 2* | SC10-4225 | Yes |
| *Command Reference* | SC10-4226 | No |
| *Data Movement Utilities Guide and Reference* | SC10-4227 | Yes |
| *Data Recovery and High Availability Guide and Reference* | SC10-4228 | Yes |
| *Developing ADO.NET and OLE DB Applications* | SC10-4230 | Yes |
| *Developing Embedded SQL Applications* | SC10-4232 | Yes |

Table 34. DB2 technical information (continued)

| Name | Form Number | Available in print |
|---|---|---|
| *Developing SQL and External Routines* | SC10-4373 | No |
| *Developing Java Applications* | SC10-4233 | Yes |
| *Developing Perl and PHP Applications* | SC10-4234 | No |
| *Getting Started with Database Application Development* | SC10-4252 | Yes |
| *Getting started with DB2 installation and administration on Linux and Windows* | GC10-4247 | Yes |
| *Message Reference Volume 1* | SC10-4238 | No |
| *Message Reference Volume 2* | SC10-4239 | No |
| *Migration Guide* | GC10-4237 | Yes |
| *Net Search Extender Administration and User's Guide* **Note:** HTML for this document is not installed from the HTML documentation CD. | SH12-6842 | Yes |
| *Performance Guide* | SC10-4222 | Yes |
| *Query Patroller Administration and User's Guide* | GC10-4241 | Yes |
| *Quick Beginnings for DB2 Clients* | GC10-4242 | No |
| *Quick Beginnings for DB2 Servers* | GC10-4246 | Yes |
| *Spatial Extender and Geodetic Data Management Feature User's Guide and Reference* | SC18-9749 | Yes |
| *SQL Guide* | SC10-4248 | Yes |
| *SQL Reference, Volume 1* | SC10-4249 | Yes |
| *SQL Reference, Volume 2* | SC10-4250 | Yes |
| *System Monitor Guide and Reference* | SC10-4251 | Yes |
| *Troubleshooting Guide* | GC10-4240 | No |
| *Visual Explain Tutorial* | SC10-4319 | No |
| *What's New* | SC10-4253 | Yes |
| *XML Extender Administration and Programming* | SC18-9750 | Yes |
| *XML Guide* | SC10-4254 | Yes |
| *XQuery Reference* | SC18-9796 | Yes |

Table 35. DB2 Connect-specific technical information

| Name | Form Number | Available in print |
|---|---|---|
| *DB2 Connect User's Guide* | SC10-4229 | Yes |

Table 35. DB2 Connect-specific technical information (continued)

| Name | Form Number | Available in print |
|---|---|---|
| *Quick Beginnings for DB2 Connect Personal Edition* | GC10-4244 | Yes |
| *Quick Beginnings for DB2 Connect Servers* | GC10-4243 | Yes |

Table 36. WebSphere® Information Integration technical information

| Name | Form Number | Available in print |
|---|---|---|
| *WebSphere Information Integration: Administration Guide for Federated Systems* | SC19-1020 | Yes |
| *WebSphere Information Integration: ASNCLP Program Reference for Replication and Event Publishing* | SC19-1018 | Yes |
| *WebSphere Information Integration: Configuration Guide for Federated Data Sources* | SC19-1034 | No |
| *WebSphere Information Integration: SQL Replication Guide and Reference* | SC19-1030 | Yes |

**Note:** The DB2 Release Notes provide additional information specific to your product's release and fix pack level. For more information, see the related links.

**Related concepts:**

- "Overview of the DB2 technical information" on page 381
- "About the Release Notes" in *Release notes*

**Related tasks:**

- "Ordering printed DB2 books" on page 384

## Ordering printed DB2 books

If you require printed DB2 books, you can buy them online in many but not all countries or regions. You can always order printed DB2 books from your local IBM representative. Keep in mind that some softcopy books on the *DB2 PDF Documentation* CD are unavailable in print. For example, neither volume of the *DB2 Message Reference* is available as a printed book.

Printed versions of many of the DB2 books available on the DB2 PDF Documentation CD can be ordered for a fee from IBM. Depending on where you are placing your order from, you may be able to order books online, from the IBM Publications Center. If online ordering is not available in your country or region, you can always order printed DB2 books from your local IBM representative. Note that not all books on the DB2 PDF Documentation CD are available in print.

**Note:** The most up-to-date and complete DB2 documentation is maintained in the DB2 Information Center at http://publib.boulder.ibm.com/infocenter/db2help/.

**Procedure:**

To order printed DB2 books:

- To find out whether you can order printed DB2 books online in your country or region, check the IBM Publications Center at http://www.ibm.com/shop/publications/order. You must select a country, region, or language to access publication ordering information and then follow the ordering instructions for your location.
- To order printed DB2 books from your local IBM representative:
  - Locate the contact information for your local representative from one of the following Web sites:
    - The IBM directory of world wide contacts at www.ibm.com/planetwide
    - The IBM Publications Web site at http://www.ibm.com/shop/publications/order. You will need to select your country, region, or language to the access appropriate publications home page for your location. From this page, follow the ″About this site″ link.
  - When you call, specify that you want to order a DB2 publication.
  - Provide your representative with the titles and form numbers of the books that you want to order.

**Related concepts:**

- "Overview of the DB2 technical information" on page 381

**Related reference:**

- "DB2 technical library in hardcopy or PDF format" on page 382

## Displaying SQL state help from the command line processor

DB2 returns an SQLSTATE value for conditions that could be the result of an SQL statement. SQLSTATE help explains the meanings of SQL states and SQL state class codes.

**Procedure:**

To invoke SQL state help, open the command line processor and enter:

```
? sqlstate or ? class code
```

where *sqlstate* represents a valid five-digit SQL state and *class code* represents the first two digits of the SQL state.

For example, `? 08003` displays help for the 08003 SQL state, and `? 08` displays help for the 08 class code.

**Related tasks:**

- "Invoking command help from the command line processor" in *Command Reference*
- "Invoking message help from the command line processor" in *Command Reference*

## Accessing different versions of the DB2 Information Center

For DB2 Version 9 topics, the DB2 Information Center URL is http://publib.boulder.ibm.com/infocenter/db2luw/v9/.

For DB2 Version 8 topics, go to the Version 8 Information Center URL at: http://publib.boulder.ibm.com/infocenter/db2luw/v8/.

**Related tasks:**

- "Setting up access to DB2 contextual help and documentation" in *Administration Guide: Implementation*

## Displaying topics in your preferred language in the DB2 Information Center

The DB2 Information Center attempts to display topics in the language specified in your browser preferences. If a topic has not been translated into your preferred language, the DB2 Information Center displays the topic in English.

**Procedure:**

To display topics in your preferred language in the Internet Explorer browser:

1. In Internet Explorer, click the **Tools** —> **Internet Options** —> **Languages...** button. The Language Preferences window opens.
2. Ensure your preferred language is specified as the first entry in the list of languages.
   - To add a new language to the list, click the **Add...** button.

     **Note:** Adding a language does not guarantee that the computer has the fonts required to display the topics in the preferred language.
   - To move a language to the top of the list, select the language and click the **Move Up** button until the language is first in the list of languages.
3. Clear the browser cache and then refresh the page to display the DB2 Information Center in your preferred language.

To display topics in your preferred language in a Firefox or Mozilla browser:

1. Select the **Tools** —> **Options** —> **Languages** button. The Languages panel is displayed in the Preferences window.
2. Ensure your preferred language is specified as the first entry in the list of languages.
   - To add a new language to the list, click the **Add...** button to select a language from the Add Languages window.
   - To move a language to the top of the list, select the language and click the **Move Up** button until the language is first in the list of languages.
3. Clear the browser cache and then refresh the page to display the DB2 Information Center in your preferred language.

On some browser and operating system combinations, you might have to also change the regional settings of your operating system to the locale and language of your choice.

**Related concepts:**

- "Overview of the DB2 technical information" on page 381

## Updating the DB2 Information Center installed on your computer or intranet server

If you have a locally-installed DB2 Information Center, updated topics can be available for download. The 'Last updated' value found at the bottom of most topics indicates the current level for that topic.

To determine if there is an update available for the entire DB2 Information Center, look for the 'Last updated' value on the Information Center home page. Compare the value in your locally installed home page to the date of the most recent downloadable update at http://www.ibm.com/software/data/db2/udb/support/icupdate.html. You can then update your locally-installed Information Center if a more recent downloadable update is available.

Updating your locally-installed DB2 Information Center requires that you:

1. Stop the DB2 Information Center on your computer, and restart the Information Center in stand-alone mode. Running the Information Center in stand-alone mode prevents other users on your network from accessing the Information Center, and allows you to download and apply updates.
2. Use the Update feature to determine if update packages are available from IBM.

   **Note:** Updates are also available on CD. For details on how to configure your Information Center to install updates from CD, see the related links.

   If update packages are available, use the Update feature to download the packages. (The Update feature is only available in stand-alone mode.)
3. Stop the stand-alone Information Center, and restart the DB2 Information Center service on your computer.

**Procedure:**

To update the DB2 Information Center installed on your computer or intranet server:

1. Stop the DB2 Information Center service.
   - On Windows, click **Start → Control Panel → Administrative Tools → Services**. Then right-click on **DB2 Information Center** service and select **Stop**.
   - On Linux, enter the following command:

     ```
     /etc/init.d/db2icdv9 stop
     ```
2. Start the Information Center in stand-alone mode.
   - On Windows:
     a. Open a command window.
     b. Navigate to the path where the Information Center is installed. By default, the DB2 Information Center is installed in the `C:\Program Files\IBM\DB2 Information Center\Version 9` directory.
     c. Run the `help_start.bat` file using the fully qualified path for the DB2 Information Center:

        ```
        <DB2 Information Center dir>\doc\bin\help_start.bat
        ```
   - On Linux:

   a. Navigate to the path where the Information Center is installed. By default, the DB2 Information Center is installed in the /opt/ibm/db2ic/V9 directory.
   b. Run the help_start script using the fully qualified path for the DB2 Information Center:

      ```
      <DB2 Information Center dir>/doc/bin/help_start
      ```

   The systems default Web browser launches to display the stand-alone Information Center.

3. Click the Update button ( ). On the right hand panel of the Information Center, click **Find Updates**. A list of updates for existing documentation displays.
4. To initiate the download process, check the selections you want to download, then click **Install Updates**.
5. After the download and installation process has completed, click **Finish**.
6. Stop the stand-alone Information Center.
   - On Windows, run the help_end.bat file using the fully qualified path for the DB2 Information Center:

     ```
     <DB2 Information Center dir>\doc\bin\help_end.bat
     ```

     **Note:** The help_end batch file contains the commands required to safely terminate the processes that were started with the help_start batch file. Do not use Ctrl-C or any other method to terminate help_start.bat.
   - On Linux, run the help_end script using the fully qualified path for the DB2 Information Center:

     ```
     <DB2 Information Center dir>/doc/bin/help_end
     ```

     **Note:** The help_end script contains the commands required to safely terminate the processes that were started with the help_start script. Do not use any other method to terminate the help_start script.
7. Restart the DB2 Information Center service.
   - On Windows, click **Start** → **Control Panel** → **Administrative Tools** → **Services**. Then right-click on **DB2 Information Center** service and select **Start**.
   - On Linux, enter the following command:

     ```
     /etc/init.d/db2icdv9 start
     ```

The updated DB2 Information Center displays the new and updated topics.

**Related concepts:**
- "DB2 Information Center installation options" in *Quick Beginnings for DB2 Servers*

**Related tasks:**
- "Installing the DB2 Information Center using the DB2 Setup wizard (Linux)" in *Quick Beginnings for DB2 Servers*
- "Installing the DB2 Information Center using the DB2 Setup wizard (Windows)" in *Quick Beginnings for DB2 Servers*

## DB2 tutorials

The DB2 tutorials help you learn about various aspects of DB2 products. Lessons provide step-by-step instructions.

**Before you begin:**

You can view the XHTML version of the tutorial from the Information Center at http://publib.boulder.ibm.com/infocenter/db2help/.

Some lessons use sample data or code. See the tutorial for a description of any prerequisites for its specific tasks.

**DB2 tutorials:**

To view the tutorial, click on the title.

*Native XML data store*
: Set up a DB2 database to store XML data and to perform basic operations with the native XML data store.

*Visual Explain Tutorial*
: Analyze, optimize, and tune SQL statements for better performance using Visual Explain.

**Related concepts:**

- "Visual Explain overview" in *Administration Guide: Implementation*

## DB2 troubleshooting information

A wide variety of troubleshooting and problem determination information is available to assist you in using DB2 products.

**DB2 documentation**
: Troubleshooting information can be found in the DB2 Troubleshooting Guide or the Support and Troubleshooting section of the DB2 Information Center. There you will find information on how to isolate and identify problems using DB2 diagnostic tools and utilities, solutions to some of the most common problems, and other advice on how to solve problems you might encounter with your DB2 products.

**DB2 Technical Support Web site**
: Refer to the DB2 Technical Support Web site if you are experiencing problems and want help finding possible causes and solutions. The Technical Support site has links to the latest DB2 publications, TechNotes, Authorized Program Analysis Reports (APARs or bug fixes), fix packs, and other resources. You can search through this knowledge base to find possible solutions to your problems.

  Access the DB2 Technical Support Web site at http://www.ibm.com/software/data/db2/udb/support.html

**Related concepts:**

- "Introduction to problem determination" in *Troubleshooting Guide*
- "Overview of the DB2 technical information" on page 381

## Terms and Conditions

# Appendix C. Notices

IBM may not offer the products, services, or features discussed in this document in all 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
Armonk, NY 10504-1785
U.S.A.

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country/region or send inquiries, in writing, to:

IBM World Trade Asia Corporation
Licensing
2-31 Roppongi 3-chome, Minato-ku
Tokyo 106, Japan

**The following paragraph does not apply to the United Kingdom or any other country/region where such provisions are inconsistent with local law:** 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 states 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 Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product, and use of those Web sites 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.

## Trademarks

# Index

# Contacting IBM

To contact IBM in your country or region, check the IBM Directory of Worldwide Contacts at http://www.ibm.com/planetwide

To learn more about DB2 products, go to http://www.ibm.com/software/data/db2/.

Printed in USA

SC10-4232-00

Spine information:

IBM DB2 **DB2 Version 9** **Developing Embedded SQL Applications**