Db2 11.1 for Linux, UNIX, and Windows

Developing embedded SQL and XQuery database applications

Db2 11.1 for Linux, UNIX, and Windows

Developing embedded SQL and XQuery database applications

Notice regarding this document

This document in PDF form is provided as a courtesy to customers who have requested documentation in this format. It is provided As-Is without warranty or maintenance commitment.

Contents

Figures

Tables

Introduction to embedded SQL

Embedded SQL applications connect to databases and execute embedded SQL statements. The embedded SQL statements are contained in a package that must be bound to the target database server.

You can develop embedded SQL applications for the Db2® database in the following host programming languages: C, C++, and COBOL.

Building embedded SQL applications involves two prerequisite steps before 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 information:

Installing and configuring Optim Performance Manager Extended Insight

Embedding SQL statements in a host language

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

The elements you must create when you write an embedded SQL application include:

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, which consists of 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.

Embedded SQL statements in C and C++ applications

Before you can use the SQL statements, you must set up and enable your application to support embedded SQL.

Embedded SQL C and C++ applications consist of three main elements to setup and issue 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, which consists of 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 issue an SQL statement statically within a C application, you need to include a `EXEC SQL` statement within your application code:

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

The following example demonstrates how to issue 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 issued 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 sample code:

  ```
  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.

Embedded SQL statements in FORTRAN applications

You can include embedded SQL statements in FORTRAN applications. Before you can use the SQL statements, you must setup and enable your application to support embedded SQL.

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

Correct FORTRAN Element 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 will then be 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 not valid 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.

Embedded SQL statements in COBOL applications

You can include embedded SQL statements in COBOL applications. Before you can use the SQL statements, you must set up and enable your application to support embedded SQL.

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

Correct COBOL Element 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 seem 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.

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 code 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.

Supported development software for embedded SQL applications

Before you begin writing embedded SQL applications, you must determine if your development software is supported. The operating system that you are developing for determines which compilers, interpreters, and development software you must use.

Db2 database systems support compilers, interpreters, and related development software for embedded SQL applications in the following operating systems:

- AIX®
- Linux
- 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

Setting up the embedded SQL development environment

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

Before you begin

- Db2 data server installed on a supported platform
- Db2 client installed
- Supported embedded SQL application development software installed - see "Supported embedded SQL application development software installed" in *Getting Started with Database Application Development*

About this task

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.

Designing embedded SQL applications

When designing embedded SQL applications you must use static or dynamic executed SQL statements.

There are two types of static SQL statements: 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.

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 to start them. You can use the Db2 APIs to perform the Db2 administrative functions from within your application program. For example, to re-create 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 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.

However, BUDGET, 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.

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 issued 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 issued 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;
```

Dynamic SQL in embedded SQL programs

An example of a dynamically issued 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;
```

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 text that is processed dynamically.

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 (;).

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

There are several factors that you must consider before determining whether to issue a static or dynamic SQL statement in an embedded SQL application.

The following table lists the considerations associated with use of static and dynamic SQL statements.

Note: These are general suggestions only. Your application 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
Uniformity of data being queried or operated upon by the SQL statement	
• Uniform data distribution	• Static
• Slight non-uniformity	• Either
• Highly non-uniform distribution	• Dynamic
Quantity of range predicates within the query	
• Few	• Static
• Some	• Either
• Many	• Dynamic
Likelihood of repeated SQL statement execution	
• Runs many times (10 or more times)	• Either
• Runs a few times (less than 10 times)	• Either
• Runs once	• Static
Nature of Query	
• Random	• Dynamic
• Permanent	• Either
Types of SQL statements (DML/DDL/DCL)	
• Transaction Processing (DML Only)	• Either
• Mixed (DML and DDL - DDL affects packages)	• Dynamic
• Mixed (DML and DDL - DDL does not affect packages)	• Either
Frequency with which the `RUNSTATS` command is issued	
• Infrequently	• Static
• Regularly	• Either
• Frequently	• 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 slower due to the increased resource use associated with 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 issued 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 are times when it does not matter whether 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 issued dynamically that there might be one statement that might more suitably be issued as static SQL. In such a case, to be consistent in your coding, it might make sense to issue that one statement dynamically too. Note that the considerations in the previous table are listed roughly in order of importance.

Do not assume that a static version of an SQL statement is always faster than the equivalent dynamic statement. In some cases, static SQL is faster because of the resource use required to prepare the dynamic statement. In other cases, the same statement prepared dynamically issues 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
   - Simple SQL statements

   Simple SQL statements without host variables perform well from a performance perspective in that there is no runtime performance increase, and the Db2 optimizer 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 statement avoids the runtime resource usage associated with the PREPARE and catalog locks acquired during statement compilation. Unfortunately, the full power of the optimizer cannot be used 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 on-demand queries. From the CLP, SQL statements can only be issued 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.

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 issue 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 runtime 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 issued. 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.

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

You can build embedded SQL applications on both 32-bit and 64-bit operating systems. However, there are separate building and running considerations. Build scripts contain a check to determine the bit width. If the bit width 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 later in this section. 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 IPF
- Linux on x64
- Linux on POWER®
- Linux on System z®
- 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 System z.

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.

Restrictions on embedded SQL applications

Each supported host language has its own set of limitations and specifications.

C/C++ makes use of a sequence of three characters called trigraphs to overcome the limitations of displaying certain special characters. COBOL has a set of rules to aid in the use of object oriented COBOL applications. FORTRAN has areas of interest which can affect the precompiling processes whereas REXX is confined in certain areas such as language support.

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. You can enter these characters into a C or C++ source program by 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 following trigraphs listed might occur elsewhere in a C or C++ source program:

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

Restrictions on using COBOL to program embedded SQL applications

The restrictions for API calls in COBOL applications.

Restrictions for API calls in COBOL applications include:

- 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 be used 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.

Restrictions on using FORTRAN to program embedded SQL applications

Embedded SQL support for FORTRAN are stabilized in Db2 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 afterDb2 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.

Restrictions on using REXX to program embedded SQL applications

Restrictions on embedded SQL applications created using REXX limit the type of SQL statement that you can use, and some languages are not supported.

Following are the restrictions for embedded SQL in REXX applications:

- Embedded SQL support for REXX stabilized in Db2 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.

Recommendations for developing embedded SQL applications with XML and XQuery

If you are developing embedded SQL applications that use XML and XQuery data, you must consider if all the required data is available, and what type of data it is.

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 recommendations and restrictions 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 issue XQuery expressions through the XMLQUERY function.
- An XQuery expression can be dynamically issued by pre-pending the expression with the string "XQUERY".

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

Note:

1. The CLI automatically uses multiple contexts to achieve thread-safe, concurrent database access on platforms that support multi-threading. While not recommended, 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 database connections.

   When this limit is reached, Db2 database systems return SQLCODE -1224 on an SQL CONNECT. Db2 Connect also has the 10-connection limitation if local users are running two-phase commit with a TP Monitor (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 database systems, follow the listed steps:

   In client sessions:

   ```
   export EXTSHM=ON
   ```

   When starting the Db2 server:

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

   On partitioned database environment, 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.

Recommendations for using multiple threads

Multithreaded applications might be difficult to maintain and use if you do not carefully plan the application in advance. When you are writing a multithreaded application, you must consider how you handle data structures.

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.

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

Code page and conntry or region codes are specific to C and C++ embedded SQL applications. On AIX, Solaris, and HP-UX, the functions that are used for runtime querying of the code page and country or region code that you use for a database connection are now thread safe.

However, 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.

Troubleshooting multi-threaded embedded SQL applications

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, following context issues must be considered:

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. When the database manager detect a deadlock, SQLCODE -911 is returned to the application and its unit of work is 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, code your application in a way that avoids deadlocks.

As an example of a deadlock that the database manager cannot 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 sample contexts are shown in following pseudocode:

```
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 that about the semaphore dependency neither context is rolled back. The unresolved dependency leaves the application suspended.

You can avoid the deadlock that can 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 rollback error, context 2 should release the semaphore. Once the semaphore has been released, context 1 can continue and context 2 is free to try again 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.

Programming embedded SQL applications

Programming embedded SQL applications involves the same steps required to assemble an application in your host 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.

Embedded SQL source files

When you develop source code that includes embedded SQL, you must 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.

Embedded SQL application template in C

You are provided with a sample embedded SQL application 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 demonstrates the basic structure required for embedded SQL applications written in C.

Sample program: template.sqc

```
 #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 = %d. \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 = %d. \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("\nID        Name        DEPT       Salary\n");
      printf(" %d          %s          %d       %f\n",
          id, name, dep, 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 = %d. \n", SQLCODE);
   }
   else
   {
      EXEC SQL EXECUTE StmtDyn USING :dept;                            8
   }
   if (SQLCODE  <0)                                                    6
   {
      printf("Execute Error:  SQLCODE = %d. \n", SQLCODE);
   }
```

```
/* 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 = %d. \n", SQLCODE);
}
EXEC SQL OPEN posCur1;
EXEC SQL FETCH posCur1 INTO :id, :name, :dept, :salary ;            9
if (SQLCODE  <0)                                                    6
{
   printf("Fetch Error:  SQLCODE = %d. \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(" %d          %s          %d       %f\n",
       id, name, dep, 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 = %d. \n", SQLCODE);
}

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

Notes to **Sample program: template.sqc**:

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 used after 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 (`EXEC SQL PREPARE StmtDyn FROM :hostVarStmtDyn;`) 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.
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.

Include files and definitions required for embedded SQL applications

Include files are required to provide functions and types that are used within the library. You must include these files before the program can use library functions. By default, include files are 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.

Include files for C and C++ embedded SQL applications

The host-language-specific include files for C and C++ have the file extension `.h`. The C and C++ include files are also called header files.

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 shown previously, 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 shown previously, 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 previous example.

  Note that if the precompiler option COMPATIBILITY_MODE is set to ORA, you can use double quotation marks to specify include file names, for example, `EXEC SQL INCLUDE "abc.h";`. The Db2 database manager provides this feature to facilitate the migration of embedded SQL C applications from other database systems.

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.

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 in the following section.

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.

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 `$HOME/sqllib/include/cobol_i` directory.

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 shown previously, 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 shown previously, 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 previously shown 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.

The include files that are intended to be used in your applications are described here:

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.

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

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.

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 ignores 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, and 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 shown previously, 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 shown previously, 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 previously shown 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.

32-bit FORTRAN header files required for Db2 database application 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.

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

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.

About this task

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 include the EXEC SQL INCLUDE SQLCA statement to avoid multiple definitions of the SQLCA. The remaining source files must use the following lines:

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

Procedure

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

- 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
  ```

What to do next

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

Connecting to Db2 databases in embedded SQL applications

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

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;
```

Note that if the precompiler option COMPATIBILITY_MODE is set to ORA, the following additional syntax for the CONNECT statement is supported. The Db2 database manager provides this feature to facilitate the migration of embedded SQL C applications from other database systems.

```
EXEC SQL CONNECT [ username IDENTIFIED BY password ][ USING  dbname ] ;
```

The parameters are described in the following table:

Parameter	Description
username	Either a host variable or a string specifying the database user name
password	Either a host variable or a string specifying the password
dbname	Either a host variable or a string specifying the database name

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'
```

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

To exchange data between an application and database, you must 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.

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

Certain predefined C and C++ data types correspond to Db2 database column types. You can declare only these C and C++ data types 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 (500 or 501)	short short int sqlint16	16-bit signed integer
INTEGER (496 or 497)	int long long int sqlint32[2]	32-bit signed integer
BIGINT (492 or 493)	long long long __int64 sqlint64[3]	64-bit signed integer
REAL[5] (480 or 481)	float	Single-precision floating point
DOUBLE[6] (480 or 481)	double	Double-precision floating point
DECIMAL(*p*,*s*) (484 or 485)	No exact equivalent; use double	Packed decimal (Consider using the CHAR and DECIMAL functions to manipulate packed decimal fields as character data.)
CHAR(1) (452 or 453)	char	Single character
CHAR(*n*) (452 or 453)	No exact equivalent; use char[*n+1*] where n is large enough to hold the data 1<=*n*<=255	Fixed-length character string

VARCHAR(*n*)
(448 or 449)
	struct tag {
	short int;
	char[*n*]
	}
	
	1<=*n*<=32 672
		Non null-terminated varying character string with 2-byte string length indicator.
		Note: A host variable structure of the following form is always treated as a VARCHAR host variable and cannot be declared:
		`struct tag {`
		`short int;`
		`char[n]`
		`}`

	Alternatively, use char[*n+1*] where n is large enough to hold the data 1<=*n*<=32 672	Null-terminated variable-length character string **Note:** Assigned an SQL type of 460/461.

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

LONG VARCHAR[8]
(456 or 457)
	struct tag {
	short int;
	char[*n*]
	}
	
	32 673<=*n*<=32 700
		Non null-terminated varying character string with 2-byte string length indicator

CLOB(*n*) (408 or 409)	sql type is clob(*n*) 1<=*n*<=2 147 483 647	Non null-terminated varying character string with 4-byte string length indicator
CLOB locator variable[7] (964 or 965)	sql type is clob_locator	Identifies CLOB entities residing on the server
CLOB file reference variable[7] (920 or 921)	sql type is clob_file	Descriptor for file containing CLOB data
BLOB(*n*) (404 or 405)	sql type is blob(*n*) 1<=*n*<=2 147 483 647	Non null-terminated varying binary string with 4-byte string length indicator
BLOB locator variable[7] (960 or 961)	sql type is blob_locator	Identifies BLOB entities on the server
BLOB file reference variable[7] (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(*p*) [4](392 or 393)	Null-terminated character form	Allow 20- 33 characters to accommodate for the null-terminator
	VARCHAR structured form	Allow 19-32 characters.

XML[8]
(988 or 989)
	struct {
	sqluint32 length;
	char data[*n*];
	}
	
	1<=*n*<=2 147 483 647
	
	SQLUDF_CLOB
		XML value

BINARY	unsigned char myBinField[*n*]; 1<= *n* <=255	Binary data

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
VARBINARY	struct myVarBinField_t {sqluint16 length;char data[*n*];} myVarBinField; 1<= *n* <=32 672	Varbinary data

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 1<=*n*<=127	Fixed-length double-byte character string

VARGRAPHIC(*n*)
(464 or 465)
	struct tag {
	short int;
	sqldbchar[*n*]
	}
	
	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 1<=*n*<=16 336	Null-terminated variable-length double-byte character string **Note:** Assigned an SQL type of 400/401.

LONG VARGRAPHIC[8]
(472 or 473)
	struct tag {
	short int;
	sqldbchar[*n*]
	}
	
	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) • Single double-byte character (for column type)

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

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

VARGRAPHIC(*n*)
(464 or 465)
	struct tag {
	short int;
	wchar_t [*n*]
	}
	
	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 1<=*n*<=16 336	Null-terminated variable-length double-byte character string **Note:** Assigned an SQL type of 400/401.

LONG VARGRAPHIC[8]
(472 or 473)
	struct tag {
	short int;
	wchar_t [*n*]
	}
	
	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*) 1<=*n*<=1 073 741 823	Non null-terminated varying double-byte character string with 4-byte string length indicator
DBCLOB locator variable[7] (968 or 969)	sql type is dbclob_locator	Identifies DBCLOB entities residing on the server
DBCLOB file reference variable[7] (924 or 925)	sql type is dbclob_file	Descriptor for file containing DBCLOB data

Table 5. SQL Data Types Mapped to C and C++ Declarations (continued)

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

Note:

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 will be displayed 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 has a type definition for 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. The character string can be from 19 - 32 bytes in length without a null terminator depending on the number of fractional seconds specified. The fractional seconds of the TIMESTAMP data type can be optionally specified with 0-12 digits of timestamp precision.

   When a timestamp value is assigned to a timestamp variable with a different number of fractional seconds, the value is either truncated or padded with 0's to match the format of the timestamp variable.
5. 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).
6. The following SQL types are synonyms for DOUBLE:
   - FLOAT
   - FLOAT(n) where $24 < n < 54$ is a synonym for DOUBLE
   - DOUBLE PRECISION
7. This is not a column type but a host variable type.
8. 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.
9. The LONG VARCHAR and LONG VARGRAPHIC data types are deprecated and might be removed in a future release. Choose the CLOB or DBCLOB data type instead.

The following items 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.

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

There is a mapping between C and C++ and Db2 data types. When you are writing your embedded SQL application, you must be aware of this mapping to ensure that you do not have unexpected data type conversions or data truncation.

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 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 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 1<=*n*<=32 672	Null-terminated varying length string

VARCHAR(*n*) FOR BIT DATA
(448 or 449)
	struct {
	sqluint16 length;
	char[*n*]
	}
	
	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[2]
(456 or 457)
	struct {
	sqluint16 length;
	char[*n*]
	}
	
	32 673<=*n*<=32 700
		Not null-terminated varying length character string

CLOB(*n*)
(408 or 409)
	struct {
	sqluint32 length;
	char data[n];
	}
	
	1<=*n*<=2 147 483 647
		Not null-terminated varying length character string with 4-byte string length indicator

BINARY(*n*) (912 or 913)	char[*n+1*] where n is large enough to hold the data 1<=*n*<=254	Fixed-length binary string

VARBINARY(*n*)
(908 or 909)
	struct {
	sqluint16 length;
	char[*n*]
	}
	
	1<=*n*<=32 672
		Not null-terminated varying length binary string

BLOB(*n*)
(404 or 405)
	struct {
	sqluint32 length;
	char data[n];
	}
	
	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(*p*) (392 or 393)	char[*p+21*] where p is large enough to hold the data 0<=*p*<=12	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 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 1<=*n*<=16 336	Not null-terminated, variable-length double-byte character string

LONG VARGRAPHIC[2]
(472 or 473)
	struct {
	sqluint16 length;
	sqldbchar[*n*]
	}
	
	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];
	}
	
	1<=*n*<=1 073 741 823
		Not null-terminated varying length character string with 4-byte string length indicator

Note:

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 will be displayed in the SQLTYPE field of the SQLDA for these data types.
2. The LONG VARCHAR and LONG VARGRAPHIC data types are deprecated and might be removed in a future release. Choose the CLOB or DBCLOB data type instead.

Supported SQL data types in COBOL embedded SQL applications

Certain predefined COBOL data types correspond to Db2 database column types. You can use only these COBOL data types 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

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

SQL Column Type[1]	COBOL Data Type	SQL Column Type Description
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. 49 length PIC S9(4) COMP-5. 49 name PIC X(*n*). 1<=*n*<=32 672	Variable-length character string
LONG VARCHAR[6] (456 or 457)	01 name. 49 length PIC S9(4) COMP-5. 49 data PIC X(*n*). 32 673<=*n*<=32 700	Long variable-length character string
CLOB(*n*) (408 or 409)	01 MY-CLOB USAGE IS SQL TYPE IS CLOB(n). 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). 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

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

SQL Column Type[1]	COBOL Data Type	SQL Column Type Description
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(*p*) (392 or 393)	01 identifier PIC X(*p+20*). 0<=*p*<=12	19 to 32 byte character string A 19 byte character string can be used, when *p* is 0.
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. 49 length PIC S9(4) COMP-5. 49 name PIC G(*n*) DISPLAY-1. 1<=*n*<=16 336	Variable length double-byte character string with 2-byte string length indicator
LONG VARGRAPHIC[6] (472 or 473)	01 name. 49 length PIC S9(4) COMP-5. 49 name PIC G(*n*) DISPLAY-1. 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). 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

Note:

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 will be displayed 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.
6. The LONG VARCHAR and LONG VARGRAPHIC data types are deprecated and might be removed in a future release. Choose the CLOB or DBCLOB data type instead.

The list of rules for supported COBOL data types are:

- 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.

Supported SQL data types in FORTRAN embedded SQL applications

Certain predefined FORTRAN data types correspond to Db2 database column types. You can declare only these FORTRAN data types 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[5] (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

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

SQL Column Type[1]	FORTRAN Data Type	SQL Column Type Description
TIMESTAMP(*p*) (392 or 393)	CHARACTER*19 to CHARACTER*32	19 to 32 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.

Note:

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 will be displayed 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 LONG VARCHAR data type is deprecated, not recommended, and might be removed in a future release. Choose the CLOB data type instead.

The rule for supported FORTRAN data types is:

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

Supported SQL data types in REXX embedded SQL applications

Certain predefined REXX data types correspond to Db2 database column types. You can declare only these REXX data types 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 $-3.40282346 \times 10^{38}$ to $3.40282346 \times 10^{38}$	Single-precision floating point
DOUBLE[3] (480 or 481)	A number in scientific notation ranging from $-1.79769313 \times 10^{308}$ to $1.79769313 \times 10^{308}$	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 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[5] (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 2000
LONG VARGRAPHIC[5] (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

Note:

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.
5. The LONG VARCHAR and LONG VARGRAPHIC data types are deprecated, not recommended, and might be removed in a future release. Use the CLOB or DBCLOB data type instead.

Host Variables in embedded SQL applications

Host variables are variables that are referenced by embedded SQL statements. Host variables 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 be used 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, this example will either fail to compile because variable `x` is not declared in function `foo2()`, or the value of `x` is not 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;
.
.
.
}
```

Declaring host variables in embedded SQL applications

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

About this task

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;`
	`  short  dept=38, age=26;`
	`  double  salary;`
	`  char  CH;`
	`  char  name1[9], NAME2[9];`
	`  short  nul_ind;`
	`EXEC SQL END DECLARE SECTION;`

COBOL	`EXEC SQL BEGIN DECLARE SECTION END-EXEC.`
	`  01 age  PIC S9(4) COMP-5 VALUE 26.`
	`  01 DEPT  PIC S9(9) COMP-5 VALUE 38.`
	`  01 salary  PIC S9(6)V9(3) COMP-3.`
	`  01 CH  PIC X(1).`
	`  01 name1  PIC X(8).`
	`  01 NAME2  PIC X(8).`
	`  01 nul-ind  PIC S9(4) COMP-5.`
	`EXEC SQL END DECLARE SECTION END-EXEC.`

FORTRAN	`EXEC SQL BEGIN DECLARE SECTION`
	`  integer*2  age  /26/`
	`  integer*4  dept  /38/`
	`  real*8  salary`
	`  character  ch`
	`  character*8  name1,NAME2`
	`  integer*2  nul_ind`
	`EXEC SQL END DECLARE SECTION`

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.

About this task

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;
```

Column data types and host variables in embedded SQL applications

Supported embedded SQL host languages have data types that correspond to the majority of the database manager data types. You can use only these host language data types in host variable declarations.

Each table column 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.

Note:

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 this 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, CHAR(SALARY+BONUS) FROM EMPLOYEE
```

The `CAST` function used in the preceding example 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 can also be subject to character conversion.

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.

About this task

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 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 white space stripping is injected and applied.

XML and XQuery restrictions on developing embedded SQL applications

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 code page 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 code page.
- `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 code page.
- `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 code page.
- `SQL TYPE IS XML AS BLOB_FILE <hostvar_name>`

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

Note:

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`).

Identifying XML values in an SQLDA

To indicate that a base type holds XML data, you must update the sqlname field in the associated SQLDA structure. If you do not indicate that a base type holds XML data, your embedded SQL application might not return the expected results.

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 must 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 must 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.

Identifying null SQL values with null indicator variables

You must prepare embedded SQL applications for receiving null values by associating a *null-indicator variable* with any host variable that can receive a null value. A null-indicator variable is shared by both the database manager and the host application.

Therefore, you must declare this variable in the application as a host variable, which corresponds to the SQL data type SMALLINT.

About this task

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 unspecified indicator variable error is returned when applications fetch a result-set that contains NULL values but fail to specify a null indicator. You can avoid the unspecified indicator variable error, even when null indicator is not specified, when you use the precompile options `UNSAFENULL YES` and `COMPATIBILITY_MODE ORA`.

Application can check the sqlca.sqlerrd[2] field to get the number of rows that are successfully fetched with the same cursor.

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, 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'`

Including SQLSTATE and SQLCODE host variables in embedded SQL applications

You can review error information that is returned in the SQLCODE and SQLSTATE fields of the SQLCA structure. The SQLCA structure is updated after every executable SQL statement and most database manager API calls.

Before you begin

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.

- The PREP option LANGLEVEL SQL92E needs to be specified

About this task

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';`
	`if ( SQLCODE < 0 )`
	`  printf( "Update Error:  SQLCODE =`

COBOL	`EXEC SQL UPDATE staff SET job = 'Clerk' WHERE job = 'Mgr' END_EXEC.`
	`IF SQLCODE LESS THAN 0`
	`  DISPLAY 'UPDATE ERROR:  SQLCODE = ', SQLCODE.`

FORTRAN	`EXEC SQL UPDATE staff SET job = 'Clerk' WHERE job = 'Mgr'`
	`if ( sqlcode .lt. 0 ) THEN`
	`  write(*,*) 'Update error:  sqlcode = ', sqlcode`

Referencing host variables in embedded SQL applications

After you have declared a host variable in your embedded SQL application code, you can reference it later in the application.

About this task

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.

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;` `printf( "Commission = %f\n", cm );`
COBOL	`EXEC SQL FETCH C1 INTO :cm END-EXEC` `DISPLAY 'Commission = ' cm`
FORTRAN	`EXEC SQL FETCH C1 INTO :cm` `WRITE(*,*) 'Commission = ', cm`
REXX	`CALL SQLEXEC 'FETCH C1 INTO :cm'` `SAY 'Commission = ' cm`

Example: Referencing XML host variables in embedded SQL applications

You can create XML host variables in embedded SQL applications so that you can read and process XML data.

The following sample applications demonstrate 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' END-EXEC.
EXEC SQL UPDATE myTable
   SET xmlCol = :xmlBuf
   WHERE id = '001' END-EXEC.

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

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

Host variables in C and C++ embedded SQL applications

Host variables are C or C++ language variables that are referenced within SQL statements. Host variables 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`. `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 runtime 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 regions, 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.

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

You must use an SQL declare section to identify host variable declarations. SQL declare sections alert 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. 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.

You can define, name, and use a host variable within the SQL declare section. In the following example, a struct type called staff_record is first defined. Then the variable named staff_detail is declared as being of type staff_record:

```
EXEC SQL BEGIN DECLARE SECTION ;

typedef struct {
 short id;
```

```
 VARCHAR name[10+1];
 short years;
 double salary;
} staff_record;

staff_record staff_detail;

EXEC SQL END DECLARE SECTION ;
...
SELECT id, name, years, salary
 FROM staff
 INTO :staff_detail
 WHERE id = 10;
...
```

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

The SQL precompiler identifies host variables by their declared name.

The following rules apply when declaring host variable names:

- 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 supports the same scope rules as the C and C++ programming languages. Therefore, you can use the same name for two different variables each existing within their own scope. In the following example, both declarations of the variable called empno are allowed; the second declaration does not cause an error:

```
file: main.sqc
...
void scope1()
{
 EXEC SQL BEGIN DECLARE SECTION ;

 short empno;

 EXEC SQL END DECLARE SECTION ;

 ...
}

void scope2()
{
 EXEC SQL BEGIN DECLARE SECTION ;

 char[15 + 1] empno;    /* this declaration is allowed */

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

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

When you are creating an embedded SQL application in C or C++, there is a template that you can use to declare your host variables and data structures.

The following example 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 */
        sql type is varbinary(12)
                  myVarBinField;          /* SQL type  908 */
        sql type is binary(4)
                  myBinField;             /* SQL type  912 */
        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;
```

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

Your embedded SQL application can declare the SQLCODE and SQLSTATE variables to handle errors or help you debug your 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 must not be specified.

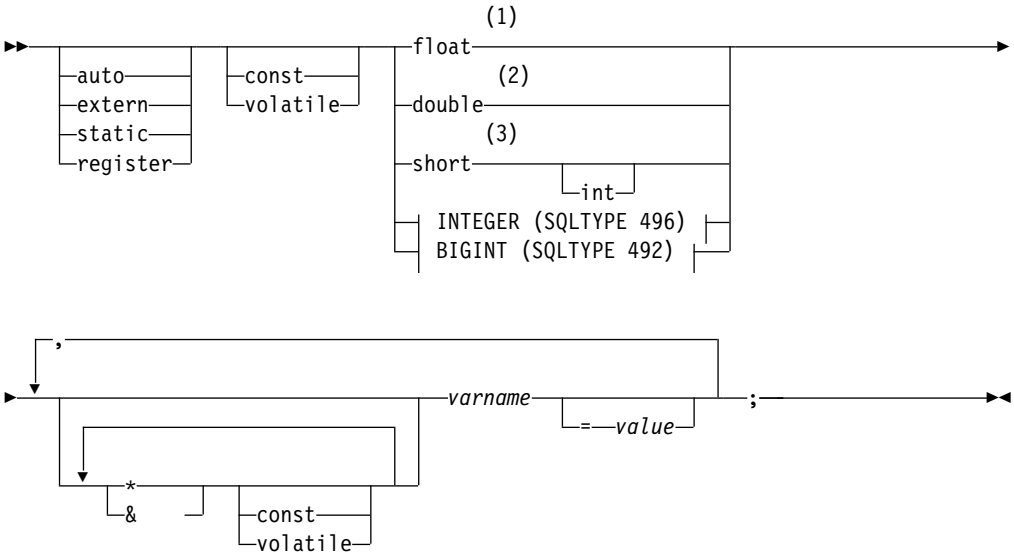
In an application that is made up of multiple source files, the SQLCODE and SQLSTATE variables can be defined in the first source file as in the previous example. Subsequent source files should modify the definitions as follows:

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

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

Numeric host variables that you declare in your embedded C or C++ application are treated as if they were declared in a C or C++ program. You can use host variables to exchange data between the embedded application and the database manager.

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` for INTEGER host variables and `sqlint64` for BIGINT host variables. 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.

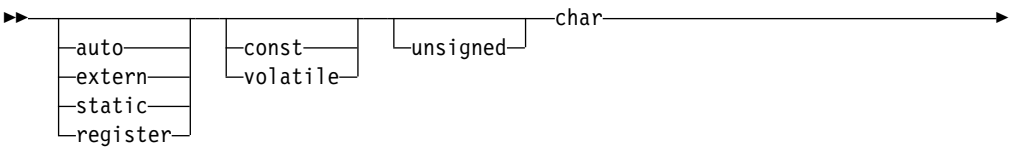
5 For maximum application portability, use `sqlint32` and `sqlint64` for INTEGER and BIGINT host variables. To use the BIGINT data type, your platform must support 64 bit integer values. By default, the use of long host variables results in the precompiler error SQL0402 on platforms where long is

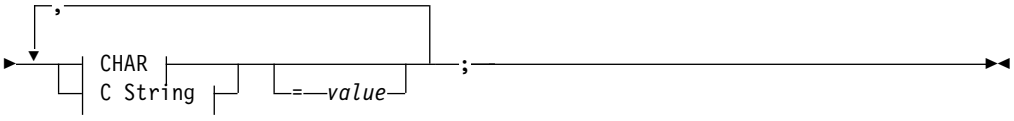
a 64 bit quantity, such as 64 BIT UNIX. Use the PREP option LONGERROR NO to force Db2 to accept long variables as acceptable host variable types and treat them as BIGINT variables.

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

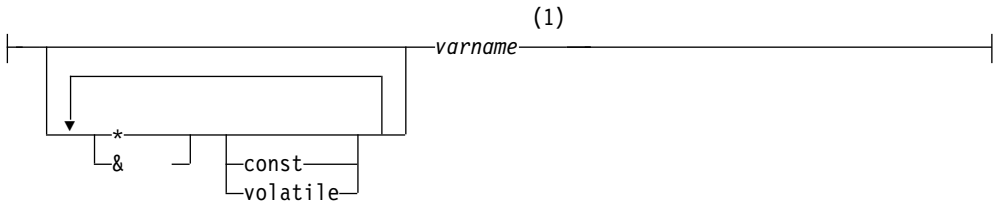
There are two forms of C and C++ variables that you can declare in an embedded SQL application. Form 1 variables are fixed length and null-terminated character host variables and form 2 are variable-length character host variables.

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

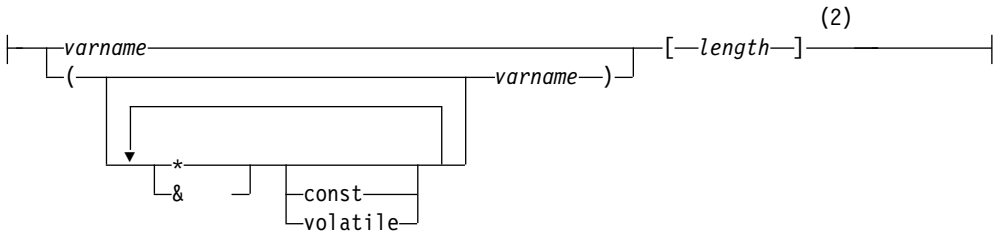




CHAR



C String

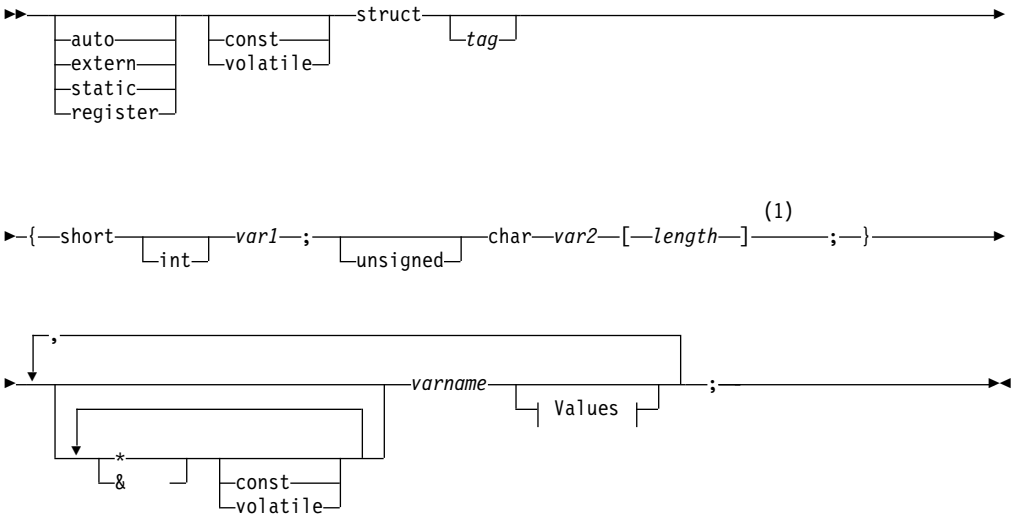


Notes:

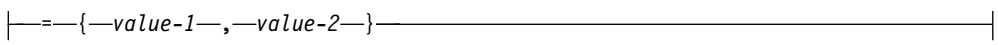
1 CHAR (SQLTYPE 452), length 1

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



Values



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, whereas **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.

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

To handle graphic data in C or C++ applications, you must use host variables based on either the `wchar_t` C or C++ data type or the `sqldbchar` data type.

You can assign graphic data 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.

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

The size and encoding of Db2 graphic data is constant from one operating system to another for a particular code page. However, the size and internal format of the ANSI C or C++ `wchar_t` data type depend on which compiler and operating system you use.

The `sqldbchar` data type is defined by Db2 to be 2 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 with either the `wchar_t` or `sqldbchar` data type. You must use the `wchar_t` data type if you build your application with the `WCHARTYPE CONVERT` precompile option.

If you build your application with the `WCHARTYPE NOCONVERT` precompile option, you can use the `sqldbchar` data type for maximum portability between different Db2 client and server environments. You can use the `wchar_t` data type with the `WCHARTYPE NOCONVERT` option, but only on environments where the `wchar_t` data type is defined as 2 bytes in length.

If you incorrectly use either a `wchar_t` or `sqldbchar` data type in host variable declarations, an error can be returned during the precompile process.

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

You can use the `WCHARTYPE` precompiler option to specify whether you want to use multibyte format or wide-character format for your graphic data.

There are two possible values for the WCHARTYPE option:

CONVERT

If you select the `WCHARTYPE CONVERT` option in Linux or UNIX operating systems, character codes are converted between the graphic host variable and the database manager. For graphic input host variables, the character code is converted from wide-character format to multibyte DBCS character format with the ANSI C function `wcstombs()` before the data is sent to the database manager. For graphic output host variables, the character code is converted from multibyte DBCS character format to wide-character format with the ANSI C function `mbstowcs()` before the data received from the database manager is stored in the host variable.

For Windows operating systems, if a conversion failure is encountered for graphic host variables, user can set the **`SkipLocalCPConversionForWcharConvert`** keyword to `ON` in the IBM data server driver configuration file (`db2dsdriver.cfg`) to avoid the failure.

The advantage to using the `WCHARTYPE CONVERT` option is that it allows your application to use the ANSI C mechanisms for dealing with wide-character strings (L-literals, 'wc' string functions, and others) without having to explicitly convert the data to multibyte format before data is sent to the database manager. The disadvantage is that the implicit conversion can have an impact on the performance of your application at run time, and it can increase memory requirements.

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

If you want the `WCHARTYPE CONVERT` option 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`.

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 multibyte format when interfacing with the database manager.

If you select the `WCHARTYPE NOCONVERT` option, declare all graphic host variables with the `sqldbchar` type for maximum portability to other Db2 client/server environments.

You must consider the guidelines include in the following list:

- 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, or for dealing with GRAPHIC data in any application (including single-byte applications) connected to a UCS-2 database.
- Avoid use of non-DBCS (non-double byte characters) characters, and wide-characters that can be converted to non-DBCS characters, in graphic strings. 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 the WCHARTYPE CONVERT setting is in effect, wide-character data that converts to DBCS data. Use character host variables to store mixed double-byte and single-byte data. Mixed data host variables are unaffected by the setting of the WCHARTYPE option.

- In applications where the WCHARTYPE NOCONVERT precompile option is used, avoid use of L-literals with graphic host variables as L-literals are in wide-character format. An L-literal is a C wide-character string literal that is 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, use graphic string constants, which are independent of the WCHARTYPE setting in SQL statements.
- The setting of the WCHARTYPE option affects graphic data that is passed to and from the database manager using the SQLDA structure and host variables. If the WCHARTYPE CONVERT setting is in effect, graphic data that is received from the application through an SQLDA is presumed to be in wide-character format, and is converted to DBCS format by implicit call to the wcstombs() function. Similarly, graphic output data that is received by an application was converted to wide-character format before the data was placed in application storage.
- Not-fenced stored procedures must be precompiled with the WCHARTYPE NOCONVERT option. Ordinary fenced stored procedures can be precompiled with either the CONVERT or NOCONVERT options, which affects the format of graphic data that is manipulated by SQL statements that are contained in the stored procedure. In either case, however, any graphic data that is passed into the stored procedure through the SQLDA is 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 is 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 occurs on the SQLDA, depending on the calling application's WCHARTYPE setting.
- Graphic data that is passed to user-defined functions (UDFs) is always in DBCS format. Likewise, any graphic data that is returned from a UDF is assumed to be in DBCS format for DBCS databases, and UCS-2 format for EUC and UCS-2 databases.
- Data that is stored in DBCLOB files by using 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.

Note:

1. For Db2 for Windows operating systems, the WCHARTYPE CONVERT option is supported for applications that are 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 converts the code page in

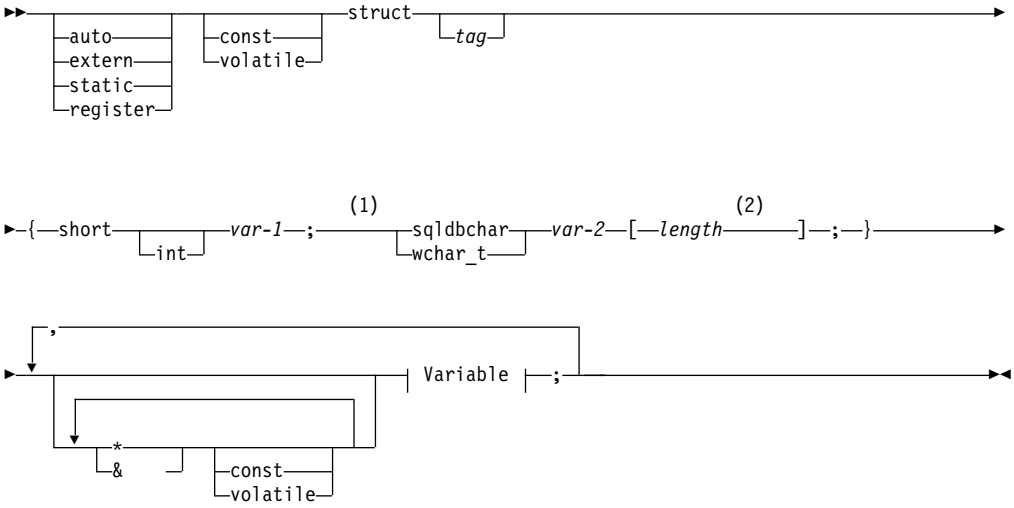
this situation; however, the Microsoft C runtime environment does not handle substitution characters for certain double byte character and it can result in run time conversion errors.

2. If you precompile C applications with 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 an environment with mixed `CONVERT` and `NOCONVERT` applications, you can encounter errors if invalid graphic data is inserted by a `NOCONVERT` application and then fetched by a `CONVERT` application.

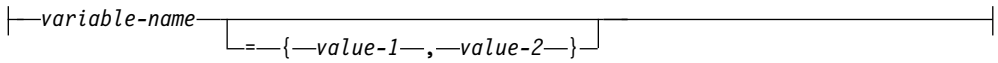
Declaration of VARGRAPHIC type host variables in the structured form in C or C++ embedded SQL applications:

VARGRAPHIC type host variables that you declare in your embedded C or C++ application are treated as if they were declared in a C or C++ program. You can use host variables to exchange data between the embedded application and the database manager.

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



Variable:



Notes:

1 To determine which of the two graphic types to 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.

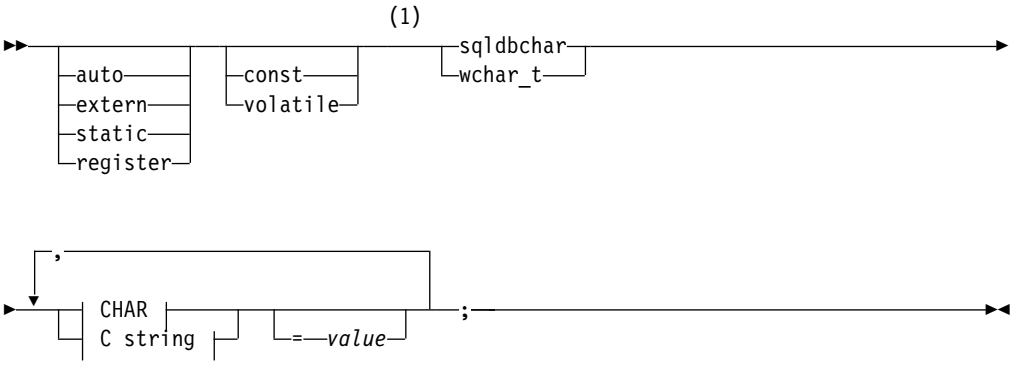
Graphic declaration (VARGRAPHIC structured form) Considerations:

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* must 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.

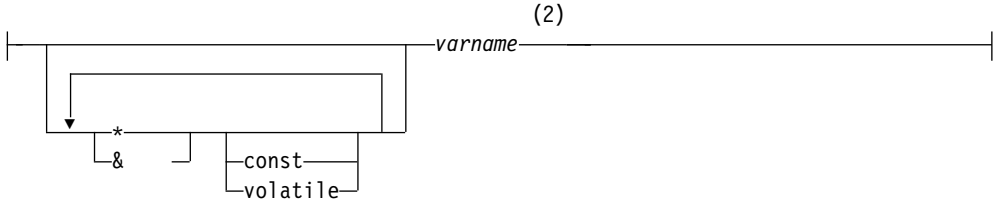
Declaration of GRAPHIC type host variables in single-graphic and null-terminated graphic forms in C and C++ embedded SQL applications:

Single and null-terminated GRAPHIC type host variables that you declare in your embedded C or C++ application are treated as if they were declared in a C or C++ program. You can use host variables to exchange data between the embedded application and the database manager.

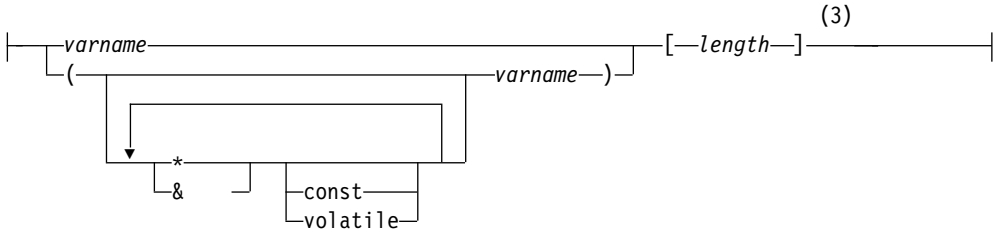
Following is the syntax for declaring a graphic host variable using the single-graphic form and the null-terminated graphic form.



CHAR



C string



Notes:

1 To determine which of the two graphic types to 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)

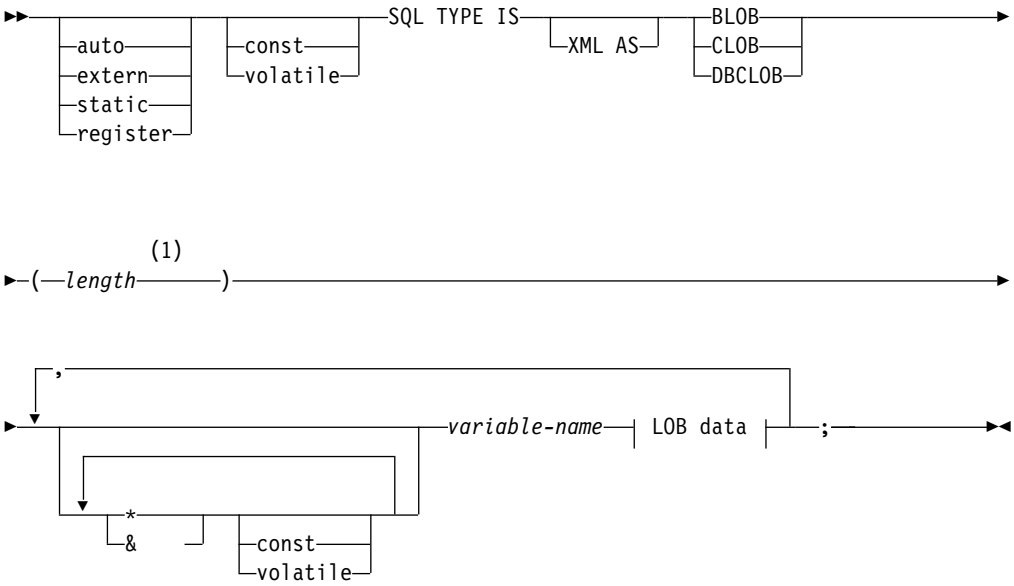
Graphic host variable considerations:

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) must 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.

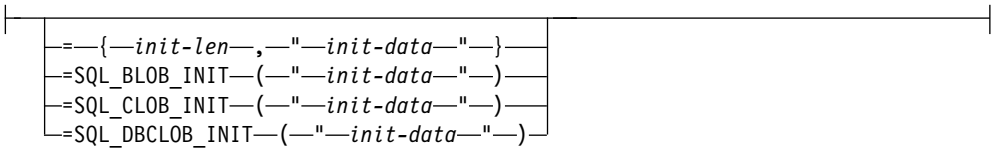
Declaration of large object type host variables in C and C++ embedded SQL applications:

Large object (LOB) type host variables that you declare in your embedded C or C++ application are treated as if they were declared in a C or C++ program. You can use host variables to exchange data between the embedded application and the database manager.

The syntax for declaring large object (LOB) host variables in C or C++ is:



LOB 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 can 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");
```

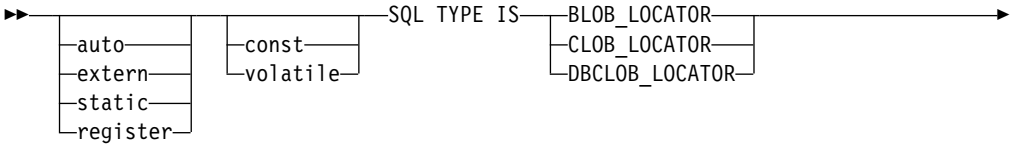
Precompiled with the WCHARTYPE CONVERT option, results in the generation of the following structure:

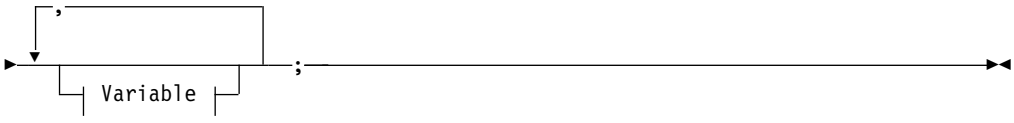
```
struct my_dbclob2_t {
     sqluint32        length;
     wchar_t          data[30000];
} my_dbclob2 = SQL_DBCLOB_INIT(L"mydbdata");
```

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

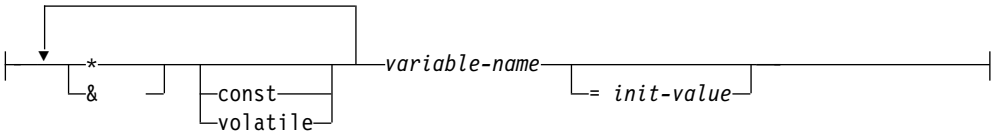
Large object (LOB) locator host variables that you declare in your embedded C or C++ application are treated as if they were declared in a C or C++ program. You can use host variables to exchange data between the embedded application and the database manager.

The syntax for declaring large object (LOB) locator host variables in C or C++ is:





Variable



LOB locator host variable considerations:

1. SQL TYPE IS, BLOB_LOCATOR, CLOB_LOCATOR, DBCLOB_LOCATOR can 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:

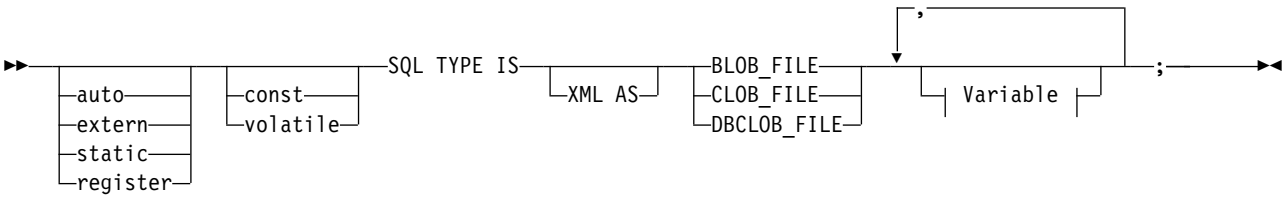
```
sqluint32 my_locator;
```

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

File reference type host variables that you declare in your embedded C or C++ application are treated as if they were declared in a C or C++ program. You can use host variables to exchange data between the embedded application and the database manager.

The syntax for declaring file reference host variables in C or C++ is:

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



Variable

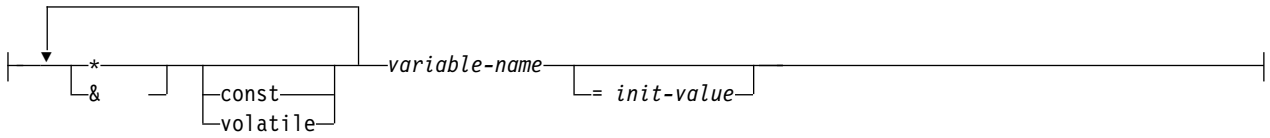


Figure 1. Syntax Diagram

Note: SQL TYPE IS, BLOB_FILE, CLOB_FILE, DBCLOB_FILE can 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: This structure is equivalent to the sqlfile structure located in the sql.h header. See Figure 1 on page 76 to refer to the syntax diagram.

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

You can declare host variables as pointers to specific data types. However, there are some formatting guidelines that you should be aware of.

Before you can declare a host variable pointer, you must consider 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 not a valid 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 shown previously in this topic.

- 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.

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

You can declare class data members as host variables, but you cannot decalre classes or objects as host variables.

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.

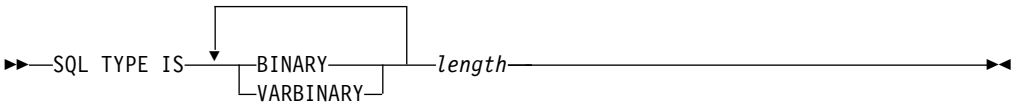
The following example shows a new method, *asWellPaidAs* that takes a second object, *otherGuy*. This method references its members indirectly through a local pointer or reference host variable, as you cannot reference its members directly within the SQL statement.

```
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;
}
```

Declaration of binary type host variables in C, C++ embedded SQL applications:

Binary host variables that you declare in your embedded C and C++ applications are treated as if they were declared in a C or C++ program. You can use host variables to exchange data between the embedded application and the database manager.

The syntax for binary and varbinary locator host variables in C, C++ is:



Example

Declaring:

```
SQL TYPE IS BINARY(4) myBinField;
```

Results in the generation of the following C code:

```
 unsigned char myBinField[4];

where length  N (1<= N <=255)
```

Declaring:

```
SQL TYPE IS VARBINARY(12) myVarBinField;
```

Results in the generation of the following C code:

```
struct myVarBinField_t { sqluint16 length;
char data[12];
} myVarBinField;

Where length is N (1<= N <=32704)
```

Embedded SQL application support of BINARY and VARBINARY:

Embedded SQL application can copy BINARY data of predetermined length after you declare the BINARY data type variable in the declare section. The VARBINARY data type variable can be declared in the declare section of the embedded SQL application with set length to copy the VARBINARY data.

The following example shows you how to use the BINARY and VARBINARY data types in an embedded application:

```
EXEC SQL BEGIN DECLARE SECTION;
sql type is binary(50)  binary1 ;
sql type is varbinary(100)  binary2 ;
EXEC SQL END DECLARE SECTION;
char  strng1[50];
char  strng2[50];

memset( binary1, 0x00, sizeof(binary1) );
memset( binary2.data, 0x00, sizeof(binary2.data) );
strcpy( strng1, "AAAAAAZZZZZMMMMMMMMMJJJJJJJJJJJJJJ" );
strcpy( strng2, "BBBBBBBBBBBBBBBCCCCCCCCCCCDDDDDDDDEEEEEEEEEEK" );
memcpy( binary1, strng1, strlen(strng1) );
memcpy( binary2.data, strng2, strlen(strng2) );
binary2.length = strlen(binary2.data);
EXEC SQL INSERT INTO test1 VALUES ( :binary1, :binary2 );
```

On retrieval from the database, the length of the data is set properly in the corresponding structure.

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 required 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';
```

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 with `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 listed 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 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 might 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.

Binary storage of variable values using the FOR BIT DATA clause in C and C++ embedded SQL applications:

You can declare certain database columns by using the FOR BIT DATA clause. These columns, which generally contain characters, are used to hold binary information.

You cannot use the standard C or C++ string type 460 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.

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, you must initialize variables using the "=" symbol, not 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;
```

Macro expansion and the DECLARE SECTION of C and C++ embedded SQL applications:

The C or C++ precompiler cannot directly process any C macro that is used in a declaration within a declare section. You must first preprocess the source file with an external C preprocessor by specifying 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 must not have the C preprocessor output any #line macros itself, as they can be incorrectly mixed with ones generated by the precompiler.

Notes on using macro expansion:

1. The command you specify through the PREPROCESSOR option must include all the required 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 example 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;
```

Host structure support in the declare section of C and C++ embedded SQL applications:

A host structure contains a list of host variables that can be referred to by embedded SQL statements. With host structure support, the C or C++ precompiler allows host variables to be grouped into a single host structure.

Host structure support 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 shown previously, 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 preceding example, `: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 might 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).

Null or truncation indicator variables and indicator tables in C and C++ embedded SQL applications:

For each host variable that can receive null values, you must declare indicator variables as a short data type.

An *indicator table* is a collection of indicator variables to be used with a host structure. An indicator table must be declared as an array of short integers. For example:

```
short ind_tab[10];
```

The preceding example declares an indicator table with 10 elements. It can be used in an SQL statement as follows:

```
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]`.

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.

As previously defined, 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* and a host variable of SQLTYPE 468 with length *n* is treated as a VARGRAPHIC data type with length *n*.

C and C++ host variable arrays:

You can use C and C++ host variable arrays for FETCH INTO, INSERT, UPDATE, and DELETE statements that are non-dynamic, when you set the precompiler option `COMPATIBILITY_MODE` to `ORA`.

For a host variable array that is declared for an INSERT, UPDATE, or DELETE statement, you must ensure that entire array elements are initialized with a value. Otherwise, unexpected data can get introduced or removed from the table.

When you specify multiple host variable arrays for one database object in an INSERT, UPDATE, or DELETE statement, you must declare the same cardinality for those arrays. Otherwise, the smallest cardinality that is declared among the arrays is used.

The total number of rows that are successfully processed is stored in the `sqlca.sqlerrd[3]` field. However, the `sqlca.sqlerrd[3]` field does not represent the number of rows that are committed successfully in the case of INSERT, UPDATE, or DELETE operations.

The total number of rows that are affected by the INSERT, UPDATE, or DELETE operation is stored in the `sqlca.sqlerrd[2]` field.

In the following example, host variable arrays *arr_in1* and *arr_in2* demonstrate the use of the `sqlca.sqlerrd[2]` and `sqlca.sqlerrd[3]` fields:

```
// Declaring host variables with cardinality of 5.
EXEC SQL BEGIN DECLARE SECTION;
  sqlint32 arr_in1[5];
  char arr_in2[5][11];
EXEC SQL END DECLARE SECTION;
...
// Populating the arrays.
for ( i = 0; i < 5; i++)
{
    arr_in1[i] = i + 1;
    sprintf(arr_in2[i], "hello%d", i + 1);
}

// A duplicate value is introduced for arr_in1 array.
// arr_in1[0]==arr_in1[4]
arr_in1[4] = 1;

// The C1 column in the table tbl1 requires an unique key
// and doesn't allow duplicate values.

EXEC SQL INSERT into tbl1 values (:arr_in1, :arr_in2);
printf("sqlca.sqlcode = %d\n", sqlca.sqlcode ); // -803

// Since arr_in1[0] and arr_in1[4] have identicle values,
// the INSERT operation fails when arr_in1[4] element is
// processed for the INSERT operation (which is 5th row
// insert attempt).
// The INSERT operation successfully processed 4 rows (not committed).
printf("sqlca.sqlerrd[3] = %d\n", sqlca.sqlerrd[3] ); //Prints 4

// The INSERT operation failed and 0 rows are impacted.
```

```
printf("sqlca.sqlerrd[2] = %d\n", sqlca.sqlerrd[2] ); //Prints 0

// No rows are present in tbl1 as the INSERT operation failed.
// C1          C2
//----------- -----------
// 0 record(s) selected.
```

Use of C or C++ host variable arrays in FETCH INTO statements

You can declare a cursor and do a bulk fetch into a variable array until the end of the row is reached. Host variable arrays that are used in the same FETCH INTO statement must have same cardinality. Otherwise, the smallest declared cardinality is used for the array.

In one FETCH INTO statement, the maximum number of records that can be retrieved is the cardinality of the array that is declared. If more rows are available after the first fetch, you can repeat the FETCH INTO statement to obtain the next set of rows.

In the following example, two host variable arrays are declared; *empno* and *lastname*. Each can hold up to 100 elements. Because there is only one FETCH INTO statement, this example retrieves 100 rows, or less.

```
// Declaring host variables
EXEC SQL BEGIN DECLARE SECTION;
    char   empno[100][8];
    char   lastname[100][15];
EXEC SQL END DECLARE SECTION;

EXEC SQL DECLARE empcr CURSOR FOR
 SELECT empno, lastname FROM employee;

EXEC SQL OPEN empcr;

EXEC SQL WHENEVER NOT FOUND GOTO end_fetch;

while (1) {
 EXEC SQL FETCH empcr INTO :empno :lastname;  /* bulk fetch       */
 ...                                          /* 100 or less rows */
 ...
}
end_fetch:
EXEC SQL CLOSE empcr;
```

Use of C or C++ host variable arrays in INSERT statements

In the following example, host variable arrays *arr_in1* and *arr_in2* are used for an INSERT statement:

```
// Declaring host variables.
EXEC SQL BEGIN DECLARE SECTION;
  sqlint32 arr_in1[3];
  char arr_in2[3][11];
EXEC SQL END DECLARE SECTION;

...
// Populating the arrays.
for ( i = 0; i < 3; i++)
{
    arr_in1[i] = 100 + i;
    sprintf(arr_in2[i], "hello%d", arr_in1[i]);
}

// The 'arr_in1' & 'arr_in2' are host variable arrays.
EXEC SQL INSERT into tbl1 values (:arr_in1, :arr_in2);
printf("sqlca.sqlcode = %d\n", sqlca.sqlcode ); // 0

// The INSERT operation inserted 3 rows without encounting an error.
printf("sqlca.sqlerrd[3] = %d\n", sqlca.sqlerrd[3] ); // 3
```

```
// The INSERT operation was successful and 3 rows has been stored in database.
printf("sqlca.sqlerrd[2] = %d\n", sqlca.sqlerrd[2] ); // 3

// The tbl1 table now contains the following rows:
//C1          C2
//----------- -----------
//        100 hello1
//        101 hello2
//        102 hello3
```

Use of C or C++ host variable arrays in UPDATE statements

In the following example, host variable arrays *arr_in1* and *arr_in2* are used for an UPDATE statement:

```
// Declaring host variables
EXEC SQL BEGIN DECLARE SECTION;
  sqlint32 arr_in1[3];
  sqlint32 arr_in2[2];
EXEC SQL END DECLARE SECTION;

...
// Populating the arrays.
for ( i = 0; i < 3; i++)
{
   arr_in1[i] = 100 + i;
}

arr_in2[0] = 1000;
arr_in2[1] = 1001;

// Table tbl2 consists of following rows before an update statement is issued.
//C1          C2
//----------- -----------
//        100        500
//        101        501
//        102        502

// The 'arr_in1' array is declared with cardinality of 3 for use in the
// SET clause of an UPDATE statement.
// The 'arr_in2' array is declared with cardinality of 2 for use in the
// WHERE clause of an UPDATE statement.
// The tbl2 table contains 3 rows.
// The following UPDATE statement will affect only 2 rows as per arr_in2
// for column c2 and remaining need to be untouched.

// The 'arr_in1' array in the following update statement is treated as
// having cardinality of 2.
EXEC SQL UPDATE tbl2 SET c2 = :arr_in2 + c2 where c1 = :arr_in1;
printf("sqlca.sqlcode = %d\n", sqlca.sqlcode ); // 0

// As there is no error in update statement, sqlca.sqlerrd[3]
// contains rows which are updated successfully.
printf("sqlca.sqlerrd[3] = %d\n", sqlca.sqlerrd[3] ); // 2

// update successful and 2 rows has been updated in database.
printf("sqlca.sqlerrd[2] = %d\n", sqlca.sqlerrd[2] ); // 2

// The tbl2 table now contains the following rows:
//C1          C2
//----------- -----------
//        100       1500
//        101       1502
//        102        503
```

Use of C or C++ host variable arrays in DELETE statements

In the following example, host variable arrays *arr_in1* and *arr_in2* are used for a DELETE statement:

```
// Declaring host variables
EXEC SQL BEGIN DECLARE SECTION;
  sqlint32 arr_in1[3];
EXEC SQL END DECLARE SECTION;
...
// Populating the arrays.
for ( i = 0; i < 3; i++)
{
   arr_in1[i] = 101 + i;
}

// Initial data in the tbl2 table:
// C1          C2
// ----------- -----------
//         100         500
//         101         501
```

```
//         102         502
//         103         503
//         104         504
// using array host while executing delete statement in where clause
// The'arr_in1' host variable array is used in the WHERE clause of
// an DELETE statement.

EXEC SQL DELETE FROM tbl2 where c1 = :arr_in1;
printf("sqlca.sqlcode = %d\n", sqlca.sqlcode ); // 0

// delete successful attempted rows are 3
printf("sqlca.sqlerrd[3] = %d\n", sqlca.sqlerrd[3] ); // 3

// delete successful and 3 rows has been deleted in database.
printf("sqlca.sqlerrd[2] = %d\n", sqlca.sqlerrd[2] ); // 3

// The tbl2 table now contains the following rows:
// C1          C2
// ----------- -----------
//         100         500
//         104         504
```

Restrictions with C or C++ host variable array support

The use of a C or C++ host variable array in embedded SQL applications is subject to the following restrictions:

- Host variables arrays are supported by C or C++ embedded SQL applications that connect to Db2 servers.
- Host variable arrays must be declared in the `DECLARE SECTION` with exact size of the array elements (cardinality).
- Specific array element cannot be specified in a SQL statement.
- The INSERT, UPDATE, or DELETE operation with host variable arrays is run as an atomic operation on the database server. If any array element causes an SQL_ERROR, current transaction is rolled back.
- Use of host variable arrays are not supported by dynamically prepared INSERT, UPDATE, or DELETE statements.
- Maximum size of array element (cardinality) is 32672.
- The following C and C++ data types are not supported for use with host variable arrays:
  - Another host variable array (nesting)
  - BLOB
  - BLOB file reference
  - BLOB locator variable
  - CLOB
  - CLOB file reference
  - CLOB locator variable
  - User-defined data type
  - XML
- From Db2 V11.1 M3 FP3 onwards, FOR N ROWS clause can be used to specify the cardinality for INSERT and MERGE statement, where N can be an integer or a host variable of type int or short. If array host variables are used, it will take the minimum cardinality value among all the host variables that are used in the SQL.
- Host variable array support is not provided for Db2 for z/OS® and Db2 for i servers.

Structure arrays:

You can use structure arrays for FETCH INTO, INSERT, UPDATE, and DELETE statements that are non-dynamic, when you set the precompiler option `COMPATIBILITY_MODE` to `ORA`.

You can use structure arrays to store multiple column data in a structure form.

For a structure array that is declared for an INSERT, UPDATE, or DELETE statement, you must ensure that all array elements are initialized with a value. Otherwise, unexpected data can get introduced or removed from the table.

The total number of rows that are successfully processed is stored in the `sqlca.sqlerrd[3]` field. However, the `sqlca.sqlerrd[3]` field does not represent the number of rows that are committed successfully in the case of INSERT, UPDATE, or DELETE operations.

The total number of rows that are impacted by the INSERT, UPDATE, or DELETE operation is stored in the `sqlca.sqlerrd[2]` field.

In one FETCH INTO statement, the maximum number of records that can be retrieved is the cardinality of the array that is declared. If more rows are available after the first fetch, you can repeat the FETCH INTO statement to obtain the next set of rows.

A structure array can be used to store multiple column data in a structure form when a FETCH INTO statement is run. In the following example, a structure array is used for a FETCH INTO statement:

```
// Declare structure array with cardinality of 3.
EXEC SQL BEGIN DECLARE SECTION;
 struct MyStruct
 {
  int c1;
  char c2[11];
 } MyStructVar[3];
EXEC SQL DECLARE cur CURSOR FOR
 SELECT empno, lastname FROM employee;
EXEC SQL END DECLARE SECTION;
...
// MyStrutVar is a structure array for host variables
EXEC SQL FETCH cur INTO :MyStructVar;
```

You can use a structure array to store multiple rows for an INSERT statement. In the following example, a structure array is used for an INSERT statement:

```
// Declare structure array with cardinality of 3.
EXEC SQL BEGIN DECLARE SECTION;
   typedef struct _st_type {
       int   id;
       char  name[21];
   } st_type;

   st_type st[3];
EXEC SQL END DECLARE SECTION;

...
// Populating the array.
for( i=0; i<3; i++)
{
      memset( &st[i], 0x00, sizeof(st_type));
      if( i==0) { st[i].id = 100; strcpy(st[i].name, "hello1");}
      if( i==1) { st[i].id = 101; strcpy(st[i].name, "hello2");}
      if( i==2) { st[i].id = 102; strcpy(st[i].name, "hello3");}
}
// The structure elements must be in
// the same order as that of the table elements.
//
```

```
EXEC SQL INSERT INTO tbl values (:st);

// Check for SQLCODE.
printf("sqlca.sqlcode = %d\n", sqlca.sqlcode ); // 0
// The INSERT operation inserted 3 rows without encounting an error
printf("sqlca.sqlerrd[3] = %d\n", sqlca.sqlerrd[3] ); // 3

// The INSERT operation was successful and 3 rows has been stored in database.
printf("sqlca.sqlerrd[2] = %d\n", sqlca.sqlerrd[2] ); // 3

// The tb11 table now contains the following rows:
// C1          C2
// ----------- -----------
//         100 hello1
//         101 hello2
//         102 hello3
```

Restrictions with the structure array support

The use of the structure array in embedded SQL applications is subject to the following restrictions:

- Structure arrays are supported by C or C++ embedded SQL applications that connect to Db2 servers.
- Structure arrays must be declared in the `DECLARE SECTION` with exact size of the array elements (cardinality).
- Specific array element cannot be specified in a SQL statement.
- The INSERT, UPDATE, or DELETE operation with structure arrays is run as an atomic operation on the database server. If any array element causes an SQL_ERROR, current transaction is rolled back.
- Use of structure arrays are not supported by dynamically prepared INSERT, UPDATE, or DELETE statements.
- When structure array is specified, only one structure array can be declared in an embedded SQL application.
- You cannot create a structure array within another structure array (for example, nested structure arrays).
- Maximum size of array element (cardinality) is 32672.
- The following C and C++ data types are not supported for use with structure arrays:
  - BLOB
  - BLOB file reference
  - BLOB locator variable
  - CLOB
  - CLOB file reference
  - CLOB locator variable
  - User-defined data type
  - XML

Indicator variable arrays:

You can use indicator arrays for FETCH INTO, INSERT, UPDATE, and DELETE statements that are non-dynamic, when you set the precompiler option `COMPATIBILITY_MODE` to `ORA`.

An indicator variable array is a short data type variable that is associated with a specific host variable array or a structure array. Each indicator variable element in the indicator variable array can contain `0` or `-1` value that indicates whether an associated host variable or structure contains a null value. If an indicator variable value is less than zero, it identifies the corresponding array value as NULL.

In FETCH INTO statements, you can use indicator variable arrays to determine whether any elements of array variables are null.

You can use the keyword **INDICATOR** to identify an indicator variable, as shown in the example.

In the following example, the indicator variable array that is called *bonus_ind* is declared. The *bonus_ind* indicator variable array can have up to 100 elements, the same cardinality as the *bonus* array variable. When the data is being fetched, if the value of *bonus* is NULL, the value in *bonus_ind* is negative.

```
EXEC SQL BEGIN DECLARE SECTION;
    char   empno[100][8];
    char   lastname[100][15];
    short  edlevel[100];
    double bonus[100];
    short  bonus_ind[100];
EXEC SQL END DECLARE SECTION;

EXEC SQL DECLARE empcr CURSOR FOR
 SELECT empno, lastname, edlevel, bonus
  FROM employee
  WHERE workdept = 'D21';

EXEC SQL OPEN empcr;

EXEC SQL WHENEVER NOT FOUND GOTO end_fetch;

while (1) {
 EXEC SQL FETCH empcr INTO :empno :lastname :edlevel,
         :bonus INDICATOR :bonus_ind
 ...
 ...
}
end_fetch:
EXEC SQL CLOSE empcr;
```

Instead of being identified by the **INDICATOR** keyword, an indicator variable can immediately follow its corresponding host variable, as shown in the following example:

```
EXEC SQL FETCH empcr INTO :empno :lastname :edlevel, :bonus:bonus_ind
```

In the following example, the indicator variable arrays *ind_in1* and *ind_in2* are declared. It can have up to three elements, the same cardinality as the *arr_in1* and *arr_in2* array variables. If the value of *ind_in1* or *ind_in2* is negative, the NULL value is inserted for the corresponding *arr_in1* or *arr_in2* value.

```
// Declare host & indicator variablesof array size 3
EXEC SQL BEGIN DECLARE SECTION;
  sqlint32 arr_in1[3];
  char arr_in2[3][11];
  short ind_in1[3]; // indicator array size is same as host
                    // variable's array size
  short ind_in2[3]; // note here indicator array size is greater
                    // than host variable's array size
EXEC SQL END DECLARE SECTION;

...
// Populating the arrays.
for ( i = 0; i < 3; i++)
{
    arr_in1[i] = i + 1;
    sprintf(arr_in2[i], "hello%d", arr_in1[i]);
}

ind_in1[0] = 0;
ind_in1[1] = SQL_NULL_DATA; // Mark it as a NULL data
ind_in1[2] = 0;

ind_in2[0] = 0;
ind_in2[1] = 0;
ind_in2[2] = SQL_NULL_DATA; // Mark it as a NULL data
```

```
// 'arr_in1' & 'arr_in2' are host variable arrays
// 'ind_in1' & 'ind_in2' are indicator variable arrays
EXEC SQL INSERT into tbl1 values (:arr_in1 :ind_in1, :arr_in2 :ind_in2);

// The tbl1 table now contains the following rows:
C1          C2
----------- -----------
          1 hello1
            hello2  // c1 is set to NULL as indicator is set
          3         // c2 is set to NULL as indicator is set
```

If the cardinality of indicator variable array does not match the cardinality of the corresponding host variable array, an error is returned.

In the following example, the indicator structure array *MyStructInd* is declared.

```
// declaring indicator structure array of size 3
EXEC SQL BEGIN DECLARE SECTION;
   ...

   struct MyStructInd
   {
       short c1_ind;
       short c2_ind;
   } MyStructVarInd[3];
EXEC SQL END DECLARE SECTION;

...

// using structure array host variables & indicators structure type
// array while executing FETCH statement
// 'MyStructVar' is structure array for host variables
// 'MyStructVarInd' is structure array for indicators
EXEC SQL FETCH cur INTO :MyStructVar :MyStructVarInd;
```

Important: The following conditions must be met when the indicator structure array is used.

- The cardinality of the indicator structure array must be equal to or greater than the cardinality of the structure array.
- All members in the indicator structure array must use the short data type.
- The number of members in the indicator structure array must match the number of members in the corresponding structure array.
- For INSERT, UPDATE and DELETE operations, application must ensure that all indicator variables are initialized with either 0 or SQL_NULL_DATA (-1).

The total number of rows that are successfully processed is stored in the sqlca.sqlerrd[3] field. However, the sqlca.sqlerrd[3] field does not represent successfully committed number of rows in the case of INSERT, UPDATE, or DELETE operations. The total number of rows that are impacted by the INSERT, UPDATE, or DELETE operation is stored in the sqlca.sqlerrd[2] field.

Host variables in COBOL

Host variables are COBOL language variables that are referenced within SQL statements. Host variables 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.

Host variable names in COBOL:

The SQL precompiler identifies host variables by their declared name.

You must comply with the following rules when declaring host variable names:

- 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 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.

Declare section for host variables in COBOL embedded SQL applications:

You must use an SQL declare section must be used to identify host variable declarations. The SQL declare 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.

Example: SQL declare section template for COBOL embedded SQL applications:

When you are creating an embedded SQL application in COBOL, there is a template that you can use to declare your host variables and data structures.

The following code 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.
```

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.

If you use these data types, you must ensure that 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

SQLSTATE and SQLCODE Variables in COBOL embedded SQL application:

To handle errors or debug your embedded SQL application, you should test the values of the SQLCODE or SQLSTATE variable. You can return these values as output parameters or as part of a diagnostic message string, or you can insert these values into a table to provide basic tracing support.

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 END-EXEC.
01 SQLSTATE PIC X(5).
01 SQLCODE  PIC S9(9) USAGE COMP.
.
.
.
EXEC SQL END DECLARE SECTION END-EXEC.
```

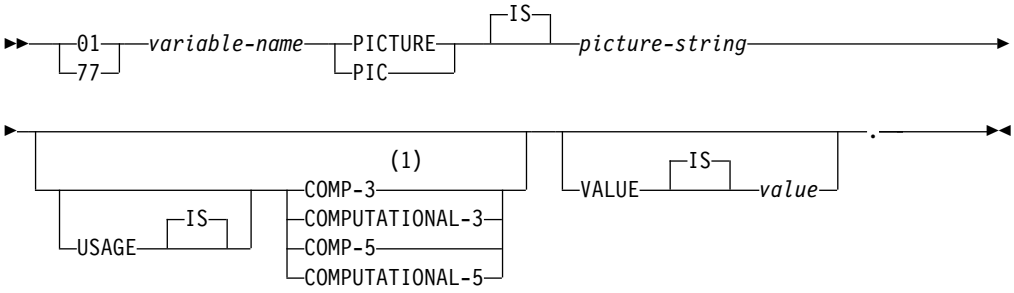
If neither of these is specified, the SQLCODE declaration is assumed during the precompile step. The SQLCODE and SQLSTATE variables can be declared using level 01 (as shown in the previous example) or level 77. Note that when using this option, the INCLUDE SQLCA statement should not be specified.

For applications made up of multiple source files, the SQLCODE and SQLSTATE declarations can be included in each source file as shown previously.

Declaration of numeric host variables in COBOL embedded SQL applications:

Numeric host variables that you declare in your embedded COBOL application are treated as if they were declared in a COBOL program. You can use host variables to exchange data between the embedded application and the database manager.

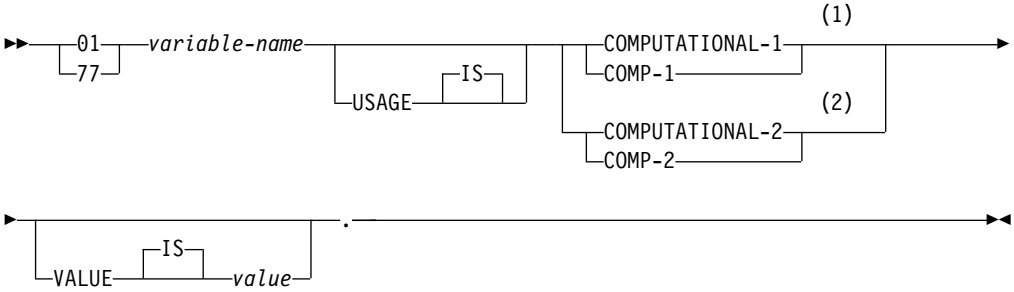
The syntax for numeric host variables is:



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:

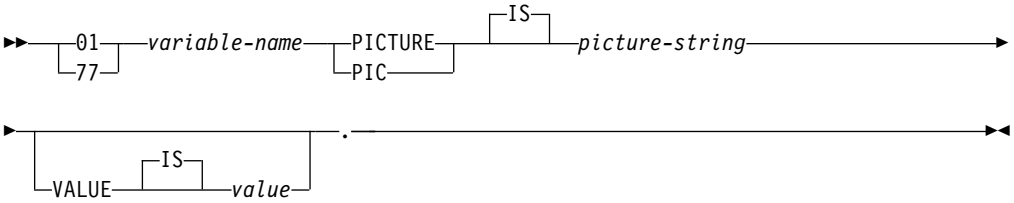
1. *Picture-string* must have one of the following forms:
   - S9(m)V9(n)
   - S9(m)V
   - S9(m)
2. Nines can be expanded (for example., "S999" instead of S9(3)")
3. *m* and *n* must be positive integers.

Declaration of fixed length and variable length character host variables in COBOL embedded SQL applications:

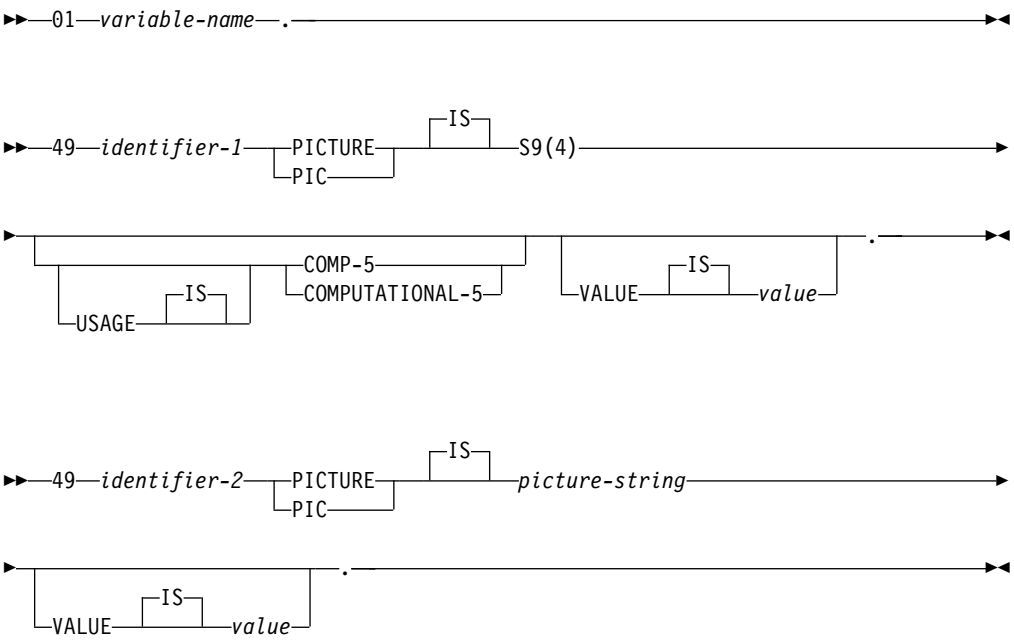
Fixed-length and variable-length character host variables that you declare in your embedded COBOL application are treated as if they were declared in a COBOL program. You can use host variables to exchange data between the embedded application and the database manager.

The syntax for character host variables is:

Fixed Length



Variable length



Character host variable consideration:

1. *Picture-string* must have the form *X(m)*. Alternatively, X's can 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 might 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 the following example, 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:

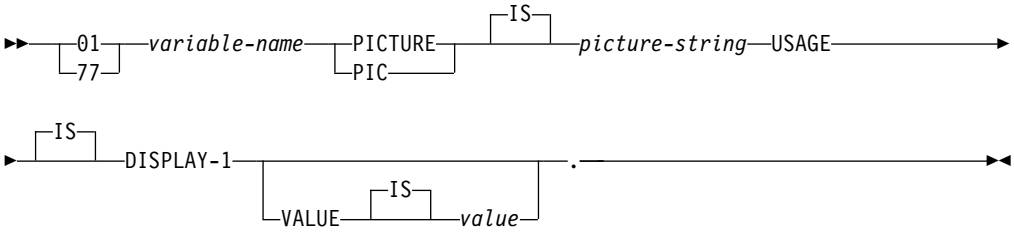
```
EXEC SQL BEGIN DECLARE SECTION END-EXEC.
01 dbname PIC X(8).
01 userid PIC X(8).
01 p-word.
    49 L PIC S9(4) COMP-5.
    49 D PIC X(18).
EXEC SQL END DECLARE SECTION END-EXEC.
PROCEDURE DIVISION.
    MOVE "sample" TO dbname.
    MOVE "userid" TO userid.
    MOVE "password" TO D OF p-word.
    MOVE 8          TO L of p-word.
EXEC SQL CONNECT TO :dbname USER :userid USING :p-word
END-EXEC.
```

Declaration of fixed length and variable length graphic host variables in COBOL embedded SQL applications:

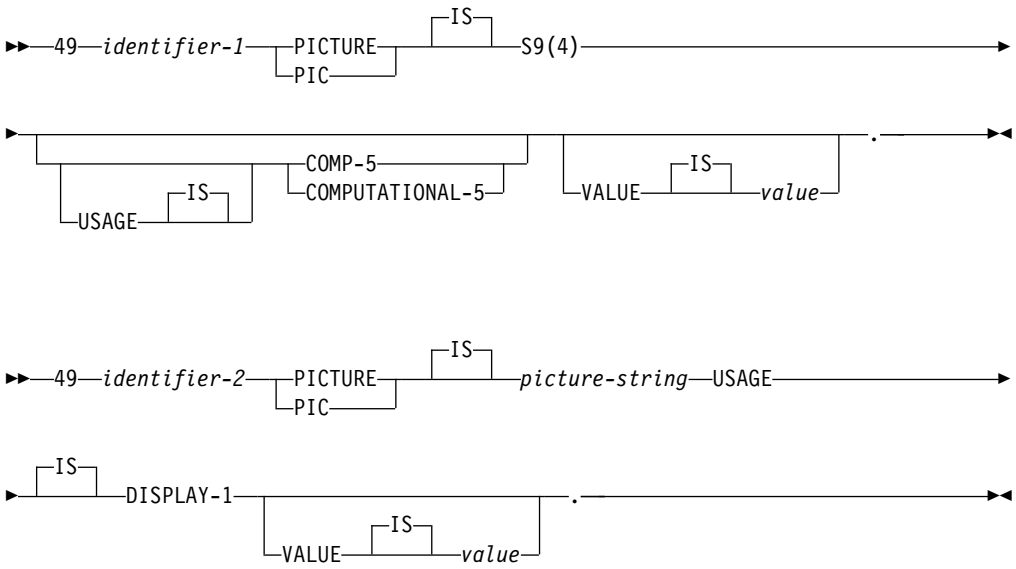
Fixed-length and variable-length graphic host variables that you declare in your embedded COBOL application are treated as if they were declared in a COBOL program. You can use host variables to exchange data between the embedded application and the database manager.

Following is the syntax for graphic host variables.

Fixed Length



Variable Length



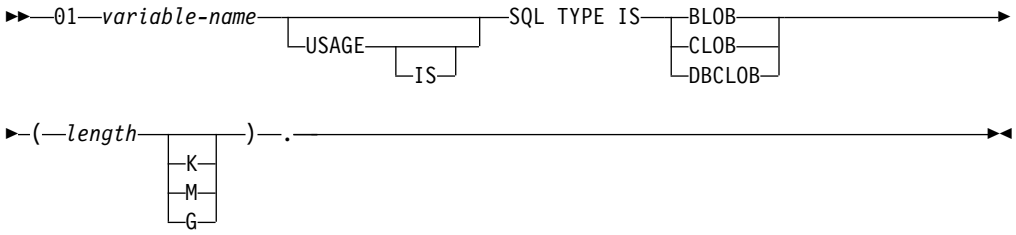
Graphic Host Variable Considerations:

1. *Picture-string* must have the form *G(m)*. Alternatively, G's can 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 might be restricted.

Declaration of large object type host variables in COBOL embedded SQL applications:

Large object (LOB) type host variables that you declare in your embedded COBOL application are treated as if they were declared in a COBOL program. You can use host variables to exchange data between the embedded application and the database manager.

The syntax for declaring large object (LOB) host variables in COBOL is:



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).
```

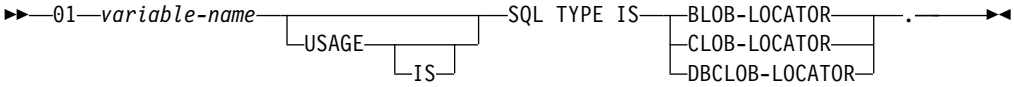
Results in the generation of the following structure:

```
01 MY-DBCLOB.
   49 MY-DBCLOB-LENGTH PIC S9(9) COMP-5.
   49 MY-DBCLOB-DATA PIC G(30000) DISPLAY-1.
```

Declaration of large object locator type host variables in COBOL embedded SQL applications:

Large object (LOB) locator type host variables that you declare in your embedded COBOL application are treated as if they were declared in a COBOL program. You can use host variables to exchange data between the embedded application and the database manager.

The syntax for declaring large object (LOB) locator host variables in COBOL is:



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.
```

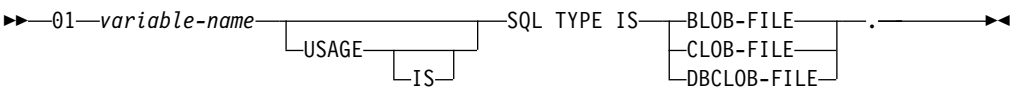
Results in the generation of the following declaration:

```
01 MY-LOCATOR PIC S9(9) COMP-5.
```

Declaration of file reference type host variables in COBOL embedded SQL applications:

File reference type host variables that you declare in your embedded COBOL application are treated as if they were declared in a COBOL program. You can use host variables to exchange data between the embedded application and the database manager.

The syntax for declaring file reference host variables in COBOL is:



- 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).
```

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 foo1.
 10 a pic s9(4) comp-5.
 10 a1 redefines a pic x(2).
 10 b pic x(10).
```

Referring to `foo1` in an SQL statement as follows:

```
... INTO :foo1 ...
```

This statement is equivalent to:

```
... INTO :foo1.a, :foo1.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 :foo1.a1 ...
```

or

```
... INTO :a1 ...
```

Japanese or Traditional Chinese EUC, and UCS-2 considerations for COBOL embedded SQL applications:

Any graphic data that is 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 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 might also consider using the VARCHAR and VARGRAPHIC scalar functions.

Binary storage of variable values using the FOR BIT DATA clause in COBOL embedded SQL applications:

You can declare certain database columns using the FOR BIT DATA clause. 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. You must use these data types when working with columns with the FOR BIT DATA attribute.

Note: The LONG VARCHAR data type is deprecated and might be removed in a future release.

Host structure support in the declare section of COBOL embedded SQL applications:

In an application program, a host structure contains a list of host variables that can be referred to by embedded SQL statements. The COBOL precompiler supports declarations of group data items in the host variable declare section.

Host structure support 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 previously 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 example shown previously. 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 previous example, `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 preceding 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 shown previously.

Null-indicator variables and null or truncation indicator variable tables in COBOL embedded SQL applications:

To receive null values in embedded SQL applications, you must associate null-indicator variables with declared host variables in your embedded SQL applications. A null-indicator variable is shared by both the database manager and the host application.

Null-indicator variables in COBOL must 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 shown previously:

```
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*.

Host variables in FORTRAN

Host variables are FORTRAN language variables that are referenced within SQL statements. Host variables 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.

Host variable names in FORTRAN embedded SQL applications:

The SQL precompiler identifies host variables by their declared name.

When you declare a host variable name, you must consider the following restrictions:

- 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.

Declare section for host variables in FORTRAN embedded SQL applications:

You must use an SQL declare section must be used to identify host variable declarations. The declare section 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.

Example: SQL declare section template for FORTRAN embedded SQL applications:

When you are creating an embedded SQL application in FORTRAN, there is a template that you can use to declare your host variables and data structures.

The following example 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
```

SQLSTATE and SQLCODE variables in FORTRAN embedded SQL application:

To handle errors or debug your embedded SQL application, you should test the values of the SQLCODE or SQLSTATE variable. You can return these values as output parameters or as part of a diagnostic message string, or you can insert these values into a table to provide basic tracing support.

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;
  CHARACTER*5 SQLSTATE
  INTEGER     SQLCOD
  .
  .
  .
EXEC SQL END DECLARE SECTION
```

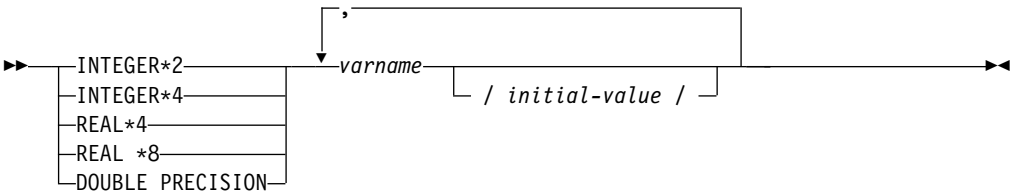
The `SQLCOD` declaration is assumed during the precompile step. The variable named `SQLSTATE` can also be `SQLSTA`. Note that when using this option, the INCLUDE SQLCA statement should not be specified.

For applications that contain multiple source files, the declarations of `SQLCOD` and `SQLSTATE` can be included in each source file, as shown previously.

Declaration of numeric host variables in FORTRAN embedded SQL applications:

Numeric host variables that you declare in your embedded FORTRAN application are treated as if they were declared in a FORTRAN program. Host variables allow you to exchange data between the embedded application and the database manager.

The following illustrates 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.

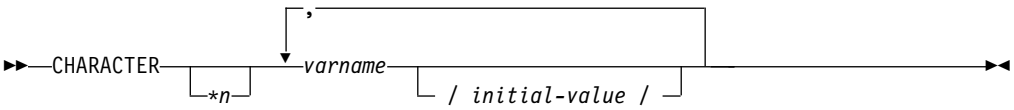
Declaration of fixed-length and variable length character host variables in FORTRAN embedded SQL applications:

You must declare character host variables when you program an embedded SQL application in FORTRAN. Host variables are treated like FORTRAN variables, and allow for the exchange of data between the embedded application and the database manager.

The syntax for fixed-length character host variables is:

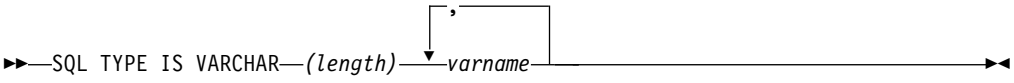
Fixed length

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 can 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 can 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
```

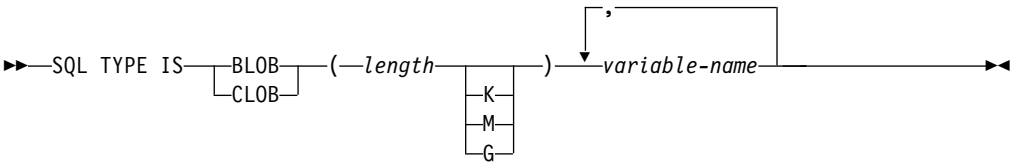
However, because blanks can be significant in passwords, you should declare host variables for passwords as VARCHAR, and have the length field set to reflect the actual password length:

```
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
```

Declaration of large object type host variables in FORTRAN embedded SQL applications:

Large object (LOB) host variables that you declare in your embedded FORTRAN application are treated as if they were declared in a FORTRAN program. Host variables allow you to exchange data between the embedded application and the database manager.

The syntax for declaring large object (LOB) host variables in FORTRAN is:



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
```

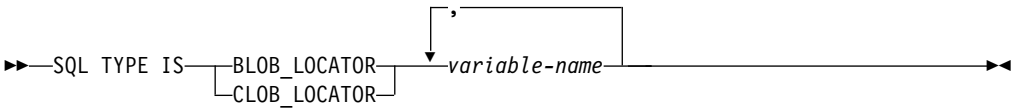
Results in the generation of the following structure:

```
 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 )
```

Declaration of large object locator type host variables in FORTRAN embedded SQL applications:

Large Object (LOB) locator type host variables that you declare in your embedded FORTRAN application are treated as if they were declared in a FORTRAN program. Host variables allow you to exchange data between the embedded application and the database manager.

The syntax for declaring large object (LOB) locator host variables in FORTRAN is:



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
```

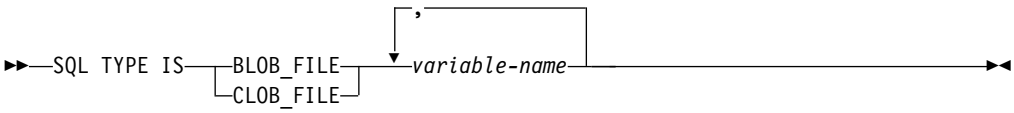
Results in the generation of the following declaration:

```
integer*4 my_locator
```

Declaration of file reference type host variables in FORTRAN embedded SQL applications:

File reference type host variables that you declare in your embedded FORTRAN application are treated as if they were declared in a FORTRAN program. Host variables allow you to exchange data between the embedded application and the database manager.

The syntax for declaring file reference host variables in FORTRAN is:



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 )
```

Considerations for graphic (multi-byte) character sets in FORTRAN embedded SQL applications:

Graphic (multi-byte) host variable data types are not supported in FORTRAN. Only mixed-character host variables are supported through the `character` data type. However, it is possible to create a user SQL descriptor area (SQLDA) that contains graphic data.

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.

Null or truncation indicator variables in FORTRAN embedded SQL applications:

You must declare indicator variables as `INTEGER*2` data types.

Host variables in REXX

Host variables are REXX language variables that are referenced within SQL statements. Host variables allow an application to exchange data with the database manager. After an 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.

Host variable names in REXX embedded SQL applications:

You can use any properly named REXX variable as a host variable. A variable name can be up to 64 characters long and cannot end with a period. A host variable name can consist of numbers, alphabetic characters, and the characters @, _, !, ., ?, and $.

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 example is 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.

Predefined REXX Variables:

The SQLEXEC function and the SQLDBS and SQLDB2 routines set predefined REXX variables as a result of certain operations.

Predefined REXX variables include:

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 or System i® 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.

Considerations while programming REXX embedded SQL applications:

REXX is an interpreted language, which means that no precompiler, compiler, or linker is used. Instead, three Db2 APIs are used to create Db2 applications in REXX. You can use these APIs to access different Db2 elements.

About this task

The three APIs that are available for creating embedded SQL applications in REXX are:

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.

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.

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.

Declaration of large object type host variables in REXX embedded SQL applications:

When you fetch a LOB column into a REXX host variable, it is stored as a string. LOB columns are handled in the same manner as other character-based SQL types such as CHAR, VARCHAR, GRAPHIC, and LONG.

On input, if the size of the contents of your host variable is larger than 32K, or if it meets other criteria that are listed in the following table, it is 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

Host variable string content	Resulting LOB type
	BLOB
:hv4="BIN'string with embedded delimiting single ", "quotation marks, beginning with BIN, any length...'"	

Declaration of large object locator type host variables in REXX embedded SQL applications:

You must declare LOB locator host variables in your application. When a REXX embedded SQL application encounters these declarations the host variables are treated as locators for the remainder of the program. Locator values are stored in REXX variables in an internal format.

The syntax for declaring LOB locator host variables in REXX is:

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

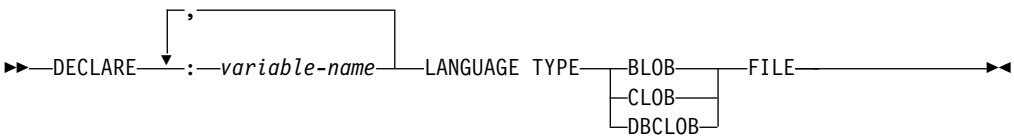
Example:

```
CALL SQLEXEC 'FREE LOCATOR :hv1, :hv2'
```

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 embedded SQL encounters these declarations, it treats the declared host variables as LOB file references for the remainder of the program.

The syntax for declaring LOB file reference host variables in REXX is:



Example:

```
CALL SQLEXEC 'DECLARE :hv3, :hv4 LANGUAGE TYPE CLOB FILE'
```

File reference variables in REXX contain three fields. For the preceding 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 requester 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.

LOB Host Variable Clearing in REXX embedded SQL applications:

On Windows operating systems, it might 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 can 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 include this statement at the end of LOB applications. Note that you can include it anywhere as a precautionary measure to clear declarations which might have been left by previous applications, such as at the beginning of a REXX SQL application.

Null or truncation indicator variables in REXX embedded SQL applications:

An indicator variable data type in REXX is a number without a decimal point.

The 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 previous 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` must not be used. The database manager does not change the value of the host variable in this case.

Considerations for using buffered inserts

Buffered inserts exhibit behaviors that can affect an application program. This behavior is caused by the asynchronous nature of the buffered inserts. Based on the values of the row's distribution key, each inserted row is placed in a buffer destined for the correct partition. These buffers are sent to their destination partitions as they become full, or an event causes them to be flushed. You must be aware of the following, and account for them when designing and coding the application:

- Certain error conditions for inserted rows are not reported when the INSERT statement is executed. They are reported later, when the first statement other than the INSERT (or INSERT to a different table) is executed, such as DELETE, UPDATE, COMMIT, or ROLLBACK. Any statement or API that closes the buffered insert statement can see the error report. Also, any invocation of the insert itself may see an error of a previously inserted row. Moreover, if a buffered insert error is reported by another statement, such as UPDATE or COMMIT, Db2 will not attempt to execute that statement.
- An error detected during the insertion of a *group of rows* causes all the rows of that group to be backed out. A group of rows is defined as all the rows inserted through executions of a buffered insert statement:
  - From the beginning of the unit of work,
  - Since the statement was prepared (if it is dynamic), or
  - Since the previous execution of another updating statement. For a list of statements that close (or flush) a buffered insert, see the description of buffered inserts in partitioned database environments.
- An inserted row may not be immediately visible to SELECT statements issued after the INSERT by the same application program, if the SELECT is executed using a cursor.

A buffered INSERT statement is either open or closed. The first invocation of the statement opens the buffered INSERT, the row is added to the appropriate buffer, and control is returned to the application. Subsequent invocations add rows to the buffer, leaving the statement open. While the statement is open, buffers may be sent to their destination partitions, where the rows are inserted into the target table's partition. If any statement or API that closes a buffered insert is invoked

while a buffered INSERT statement is open (including invocation of a *different* buffered INSERT statement), or if a PREPARE statement is issued against an open buffered INSERT statement, the open statement is closed before the new request is processed. If the buffered INSERT statement is closed, the remaining buffers are flushed. The rows are then sent to the target partitions and inserted. Only after all the buffers are sent and all the rows are inserted does the new request begin processing.

If errors are detected during the closing of the INSERT statement, the SQLCA for the new request will be filled in describing the error, and the new request is not done. Also, the entire group of rows that were inserted through the buffered INSERT statement *since it was opened* are removed from the database. The state of the application will be as defined for the particular error detected. For example:

- If the error is a deadlock, the transaction is rolled back (including any changes made before the buffered insert section was opened).
- If the error is a unique key violation, the state of the database is the same as before the statement was opened. The transaction remains active, and any changes made before the statement was opened are not affected.

For example, consider the following application that is bound with the buffered insert option:

```
EXEC SQL UPDATE t1 SET COMMENT='about to start inserts';
DO UNTIL EOF OR SQLCODE < 0;
  READ VALUE OF hv1 FROM A FILE;
  EXEC SQL INSERT INTO t2 VALUES (:hv1);
  IF 1000 INSERTS DONE, THEN DO
     EXEC SQL INSERT INTO t3 VALUES ('another 1000 done');
     RESET COUNTER;
  END;
END;
EXEC SQL COMMIT;
```

Suppose the file contains 8 000 values, but value 3 258 is not legal (for example, a unique key violation). Each 1 000 inserts results in the execution of another SQL statement, which then closes the `INSERT INTO t2` statement. During the fourth group of 1 000 inserts, the error for value 3 258 will be detected. It may be detected after the insertion of more values (not necessarily the next one). In this situation, an error code is returned for the `INSERT INTO t2` statement.

The error may also be detected when an insertion is attempted on table `t3`, which closes the `INSERT INTO t2` statement. In this situation, the error code is returned for the `INSERT INTO t3` statement, even though the error applies to table `t2`.

Suppose, instead, that you have 3 900 rows to insert. Before being told of the error on row number 3 258, the application may exit the loop and attempt to issue a COMMIT. The unique-key-violation return code will be issued for the COMMIT statement, and the COMMIT will not be performed. If the application wants to COMMIT the 3 000 rows that are in the database thus far (the last execution of `EXEC SQL INSERT INTO t3 ...` ends the savepoint for those 3 000 rows), the COMMIT has to be *reissued*. Similar considerations apply to ROLLBACK as well.

Note: When using buffered inserts, you should carefully monitor the SQLCODES returned to avoid having the table in an indeterminate state. For example, if you remove the `SQLCODE < 0` clause from the THEN DO statement in the above example, the table could end up containing an indeterminate number of rows.

Buffered inserts in partitioned database environments

A buffered insert is an insert statement that takes advantage of table queues to buffer the rows being inserted, thereby gaining a significant performance improvement. To use a buffered insert, an application must be prepared or bound with the INSERT BUF option.

Buffered inserts can result in substantial performance improvement in applications that perform inserts. Typically, you can use a buffered insert in applications where a single insert statement (and no other database modification statement) is used within a loop to insert many rows and where the source of the data is a VALUES clause in the INSERT statement. Typically the INSERT statement is referencing one or more host variables that change their values during successive executions of the loop. The VALUES clause can specify a single row or multiple rows.

Typical decision support applications require the loading and periodic insertion of new data. This data could be hundreds of thousands of rows. You can prepare and bind applications to use buffered inserts when loading tables.

To cause an application to use buffered inserts, use the PREP command to process the application program source file, or use the BIND command on the resulting bind file. In both situations, you must specify the INSERT BUF option.

Note: Buffered inserts cause the following steps to occur:

1. The database manager opens one 4 KB buffer for each database partition on which the table resides.
2. The INSERT statement with the VALUES clause issued by the application causes the row (or rows) to be placed into the appropriate buffer (or buffers).
3. The database manager returns control to the application.
4. The rows in the buffer are sent to the partition when the buffer becomes full, or an event occurs that causes the rows in a partially filled buffer to be sent. A partially filled buffer is flushed when one of the following occurs:
   - The application issues a COMMIT (either explicitly, or implicitly through application termination) or ROLLBACK.
   - The application issues another statement that causes a savepoint to be taken. OPEN, FETCH, and CLOSE cursor statements do not cause a savepoint to be taken, nor do they close an open buffered insert.

     The following SQL statements will close an open buffered insert:
     - BEGIN COMPOUND SQL
     - COMMIT
     - DDL
     - DELETE
     - END COMPOUND SQL
     - EXECUTE IMMEDIATE
     - GRANT
     - INSERT to a different table
     - OPEN CURSOR for a full-select of a data change statement
     - PREPARE of the same dynamic statement (by name) doing buffered inserts
     - REDISTRIBUTE DATABASE PARTITION GROUP
     - RELEASE SAVEPOINT
     - REORG
     - REVOKE
     - ROLLBACK
     - ROLLBACK TO SAVEPOINT

- RUNSTATS
- SAVEPOINT
- SELECT INTO
- UPDATE
- Execution of any other statement, but not another (looping) execution of the buffered INSERT
- End of application

The following APIs will close an open buffered insert:

- BIND (API)
- REBIND (API)
- RUNSTATS (API)
- REORG (API)
- REDISTRIBUTE (API)

In any of these situations where another statement closes the buffered insert, the coordinator partition waits until every database partition receives the buffers and the rows are inserted. It then executes the other statement (the one closing the buffered insert), provided all the rows were successfully inserted.

The standard interface in a partitioned database environment, (without a buffered insert) loads one row at a time doing the following steps (assuming that the application is running locally on one of the database partitions):

1. The coordinator partition passes the row to the database manager that is on the same partition.
2. The database manager uses indirect hashing to determine the database partition where the row should be placed:
   - The target partition receives the row.
   - The target partition inserts the row locally.
   - The target partition sends a response to the coordinator partition.
3. The coordinator partition receives the response from the target partition.
4. The coordinator partition gives the response to the application.

   The insertion is not committed until the application issues a COMMIT.
5. Any INSERT statement containing the VALUES clause is a candidate for buffered insert, regardless of the number of rows or the type of elements in the rows. That is, the elements can be constants, special registers, host variables, expressions, functions and so on.

For a given INSERT statement with the VALUES clause, the Db2 SQL compiler might not buffer the insert based on semantic, performance, or implementation considerations. If you prepare or bind your application with the INSERT BUF option, ensure that it is not dependent on a buffered insert. This means:

- Errors can be reported asynchronously for buffered inserts, or synchronously for regular inserts. If reported asynchronously, an insert error might be reported on a subsequent insert within the buffer, or on the *other* statement that closes the buffer. The statement that reports the error is not executed. For example, consider using a COMMIT statement to close a buffered insert loop. The commit reports an SQLCODE -803 (SQLSTATE 23505) due to a duplicate key from an earlier insert. In this scenario, the commit is not executed. If you want your application to really commit, for example, some updates that are performed before it enters the buffered insert loop, you must reissue the COMMIT statement.
- Rows inserted can be immediately visible through a SELECT statement using a cursor without a buffered insert. With a buffered insert, the rows will not be

immediately visible. Do not write your application to depend on these cursor-selected rows if you precompile or bind it with the INSERT BUF option.

Buffered inserts result in the following performance advantages:

- Only one message is sent from the target partition to the coordinator partition for each buffer received by the target partition.
- A buffer can contain a large number of rows, especially if the rows are small.
- Parallel processing occurs as insertions are being done across partitions while the coordinator partition is receiving new rows.

An application that is bound with INSERT BUF should be written so that the same INSERT statement with VALUES clause is iterated repeatedly before any statement or API that closes a buffered insert is issued.

Note: You should do periodic commits to prevent the buffered inserts from filling the transaction log.

Restrictions on using buffered inserts

The following restrictions apply to buffered inserts:

- For an application to take advantage of the buffered inserts, one of the following must be true:
  - The application must either be prepared through PREP or bound with the BIND command and the INSERT BUF option is specified.
  - The application must be bound using the BIND or the PREP API with the SQL_INSERT_BUF option.
- If the INSERT statement with VALUES clause includes long fields or LOBS in the explicit or implicit column list, the INSERT BUF option is ignored for that statement and a normal insert section is done, not a buffered insert. This is not an error condition, and no error or warning message is issued.
- INSERT with fullselect is not affected by INSERT BUF. A buffered insert does not improve the performance of this type of INSERT.
- Buffered inserts can be used only in applications, and not through CLP-issued inserts, as these are done through the EXECUTE IMMEDIATE statement.

The application can then be run from any supported client platform.

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 by using XQuery expressions.

Before you begin

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 code page conversion.

- Declare XML host variables within your embedded SQL applications.

About this task

- 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 white space (STRIP) handling. Otherwise, an XML host variable is required.

To issue 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".

These examples return data from the XML documents in table CUSTOMER from the sample database.

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 issued 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 (for $a in db2-fn:xmlcolumn("CUSTOMER.INFO")
        /*:customerinfo[*:addr/*:city = "Toronto"]/@Cid return data($a))");

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 and XMLEXISTS predicate

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(data($INFO/*:customerinfo/@Cid)')
FROM customer
WHERE XMLEXISTS('$INFO/*:customerinfo[*:addr/*:city = "Toronto"]');

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 issued 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 (for $a in db2-fn:xmlcolumn("CUSTOMER.INFO")/*:customerinfo
            [*:addr/*:city = "Toronto"]/@Cid return data($a)))" 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.
```

Executing SQL statements in embedded SQL applications

The way you execute SQL statements in embedded SQL applications depends on if the statement is statically or dynamically executed. However, you must use the `EXEC SQL` command regardless of the type of statement that you use.

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 can be prepared with input parameters. Information that is read can 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 toward assisting in troubleshooting an application.

Comments in embedded SQL applications

The comments in any application are important for making the application code understandable.

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.

Executing static SQL statements in embedded SQL applications

You cannot modify static SQL statements at run time. You can use static SQL statements for tasks such as initialization and cleanup.

SQL statements can be executed statically in a host language using the following approach:

- C or C++ (**tbmod.sqc/tbmod.sqC**)

  The following three examples are from the **tbmod** 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 AND job != 'Sales';
  ```

- 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.
```

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. Therefore, for dynamic SQL applications, you must preprocess 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.

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. There are two types of SQLVAR entries: base SQLVAR entries and secondary SQLVAR entries.

About this task

The following diagram describes the structure of the SQLDA.

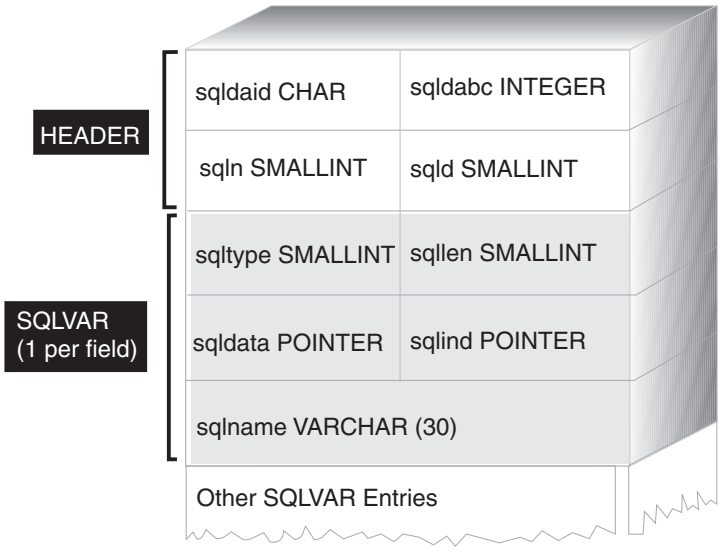


Figure 2. The SQL Descriptor Area (SQLDA)

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:

Procedure

- 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 any of the columns returned has a LOB or user defined type, provide an SQLDA with the exact number of SQLVAR entries.

What to do next

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.

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.

About this task

You can only allocate a smaller SQLDA structure with programming languages, such as C and C++, that support the dynamic allocation of memory.

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 cannot 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.

Allocating an SQLDA structure with sufficient SQLVAR entries for dynamically executed SQL statements:

After you determine the number of columns in the result table, you must allocate storage for a second full-size SQLDA. The first SQLDA is used for input parameters and the second full-size SQLDA is used for output parameters.

About this task

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 items:

Procedure

- 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.

What to do next

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.

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`.

What to do next

Using the DESCRIBE statement is preferred because the costs of preparing the statement a second time are avoided. The DESCRIBE statement 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.

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 tasks:

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.

What to do next

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.

Processing the cursor in a dynamically executed SQL program:

After you allocate the SQLDA structure, you can open the cursor associated with the SELECT statement and fetch rows.

About this task

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 can have following lines:

```
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" ) ;
```

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.

About this task

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

```
      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: This example was written for 32-bit FORTRAN.

In languages not supporting dynamic memory allocation, an SQLDA with the required 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.

Transferring data in a dynamically executed SQL program using an SQLDA structure:

You have greater flexibility when you transfer data using an SQLDA instead of 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.

About this task

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

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)

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.

Processing interactive SQL statements in dynamically executed SQL programs:

You can write an application using dynamic SQL to process arbitrary SQL statements. For example, if an application accepts SQL statements from a user, the application must be able to issue the statements without any prior knowledge of the statements.

About this task

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.

Use the PREPARE and DESCRIBE statements with an SQLDA structure so that the application can determine the type of SQL statement being issued, and act accordingly.

Determination of statement type in dynamically executed SQL programs:

When an SQL statement is prepared, you can determine information concerning the type of statement 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.

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 steps:

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.

Saving SQL requests from end users:

If the users of your application can issue SQL requests from the application, you might want to save these requests.

About this task

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, CLOB, VARGRAPHIC or DBCLOB. Note that the 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.

Providing variable input to dynamically executed SQL statements by using parameter markers

In a dynamic SQL statement, parameter markers that are indicated by a question mark (?) or a colon followed by a name (:*name*) are substituting host variables.

About this task

A dynamic SQL statement cannot contain host variables because host variable information (data type and length) is available only during application precompilation; during execution, host variable information is unavailable. In a dynamic SQL statement, parameter markers are used instead of host variables. A parameter marker is indicated by a question mark (?) or a colon followed by a name (:*name*) and indicates where to substitute a host variable inside an SQL statement.

For example, assume that you want to use a dynamic SQL statement to delete data from a table called TEMPL based on the value of an employee number. You might specify the DELETE statement as follows, using a parameter marker:

```
DELETE FROM TEMPL WHERE EMPNO = ?
```

To execute this statement, specify a host variable or an SQLDA structure for the USING clause of the EXECUTE statement. The contents of the host variable is used to specify the value of EMPNO.

The data type and length of the parameter marker depend on the context of the parameter marker 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 specification to specify the data type. A parameter marker for which you use a CAST specification is a typed parameter marker. A typed parameter marker is treated like a host variable of the data type used in the CAST specification. For example, the statement `SELECT ? FROM SYSCAT.TABLES` is invalid because the data type of the result column is unknown. However, the statement `SELECT CAST(? AS INTEGER) FROM SYSCAT.TABLES` is valid because the CAST specification indicates that the parameter marker represents an INTEGER value; the data type of the result column is known.

If the SQL statement contains more than one parameter marker, the USING clause of the EXECUTE statement must specify one of the following types of information:

- A list of host variables, one variable for each parameter marker
- An SQLDA that has one SQLVAR entry for each parameter marker for non-LOB data types or two SQLVAR entries per parameter marker for LOB data types

The host variable list or SQLVAR entries are matched according to the order of the parameter markers in the statement, and the data types must be compatible.

Note: Using a parameter marker in a dynamic SQL statement is like using a host variable in a static SQL statement in that the optimizer does not use distribution statistics and might not choose the best access plan.

The rules that apply to parameter markers are described in the PREPARE statement topic.

Example of parameter markers in a dynamically executed SQL program:

In the statement string of a dynamic SQL statement, a parameter marker represents a value that will be provided by the application program. The value of a parameter marker is provided on the EXECUTE or OPEN statement that is associated with the dynamic SQL statement.

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.
```

Calling procedures in embedded SQL applications

You can call procedures from embedded SQL applications by formulating and executing the CALL statement with an appropriate procedure reference and parameters. You can issue the CALL statement either statically or dynamically within embedded SQL applications.

However, for each programming language there are different methods to issue this command. No matter which host language, each host variable used in the 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.

Calling stored procedures in C and C++ embedded SQL applications:

You can use the anonymous blocks or the `EXEC SQL CALL` statements to call stored procedures in C and C++ embedded SQL applications.

Calling stored procedures by using the `EXEC SQL CALL` statement

Db2 supports the use of input, output, and input and output parameters in SQL procedures. The IN, OUT, and INOUT keywords 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, you can call an INOUT_PARAM stored procedure by using the following statement:

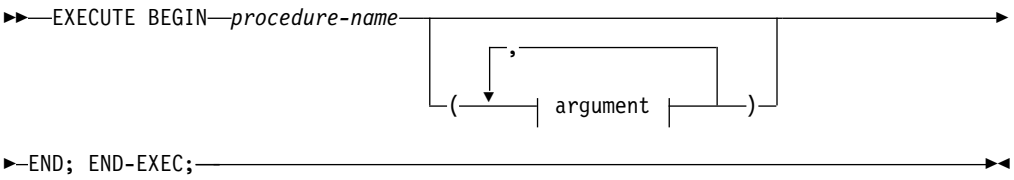
```
EXEC SQL CALL INOUT_PARAM(:inout_median:medianind, :out_sqlcode:codeind,
                          :out_buffer:bufferind);
```

In the previous statement *inout_median*, *out_sqlcode*, and *out_buffer* are host variables and *medianind*, *codeind*, and *bufferind* are null indicator variables.

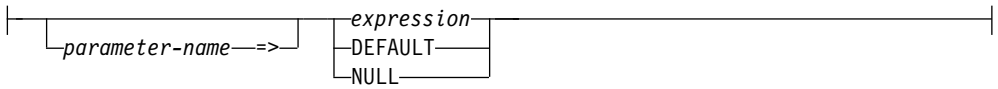
Note: You can also call stored procedures dynamically by preparing a CALL statement.

Calling stored procedures by using the anonymous block

C and C++ embedded SQL applications can call stored procedures by using an anonymous block when the PRECOMPILE option **`COMPATIBILITY_MODE`** is set to `ORA`.



argument:



Parameter description

procedure-name
A name of the procedure, which is described in the catalogue, that you want to call.

argument **description**

parameter-name
The name of the parameter that the argument is assigned to. When you assign an argument to a paramater by name, all the arguments that follow the (parameter) must be assigned by name.

You can only specify a named argument once (implicitly or explicitly).

Named arguments are not supported on a call to an uncataloged procedure.

expression **or DEFAULT or NULL**
Each specification of *expression*, the DEFAULT keyword, or the NULL keyword is an argument of the CALL. The *n*th unnamed argument of the CALL statement corresponds to the *n*th parameter that is defined in the CREATE PROCEDURE statement for the procedure.

Named arguments correspond to the same named parameter, regardless of the order in which arguments are specified.

The DEFAULT keyword is used in the CREATE PROCEDURE statement if you have specified it; otherwise the null value is used as the default.

If the NULL keyword is specified, the null value is passed as the parameter value.

Each argument of the CALL statement must be compatible with the corresponding parameter in the procedure definition as follows:

- IN parameter
  - The argument must be assignable to the parameter.
  - The assignment of a string argument uses the storage assignment rules.
- OUT parameter
  - The argument must be a single variable or parameter marker.
  - The argument must be assignable to the parameter.
  - The assignment of a string argument uses the retrieval assignment rules.
- INOUT parameter
  - The argument must be a single variable or parameter marker.
  - The argument must be assignable to the parameter.
  - The assignment of a string argument uses the storage assignment rules on invocation and the retrieval assignment rules on return.

Calling stored procedures from REXX:

You can write stored procedures in any language that is supported on a server, except for REXX on AIX operating systems.

Client applications can be written in REXX on AIX operating systems, but as with other languages, client applications cannot call a stored procedure written in REXX on AIX.

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. You can retrieve data by 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.

Scrolling through previously retrieved data in embedded SQL applications:

When an application retrieves data from the database, the FETCH statement allows you to scroll forward through the data. There is no equivalent SQL statement that you can use to scroll backwards through the result set.

You can use the CLI and the Db2 JDBC Driver to run 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.

Keeping a copy of fetched data in embedded SQL applications:

In some situations, it might 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 one of the following tasks:

- 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.

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, issue 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 statement:

  ```
  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.

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 can occur even if you were to issue the same SQL statement, with the same host variables, a second time. For example, the statistics in the catalog can be updated between executions, or indexes can be created or dropped. You can then issue 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 can choose to use an index on the new predicate. For example, it can 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 can choose an index on LOCATION for both statements. Yet changing the value of DEPTNO in the second statement to the following example can 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.

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 issue one statement many times with different values of the variables.

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*.

About this task

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

To process a cursor:

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.

What to do next

An application can use several cursors concurrently. Each cursor requires its own set of DECLARE CURSOR, OPEN, CLOSE, and FETCH statements.

Updating and deleting retrieved data in statically executed SQL applications:

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.

About this task

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.

Example of a fetch in a statically executed SQL program:

A fetch is an SQL action that positions a cursor on the next row of its result table and assigns the values of that row to host variables.

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 must be deleted or updated.

The REXX language does not support static SQL, so a sample is not provided.

- C and C++ (**tbmod.sqc/tbmod.sqC**)

  The following 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 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.
```

Error message retrieval in embedded SQL applications

The method that you use to retrieve error information depends on the language that you used to write the application.

- 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;
}
```

C developers can also use an equivalent function, sqlglm(), which has the signature:

```
sqlglm(char *message_buffer_ptr, int *buffer_size_ptr, int *msg_size_ptr)
```

```
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
```

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. The SQLWARN field contains an array of warning indicators, even if SQLCODE is zero.

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 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.

Exit list routine considerations:

You must not use SQL or Db2 API calls in exit list routines.

Note: You cannot disconnect from a database in an exit routine.

Exception, signal, and interrupt handler considerations:

An exception, signal, or interrupt handler is a routine that gains control when an exception, signal, or interrupt occurs. The type of handler used is determined by your operating environment.

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 that SIGINT can be generated by a different key sequence on your machine.

Do not put SQL statements 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. Before issuing a ROLLBACK, call the INTERRUPT API (`sqleintr/sqlgintr`). This API interrupts the current SQL query (if the application is executing one) and lets the ROLLBACK begin immediately.

Refer to your platform documentation for specific details on the various handler considerations.

How to disconnect from embedded SQL applications

The disconnect statement is the final step you must take when you are working with a database.

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;
```

Applications that are precompiled with the **COMPATIBILITY_MODE** ORA option can issue one of the following statements to achieve the ROLLBACK operation along with the disconnect operation in a single statement:

- `EXEC SQL ROLLBACK RELEASE;`
- `EXEC SQL ROLLBACK WORK RELEASE;`

Applications that are precompiled with the **COMPATIBILITY_MODE** ORA option can issue one of the following statements to achieve the COMMIT operation along with the disconnect operation in a single statement:

- `EXEC SQL COMMIT RELEASE;`
- `EXEC SQL COMMIT WORK RELEASE;`

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
```

Embedded SQL/COBOL Support for MRI and MRF

Db2 Precompiler supports the SQL array in INSERT, UPDATE, and DELETE. It allows the application to insert/update/delete the rows in the target database from the COBOL application. Also, it supports the FETCH statement to fetch the multiple rows from the specified cursor from the server.

Supporting Array INSERT/UPDATE/DELETE

The Db2 ESQL enables arrays to be passed as a host variable, while ESQL calls the INSERT, UPDATE, and DELETE statements as their input arguments from the COBOL application. Also, Db2 embedded runtime supports the bulk insert against Db2 except LOB columns.

To pass a cardinality for the array insert, the Db2 supports the “For n ROWS” clause in INSERT, UPDATE, and DELETE SQL’s. The variable "n" within the "For n ROWS" clause can be an integer in the range 2 - 32767, a host variable declared as integer, or short data type.

If the “For n ROWS” clause is not specified, the Db2 precompiler for COBOL takes the cardinality of the array size based on the declared size of

host variables that are used in the SQL. If the host variables used are of different sizes, the minimum size of all the host variables are used as the cardinality of the bulk/array INSERT.

Users can check sqlca.sqlerrd(2) when an error occurs during an array operation. Processing stops at the row that caused the error. Thus, sqlerrd[2] gives the row number on which error occurred. Users can check sqlca.sqlerrd(3), which indicates the number of impacted records as a result of the array INSERT/UDPATE/DELETE operation.The following example demonstrates the array INSERT through COBOL.

```
Identification Division.
Program-ID. "arrayfetch".

Data Division.
Working-Storage Section.

   copy "sqlca.cbl".

   EXEC SQL BEGIN DECLARE SECTION END-EXEC.
01 cnt           pic s9(4) comp-5.
01 insert-rec.
  03 c1                pic x(18) OCCURS 5 TIMES.
  03 c2                pic s9(9) comp-5 OCCURS 5 TIMES.
    EXEC SQL END DECLARE SECTION END-EXEC.

77 errloc          pic x(80).

Procedure Division.
Main Section.
  display "Sample COBOL program: ARRAY INSERT".

  EXEC SQL CONNECT TO sample END-EXEC

  MOVE 5 to cnt.
  MOVE "Row1" to c1(1).
  MOVE "Row2" to c1(2).
  MOVE "Row3" to c1(3).
  MOVE "Row4" to c1(4).
  MOVE "Row5" to c1(5).
  MOVE 1 to c2(1).
  move 10 to c2(2).
  MOVE 50 to c2(3).
  MOVE 100 to c2(4).
  MOVE 500 to c2(5).
  EXEC SQL INSERT INTO test VALUES (:c2, :c1
      ) FOR :cnt ROWS END-EXEC.

  EXEC SQL CONNECT RESET END-EXEC.
  move "CONNECT RESET" to errloc.
  call "checkerr" using SQLCA errloc.
End-Main.
  go to End-Prog.

End-Prog.
  stop run.
```

Support of Multirow Fetch

Db2 Precompiler and embedded Runtime support the fetch of multiple rows by using a single FETCH statement. Db2 ESQL enables arrays to be passed as a host variable (and as their indicators) while ESQL calls the FETCH statements as their output arguments (bind-outs) from the COBOL application.

To pass the cardinality for the array FETCH statement, the Db2 Precompiler supports the “For n ROWS” clause in FETCH statement. The variable "n" in the "For n ROWS" clause can be an integer in the range 2 - 32767, a host variable declared as integer, or a short data type.

If the “For n ROWS” clause is not specified, the Db2 precompiler for COBOL takes the cardinality of the array size based on the declared size of host variables that are used in the SQL. If the host variables used are of different sizes, the minimum size of all the host variables are used as the cardinality of the bulk/array.

Users can check sqlca.sqlerrd(3), which indicates the number of records fetched so far on this cursor.The following example demonstrates the array FETCH through COBOL.

```
Identification Division.
Program-ID. "openftch".

Data Division.
Working-Storage Section.

  copy "sqlca.cbl".
  EXEC SQL BEGIN DECLARE SECTION END-EXEC.
01 dept-rec.
  03 pname             pic x(10) OCCURS 5 TIMES.
  03 dept              pic s9(9) comp-5 OCCURS 5 TIMES.
  03 cnt               pic s9(9) comp-5.
01 userid            pic x(8).
01 passwd.
  49 passwd-length   pic s9(4) comp-5 value 0.
  49 passwd-name     pic x(18).

  EXEC SQL END DECLARE SECTION END-EXEC.

77 errloc          pic x(80).

Procedure Division.
Main Section.
  display "Sample COBOL program: OPENFTCH".

  EXEC SQL CONNECT TO sample END-EXEC

  MOVE 5 TO cnt.
  EXEC SQL DECLARE c1 CURSOR FOR SELECT name, dept         1
      FROM staff
      WHERE job='Mgr' END-EXEC.

  EXEC SQL OPEN c1 END-EXEC.                               2
  move "OPEN" to errloc.
  call "checkerr" using SQLCA errloc.

*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.                              5
  move "CLOSE" to errloc.
  call "checkerr" using SQLCA errloc.

  EXEC SQL ROLLBACK END-EXEC.
  move "ROLLBACK" to errloc.
  call "checkerr" using SQLCA errloc.
  display "On second thought -- changes rolled back.".

  EXEC SQL CONNECT RESET END-EXEC.
  move "CONNECT RESET" to errloc.
```

```
  call "checkerr" using SQLCA errloc.
End-Main.
  go to End-Prog.

Fetch-Loop Section.
  EXEC SQL FETCH c1 FOR :cnt ROWS INTO :pname,
              :dept END-EXEC.

  display pname(1), " in dept", dept(1), "will be fetched".
  display pname(2), " in dept", dept(2), "will be fetched".
  display pname(3), " in dept", dept(3), "will be fetched".
  display pname(4), " in dept", dept(4), "will be fetched".
  display pname(5), " in dept", dept(5), "will be fetched".
  display "blank line.....................,,,,,,,,,,".

End-Fetch-Loop. exit.

End-Prog.
  stop run.
```

Array declaration by using the OCCURS clause

COBOL supports the declaration of array's by using the OCCURS clause. Multirow INSERT and FETCH is supported for array when it is declared as the following

```
01 Monthly-sales-rec.
          03 Monthly-sales     pic s9(9) comp-5 OCCURS 12 TIMES.
```

The previous declartion specifices 12 fields, all of which have the same PIC. The individual fields are referenced by using subscripts such as MONTHLY-SALES(1).

Restrictions

- The Multi-row fetch and array INSERT do not support the LOB array's. The Db2 precompiler throws SQL1727N error, if the application uses the LOB array's in INSERT/UPDATE/DELETE/FETCH statements.
- The Multi-row fetch and array insert do not support table of records (array of structure).
- The Db2 precompiler throws SQL0104N error if the application uses the NON-ATOMIC keyword in the INSERT statement because the Db2 server does not support this keyword.
- The Db2 embedded SQL supports running the array insert/update/delete operations in atomic mode only.
- The Db2 Precompiler does not consider the declaration of array of structure/record.
- The Db2 server and client does not support ROWSET cursors. Hence the Db2 precompiler strips off the WITH ROWSET POSITIONING keyword from the DECLARE CURSOR statement, and the NEXT ROWSET keyword from the FETCH statement.

Building embedded SQL applications

After you have created the source code for your embedded SQL application, you must follow additional steps to build the application. You should consider building 64-bit executable files 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 runtime 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.

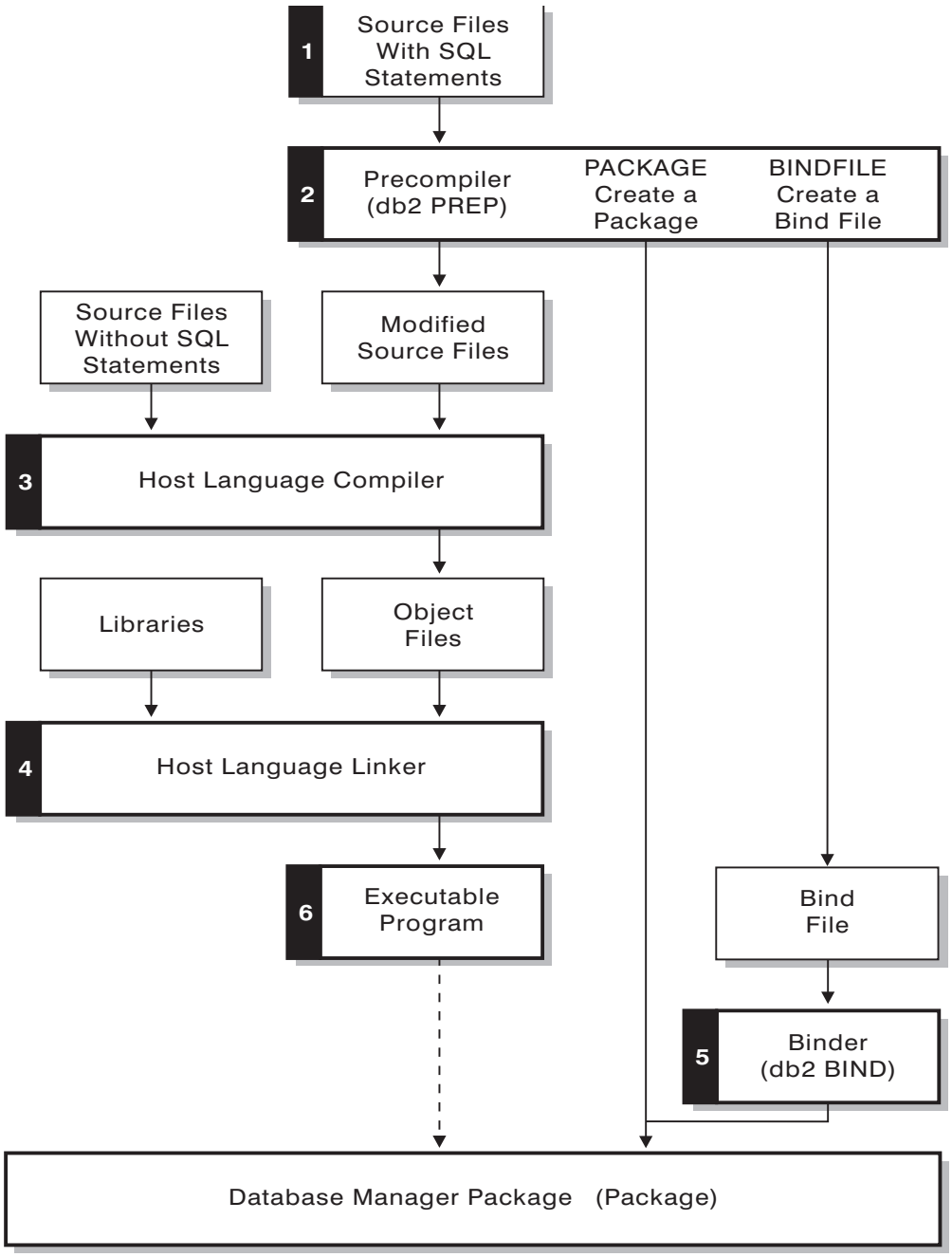


Figure 3. Preparing Programs Written in Compiled Host Languages

Precompilation of embedded SQL applications with the PRECOMPILE command

After you create the source files for an embedded SQL application, 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 runtime 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*.

Restriction: Running an embedded application on an older client version than the client where precompilation occurred is not supported, regardless of where the application was compiled. For example, it is not supported to precompile an embedded application on a Db2 V9.5 client and then attempt to run the application on a Db2 V9.1 client.
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 might need to use the **FUNCPATH** parameter when you precompile your application. This parameter 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
```

Restriction: The byte order mark (BOM) with UTF-8 for a C embedded SQL source file is not supported.

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 runtime API calls. It is given the appropriate host language extension.

Package
: If you use the **PACKAGE** parameter (the default), or do not specify any of the **BINDFILE**, **SYNTAX**, or **SQLFLAG** parameters, the package is stored in the connected database. The package contains all the information required to issue the static SQL statements of a particular source file against this database only. Unless you specify a different name with the **PACKAGE USING** parameter, the precompiler forms the package name from the first 8 characters of the source file name.

If you use the **PACKAGE** parameter 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** parameter, the bind file (if the **BINDFILE** parameter is used) and the package (either if bound at **PREP** time or if bound separately) is 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** parameter, 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** parameter, 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** parameter. Thus, if you are using the CLP and want the binding to be deferred, you need to specify the **BINDFILE** parameter.

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 issue them at run time generates an error.

Message File

If you use the **MESSAGES** parameter, 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** parameter, precompilation messages are written to the standard output.

Precompilation of embedded SQL applications that access more than one database server

You must write your embedded SQL applications such that the application is able to distinguish which database server receives each SQL statement.

To precompile an application program that accesses more than one server, you can do one of the following tasks:

- 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.

Embedded SQL application packages and access plans

The precompiler produces a package in the database. The package contains access plans selected by the Db2 optimizer for the static SQL statements in your application. You can optionally specify if you also want a bind file generated.

The access plans contain the information required by the database manager to issue 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 issue 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.

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 for z/OS Version 9.1 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 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. Db2 for IBM i 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 (that is, accepting a single schema name), a SET CURRENT PACKAGESET statement have to be issued before trying to invoke each package to specify the specific schema. For this example, five SET CURRENT PACKAGESET statements 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, 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 for z/OS, Version 8 or later, supports the SET CURRENT PACKAGE PATH statement. If you issue this statement against a Db2 server or against Db2 for IBM i, -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 for 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 bind process stores a timestamp in the bind file. Db2 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 for 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 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.

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** parameter on the **PREP** command. As well, the version ID will be "" (an empty string) unless it is specified by the **VERSION** parameter 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 be executed from packages SCHEMA1.PKG versions VER1 and VER2. 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.

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.

Compiling and linking source files containing embedded SQL

You can precompile embedded SQL programs using the `PRECOMPILE` command. You must then compile and link the resultant modified source files with the appropriate host language compiler.

About this task

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 objects:

- 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.

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. You can also display the precompile options used to create the bind file using the Db2 bind file description (**db2bfd**) utility. This can be useful in problem determination related to the bind file for your application.

You can set the STATICASDYNAMIC string on the **GENERIC** parameter of the **BIND** command to "yes" to instruct the Db2 database manager to store all statements in the catalogs and mark them as incremental bind. At run time, when the package is first loaded, the database manager uses the current session environment (rather than the package) to set up the section entries and other entities (text is populated and the package cache is accessed). Thereafter, the statements in the bound file behave the same as they would if you were using dynamic SQL. For example, sections will be implicitly recompiled for Database Definition Language invalidations, special register updates, and so on. The Db2 database manager provides this feature to facilitate the migration of embedded SQL C applications from other database systems.

Effect of DYNAMICRULES bind option on dynamic SQL

The **PRECOMPILE** command and **BIND** command parameter **DYNAMICRULES** determines which rules apply to dynamic SQL at run time.

In particular, the **DYNAMICRULES** parameter 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 runtime environment of a package controls how dynamic SQL statements behave at run time. The two possible runtime environments are:

- The package runs as part of a stand-alone program

- The package runs within a routine context

The combination of the **DYNAMICRULES** value and the runtime 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 the Db2 database) 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 the Db2 database.
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 runtime environment that yields each dynamic SQL behavior.

Table 18. How DYNAMICRULES and the Runtime 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 19. 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 **`BIND OWNER`** command parameter	ID of User Executing Package	Routine definer (not the routine's package owner)	Current statement authorization ID when routine is invoked.
Default qualifier for unqualified objects	The implicit or explicit value of the **`BIND QUALIFIER`** command parameter	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

Using special registers to control the statement compilation environment

For dynamically prepared statements, the special registers can be specified to define the statement compilation environment.

Following special registers controls 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.

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** command parameter is specified. The **PACKAGE** command parameter 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** parameter on the **PREP** command. The version of a package is given by the **VERSION** precompile parameter 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).

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 were marked invalid or inoperative or if the database statistics changed since the last binding.

In some situations, however, you might want to rebind packages that are valid. For example, you might want to take advantage of a newly created index, or use 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.

When the package is marked inoperative, the next use of a statement in this package causes an implicit rebind of the package using non-conservative binding semantics in order to be able to resolve to SQL objects considering the latest changes in the database schema that caused that package to become inoperative.

For static DML in packages, the packages can rebind implicitly, or by explicitly issuing the **REBIND** command (or corresponding API), or the **BIND** command (or

corresponding API). The implicit rebind is performed with conservative binding semantics if the package is marked invalid, but uses non-conservative binding semantics when the package is marked inoperative.

You must use the **BIND** command to rebind a package for a program which was 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. The **REBIND** command provides the option to resolve with conservative binding semantics (**RESOLVE CONSERVATIVE**) or to resolve by considering new routines, data types, or global variables (**RESOLVE ANY**, which is the default option). The **RESOLVE CONSERVATIVE** option can be used only if the package was not marked inoperative by the database manager (SQLSTATE 51028). 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.

In IBM Data Studio Version 3.1 or later, you can use the task assistant for rebinding packages. Task assistants can guide you through the process of setting options, reviewing the automatically generated commands to perform the task, and running these commands. For more details, see Administering databases with task assistants.

Bind considerations

If your application uses a code page that differs from the database code page, you must ensure that the code page used by the application is compatible with the database code page during the bind process.

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 parameter takes advantage of a specific optimization class when binding.
- The **EXPLSNAP** bind parameter stores Explain Snapshot information for eligible SQL statements in the Explain tables.
- The **FUNCPATH** bind parameter properly resolves user-defined distinct types and user-defined functions in static SQL.

If the bind process starts but never returns, it might 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** command parameters available for that server.

Bind files are not compatible with earlier versions of Db2. 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 IBM data server client to connect to the higher-level server and create bind files which can be shipped and bound to the lower-level Db2 environment.
- Use a higher-level IBM data server 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.

Blocking considerations

When you want to turn blocking off for an embedded SQL application and the source code is not available, the application must be rebound using the **BIND** command and setting the **BLOCKING NO** clause.

Existing embedded SQL applications must be rebound using the **BIND** command and setting the **BLOCKING ALL** or **BLOCKING UNAMBIGUOUS** clauses to request blocking (if they are not already bound in this fashion). Embedded applications will retrieve the LOB values from the server a row at a time, when a block of rows have been retrieved from the server

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 database.

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.

Performance improvements when using REOPT option of the BIND command

The bind option **REOPT** can significantly improve the embedded SQL application performance.

Effects of REOPT on static SQL

The bind option **REOPT** can make static SQL statements containing host variables, global 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, global variable, and special register values is used to create this plan.

Effects of REOPT on dynamic SQL

When you specify the option **REOPT ALWAYS**, the database manager postpones preparing any statement containing host variables, parameter markers, global variables, 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. When **REOPT ALWAYS** is specified, statement concentrator is disabled.

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, global variables, and special registers. This plan is cached and will be used by subsequent requests.

Binding applications and utilities (Db2 Connect Server)

Application programs developed using embedded SQL must be bound to each database with which they will operate. For information about the binding requirements for the IBM data server package, see the topic about Db2 CLI bind files and package names.

Binding should be performed once per application, for each database. During the bind process, database access plans are stored for each SQL statement that will be executed. These access plans are supplied by application developers and are contained in *bind files* which are created during precompilation. Binding is a process of processing these bind files by an IBM mainframe database server.

Because several of the utilities supplied with Db2 Connect are developed using embedded SQL, they must be bound to an IBM mainframe database server before they can be used with that system. If you do not use the Db2 Connect utilities and interfaces, you do not have to bind them to each of your IBM mainframe database servers. The lists of bind files required by these utilities are contained in the following files:

- `ddcsmvs.lst` for System z
- `ddcsvse.lst` for VSE
- `ddcsvm.lst` for VM
- `ddcs400.lst` for IBM Power Systems™

Binding one of these lists of files to a database will bind each individual utility to that database.

If a Db2 Connect Server product is installed, the Db2 Connect utilities must be bound to each IBM mainframe database server before they can be used with that

system. Assuming the clients are at the same fix pack level, you need to bind the utilities only once, regardless of the number of client platforms involved.

For example, if you have 10 Windows clients, and 10 AIX clients connecting to Db2 for z/OS via Db2 Connect Enterprise Edition on a Windows server, perform one of the following steps:

- Bind `ddcsmvs.lst` from one of the Windows clients.
- Bind `ddcsmvs.lst` from one of the AIX clients.
- Bind `ddcsmvs.lst` from the Db2 Connect server.

This example assumes that:

- All the clients are at the same service level. If they are not then, in addition, you might need to bind from each client of a particular service level
- The server is at the same service level as the clients. If it is not, then you need to bind from the server as well.

In addition to Db2 Connect utilities, any other applications that use embedded SQL must also be bound to each database that you want them to work with. An application that is not bound will usually produce an SQL0805N error message when executed. You might want to create an additional bind list file for all of your applications that need to be bound.

For each IBM mainframe database server that you are binding to, perform the following steps:

1. Make sure that you have sufficient authority for your IBM mainframe database server management system:

   System z

   The authorizations required are:

   - SYSADM or
   - SYSCTRL or
   - BINDADD *and* CREATE IN COLLECTION NULLID

   Note: The BINDADD and the CREATE IN COLLECTION NULLID privileges provide sufficient authority **only** when the packages do not already exist. For example, if you are creating them for the first time.

   If the packages already exist, and you are binding them again, then the authority required to complete the task(s) depends on who did the original bind.

   A) If you did the original bind and you are doing the bind again, then having any of the previously listed authorities will allow you to complete the bind.

   B) If your original bind was done by someone else and you are doing the second bind, then you will require either the SYSADM or the SYSCTRL authorities to complete the bind. Having just the BINDADD and the CREATE IN COLLECTION NULLID authorities will not allow you to complete the bind. It is still possible to create a package if you do not have either SYSADM or SYSCTRL privileges. In this situation you would need the BIND privilege on each of the existing packages that you intend to replace.

VSE or VM

The authorization required is DBA authority. If you want to use the GRANT option on the bind command (to avoid granting access to each Db2 Connect package individually), the NULLID user ID must have the authority to grant authority to other users on the following tables:

- system.syscatalog
- system.syscolumns
- system.sysindexes
- system.systabauth
- system.syskeycols
- system.syssynonyms
- system.syskeys
- system.syscolauth
- system.sysuserauth

On the VSE or VM system, you can issue:

```
grant select on table to nullid with grant option
```

IBM Power Systems

*CHANGE authority or higher on the NULLID collection.

2. Issue commands similar to the following commands:

```
db2 connect to DBALIAS user USERID using PASSWORD
db2 bind path@ddcsmvs.lst blocking all
      sqlerror continue messages ddcsmvs.msg grant public
db2 connect reset
```

Where *DBALIAS*, *USERID*, and *PASSWORD* apply to the IBM mainframe database server, `ddcsmvs.lst` is the bind list file for z/OS, and *path* represents the location of the bind list file.

For example *drive*`:\sqllib\bnd\` applies to all Windows operating systems, and *INSTHOME*`/sqllib/bnd/` applies to all Linux and UNIX operating systems, where *drive* represents the logical drive where Db2 Connect was installed and *INSTHOME* represents the home directory of the Db2 Connect instance.

You can use the grant option of the **bind** command to grant EXECUTE privilege to PUBLIC or to a specified user name or group ID. If you do not use the grant option of the **bind** command, you must GRANT EXECUTE (RUN) individually.

To find out the package names for the bind files, enter the following command:

```
ddcspkgn @bindfile.lst
```

For example:

```
ddcspkgn @ddcsmvs.lst
```

might yield the following output:

```
Bind File                     Package Name
----------------------------- ------------------------------
f:\sqllib\bnd\db2ajgrt.bnd    SQLAB6D3
```

To determine these values for Db2 Connect execute the **ddcspkgn** utility, for example:

```
ddcspkgn @ddcsmvs.lst
```

Optionally, this utility can be used to determine the package name of individual bind files, for example:

```
ddcspkgn bindfile.bnd
```

Note:

a. Using the bind option **sqlerror continue** is required; however, this option is automatically specified for you when you bind applications using the Db2 tools or the Command Line Processor (CLP). Specifying this option turns bind errors into warnings, so that binding a file containing errors can still result in the creation of a package. In turn, this allows one bind file to be used against multiple servers even when a particular server implementation might flag the SQL syntax of another to be invalid. For this reason, binding any of the list files `ddcsxxx.lst` against any particular IBM mainframe database server should be expected to produce some warnings.

b. If you are connecting to a Db2 database through Db2 Connect, use the bind list `db2ubind.lst` and do not specify **sqlerror continue**, which is only valid when connecting to a IBM mainframe database server. Also, to connect to a Db2 database, it is recommended that you use the Db2 clients provided and not Db2 Connect.

3. Use similar statements to bind each application or list of applications.
4. If you have remote clients from a previous release of Db2, you might need to bind the utilities on these clients to Db2 Connect.

Package storage and maintenance

You can create packages by precompiling and binding an application program. The package contains an optimized access plan that 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.

Package versioning

If you need to create multiple versions of an application, you can use the **VERSION** parameter in 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 that you have an application called `foo1`, which is compiled from `foo1.sqc`. You would precompile and bind the package `foo1` 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 `foo1.sqc`, then repeat the process of recompiling, binding, and sending the application to the users. If the **VERSION** parameter was not specified for either the first or second precompilation of `foo1.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 `foo1`, precompile it using the **VERSION** parameter, as follows:

```
DB2 PREP FOO1.SQC VERSION V1.1
```

This first version of the program may now be run. When you build the new version of `foo1`, precompile it with the command:

```
DB2 PREP FOO1.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** parameter of the **PRECOMPILE** or **BIND** commands with the **VERSION** parameter of the **PRECOMPILE** command. You use the **ACTION** parameter 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 `foo1` 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** parameter to accomplish package versioning. Instead, you should use different package names (either by renaming the updated source file, or by using the **PACKAGE USING** parameter to explicitly rename the package).

Resolution of unqualified table names

You can handle unqualified table names in your application by binding user packages with **COLLECTION** parameters, or by creating views or aliases.

Use one of the following methods to handle unqualified table names:

- 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 this 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 typically 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 a public alias to point to the required table.
- 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 required table.

Building embedded SQL applications using the sample build script

The files used to demonstrate building sample programs are called script files on UNIX and Linux operating systems, and batch files on Windows operating systems. These files are collectively called build files and 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 20. 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 sqlpl 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 21. Build files by language and platform

Platform —> Language	AIX	HP-UX	Linux	Solaris	Windows
C samples/c	bldapp bldrtn bldmt bldmc	bldapp bldrtn bldmt bldmc	bldapp bldrtn bldmt bldmc	bldapp bldrtn bldmt bldmc	bldapp.bat bldrtn.bat bldmt.bat bldmc.bat
C++ samples/cpp	bldapp bldrtn bldmt bldmc	bldapp bldrtn bldmt bldmc	bldapp bldrtn bldmt bldmc	bldapp bldrtn bldmt bldmc	bldapp.bat bldrtn.bat bldmt.bat bldmc.bat
IBM COBOL samples/cobol	bldapp bldrtn	n/a	n/a	n/a	bldapp.bat bldrtn.bat
Micro Focus COBOL samples/cobol_mf	bldapp bldrtn	bldapp bldrtn	bldapp bldrtn	bldapp bldrtn	bldapp.bat 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 previously) 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.

Error-checking utilities

The Db2 Client provides several utility files. The utility files contain functions that you can use 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 CLI utility file, `utilcli.c`, does not use these Db2 APIs; instead it uses equivalent 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 22. 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 samples/c	utilapi.c	utilapi.h	utilemb.sqc	utilemb.h
C++ samples/cpp	utilapi.C	utilapi.h	utilemb.sqC	utilemb.h
IBM COBOL samples/cobol	checkerr.cbl	n/a	n/a	n/a
Micro Focus COBOL 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 file. Both the `makefile` and build files in the `samples` directories do this for the programs that require the error-checking utilities.

The 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 example demonstrates 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.

Building applications and routines written in C and C++

You are provided with build scripts for various operating systems with your Db2 product. You can build embedded SQL applications in C and C++ with these files.

Aside from build scripts that you can use to build applications, there is a specific `bldrtn` script provided that you can use to build routines, such as 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.

Compile and link options for C and C++:

AIX C embedded SQL and Db2 API applications compile and link options:

The compile and link options for building C embedded SQL and Db2 API applications with the IBM C for AIX compiler are available 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.

AIX C++ embedded SQL and Db2 administrative API applications compile and link options:

The compile and link options for building C++ embedded SQL and Db2 administrative API applications with the IBM XL C/C++ for AIX compiler are available 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.

Linux C application compile and link options:

The compile and link options for building C embedded SQL and Db2 API applications with the Linux C compiler are available 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 flags:

- `-m31` on Linux for zSeries only, to build a 32-bit library;
- `-m32` on Linux for x86, x64 and POWER, to build a 32-bit library;
- `-m64` on Linux for zSeries, POWER, x64, 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 flags:
- `-m31` on Linux for zSeries only, to build a 32-bit library;
- `-m32` on Linux for x86, x64 and POWER, to build a 32-bit library;
- `-m64` on Linux for zSeries, POWER, x64, 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.

Linux C++ application compile and link options:

The compile and link options for building C++ embedded SQL and Db2 API applications with the Linux C++ compiler are available 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 flags:
- `-m31` on Linux for zSeries only, to build a 32-bit library;

- `-m32` on Linux for x86, x64 and POWER, to build a 32-bit library;
- `-m64` on Linux for zSeries, POWER, x64, 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:

g++ Use the compiler as a front end for the linker.

$EXTRA_C_FLAGS
Contains one of the following flags:
- `-m31` on Linux for zSeries only, to build a 32-bit library;
- `-m32` on Linux for x86, x64 and POWER, to build a 32-bit library;
- `-m64` on Linux for zSeries, POWER, x64, 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 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.

Windows C and C++ application compile and link options:

The compile and link options for building C and C++ embedded SQL and Db2 API applications on Windows with the Microsoft Visual C++ compiler are available 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.

Building applications in C or C++ using the sample build script (UNIX):

You are provided with build scripts for compiling and linking embedded SQL and Db2 administrative API programs in C or C++. The scripts are in the `sqllib/samples/c` directory for C applications and the `sqllib/samples/cpp` directory for C++ applications. The directories include sample programs that you can build with these files.

About this task

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.

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, 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 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.

About this task

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

- **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:

  – If connecting to the sample database on the same instance, enter:

    ```
    bldapp tbmod
    ```

  – If connecting to another database on the same instance, also enter the database name:

    ```
    bldapp tbmod database
    ```

  – 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:

  – If accessing the sample database on the same instance, enter the executable name:

    ```
    tbmod
    ```

  – If accessing another database on the same instance, enter the executable name and the database name:

    ```
    tbmod database
    ```

  – 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 stand-alone 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:

  – If accessing the sample database on the same instance, simply enter the executable name (without the extension):

```
dbthrds
```

- If accessing another database on the same instance, enter the executable name and the database name:

  ```
  dbthrds database
  ```

- 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
  ```

Example

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 C/C++ multi-connection applications on Windows:

You can use the provided build scripts to compile and link C and C++ embedded SQL and Db2 API programs. The scripts are in the `sqllib\samples\c` and `sqllib\samples\cpp` directories. The directories also contain sample programs that you can build with these files.

You can find the commands to build a Db2 multi-connection program in the `bldmc.bat` batch file. The commands require two databases. The compile and link options are the same as those used in the `bldapp.bat` file.

About this task

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`) 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, for the first database; and `%6` and `%7` to specify the user ID and password, for the second database.

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.

Procedure

To ensure that the TCP/IP listener is running:

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 TCPIP_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
   ```

Results

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.

Building applications and routines written in COBOL

You are provided with build scripts for various operating systems for your Db2 product. You can build embedded SQL applications written in COBOL with these files.

Aside from build scripts that you can use to build applications there is a specific `bldrtn` script so that you can build routines, such as 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'.

Compile and link options for COBOL:

IBM COBOL for AIX application compile and link options:

The compile and link options for building COBOL embedded SQL and Db2 API applications with the IBM COBOL for AIX compiler are available 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.

AIX Micro Focus COBOL application compile and link options:

The compile and link options for building COBOL embedded SQL and Db2 API application with the Micro Focus COBOL for AIX compiler are available in the `bldapp` build script.

Note that the Db2 Micro Focus COBOL include files are found by setting up the COBCPY environment variable, so no -I flag is required 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.

Linux Micro Focus COBOL application compile and link options:

These compile and link options are available 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.

Windows IBM COBOL application compile and link options:

The compile and link options for building COBOL embedded SQL and Db2 API applications on Windows operating systems with the IBM VisualAge COBOL compiler are available 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.

Windows Micro Focus COBOL application compile and link options:

The compile and link options for building COBOL embedded SQL and Db2 API applications on Windows operating systems with the Micro Focus COBOL compiler are available 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.

COBOL compiler configurations:

Configuring the IBM COBOL compiler on AIX:

Before you develop IBM COBOL applications that contain embedded SQL and Db2 API calls on AIX operating systems, you must configure the IBM COBOL compiler.

About this task

Required steps 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 **PRECOMPILE** command, 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 z 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 z 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".
  ```

Configuring the IBM COBOL compiler on Windows:

When you develop an embedded SQL application with the IBM VisualAge COBOL compiler on Windows operating system, following `db2 prep` option and compiler options must be set.

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".
  ```

Configuring the Micro Focus COBOL compiler on Windows:

When you develop an embedded SQL application with the Micro Focus COBOL compiler on Windows operating system, following `db2 prep` option and environment settings must be set.

Procedure

- When you precompile your application using the **PRECOMPILE** command, 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 previously 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.

What to do next

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 example, whenever the precompiler generates a Db2 API runtime 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 previously. If you are calling a Db2 API directly, then you will need to manually add the DB2API symbol after the "call" keyword.

Configuring the Micro Focus COBOL compiler on Linux:

To run Micro Focus COBOL routines, you must ensure that the Linux runtime linker and Db2 processes can access the dependent COBOL libraries in the `/usr/lib` directory.

About this task

Create symbolic links to `/usr/lib` for the COBOL shared libraries as root. The simplest way to create symbolic links to `/usr/lib` 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 procedures need to be done on each Db2 instance:

Procedure

- When you precompile your application using the **PRECOMPILE** command, 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"
  ```

Results

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`. .

Configuring the Micro Focus COBOL compiler on AIX:

Before you develop Micro Focus COBOL applications that contain embedded SQL and Db2 API calls on AIX operating systems, you must configure the Micro Focus COBOL compiler.

About this task

Follow the listed steps 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 **PRECOMPILE** command, 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 are 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.

Building IBM COBOL applications on AIX:

You can use the provided build scripts for compiling and linking IBM COBOL embedded SQL and Db2 administrative API programs. The scripts are in the `sqllib/samples/cobol` directory. The directories also contain sample programs that you can build with these files.

You can find commands to build a Db2 application program in the `bldapp` build script.

About this task

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.

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
```

Procedure

- 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, 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
     ```

Building UNIX Micro Focus COBOL applications:

You are provided with build scripts for compiling and linking Micro Focus COBOL embedded SQL and Db2 administrative API programs. You can find the scripts in the `sqllib/samples/cobol_mf` directory. The directory also contains sample programs that you can build with these files.

You can find the commands to build a Db2 application program in the `bldapp` build file.

About this task

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.

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
```

Procedure

- 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, 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
     ```

Building IBM COBOL applications on Windows:

You can use the provided build scripts to compile and link Db2 API and embedded SQL programs. The scripts are in the sqllib\samples\cobol directory. The directory also contains sample programs that you can build with these files.

About this task

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.

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
```

Procedure

- 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, 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
   ```

Building Micro Focus COBOL applications on Windows:

You can use build scripts provided with IBM data server client for compiling and linking Db2 API and embedded SQL programs.

Build scripts are in the `sqllib\samples\cobol_mf` directory, along with sample programs that can be built with these build script files.

About this task

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.

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
```

Procedure

- 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, 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
     ```

Building and running embedded SQL applications written in REXX

REXX applications are not precompiled, compiled, or linked. You can build and run REXX applications on Windows operating systems, and on the AIX operating system.

About this task

On Windows operating systems, 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

To build and run your REXX applications:

- 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 line 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
```

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 is 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 or System i database servers. On other databases, it behaves such as 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.

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.

About this task

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`.

To run the sample REXX program `updat`, enter:

```
rexx updat.cmd
```

Building embedded SQL applications from the command line

There are different methods that you can use to build embedded SQL applications, such as by using build scripts or the command line. You can use the command line if you want to test build options before writing a script to automate the process.

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.

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.

About this task

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.

- 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 preceding example, 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
```

Deploying and running embedded SQL applications

Embedded SQL applications are portable and can be placed in remote database components. You can compile the application in one location and run the package on a different component.

Use of the db2dsdriver.cfg configuration file by embedded SQL applications

Embedded SQL applications support use of the IBM data server driver configuration file (`db2dsdriver.cfg`) for high availability solutions with supported servers.

You can use the IBM data server driver configuration file (`db2dsdriver.cfg`) for work load balancing (WLB) and automatic client reroute (ACR) with supported servers. The WLB and ACR associated keywords are available for use with the embedded applications.

Table 23. Settings to control workload balancing behavior

Element in the db2dsdriver.cfg configuration file	Section	Value
connectionLevelLoadBalancing	<database>	Must be set to true if you want to use transaction-level workload balancing. The value is true by default. However, the default is false if the server accessed is Db2 for z/OS .
enableWLB	<wlb>	Specifies whether transaction-level workload balancing is in effect. The value is false by default.
maxTransportIdleTime	<wlb>	Specifies the maximum elapsed time in number of seconds before an idle transport is dropped. The default is 60 seconds. The minimum supported value is 0.
maxTransportWaitTime	<wlb>	Specifies the number of seconds that the client waits for a transport to become available. The default is 1 second. The minimum supported value is 0 and -1 is used to specify unlimited value.

Table 23. Settings to control workload balancing behavior (continued)

Element in the db2dsdriver.cfg configuration file	Section	Value
maxTransports	<wlb>	Specifies the maximum number of physical connections that can be made for each application process that connects to the Db2 pureScale® instance. The default is -1 (unlimited). However, the default is 1000 if the server accessed is Db2 for z/OS.
maxRefreshInterval	<wlb>	Specifies the maximum elapsed time in number of seconds before the server list is refreshed. The default is 10 seconds. The minimum supported value is 0.

Table 24. Settings to control automatic client reroute behavior

Element in the <acr> section of the db2dsdriver configuration file	Value
enableAcr	Specifies whether automatic client reroute is in effect. The default is `true`. If the server accessed is Db2 for z/OS, the **enableAcr** parameter should be enabled only when the **enableWLB** parameter is in effect.
acrRetryInterval	The number of seconds to wait between consecutive connection retries. The registry variable **DB2_CONNRETRIES_INTERVAL** overrides this value. The valid range is 0 to MAX_INT. The default is no wait (0), if **DB2_CONNRETRIES_INTERVAL** is not set. When enabling automatic client reroute to the Db2 for z/OS data sharing group, the default value of no wait is recommended.
maxAcrRetries	The maximum number of connection retries for automatic client reroute. The registry variable **DB2_MAX_CLIENT_CONNRETRIES** overrides this value. If **DB2_MAX_CLIENT_CONNRETRIES** is not set, the default is that the connection is tried again for 10 minutes. A value of 0 means that one attempt at reconnection is made. If the server accessed isDb2 for z/OS, the **maxAcrRetries** is recommended to be set to no higher than 5.
enableAlternateServerListFirstConnect	Specifies whether there is an alternate server list that is used only if a failure occurs on the first connection to the data server. The default is false. When the value of **enableAlternateServerListFirstConnect** is true, automatic client reroute with seamless failover is implicitly enabled, regardless of the other settings that are specified for automatic client reroute in the `db2dsdriver` configuration file. To use this feature, you also require an <alternateserverlist> element in the `db2dsdriver` configuration file. This parameter is not supported against Db2 for z/OS.
alternateserverlist	Specifies a set of server names and port numbers that identify alternate servers to which a connection is attempted if a failure occurs on the first connection to the database. The alternate server list is not used after the first connection. In a Db2 pureScale environment, the entries in the list can be members of a Db2 pureScale instance. In a non-Db2 pureScale environment, there is an entry for the primary server and an entry for the high availability disaster recovery (HADR) standby server. The alternate server list is not used after the first connection.
affinityFailbackInterval	The number of seconds to wait after the first transaction boundary to fail back to the primary server. Set this value if you want to fail back to the primary server. The default is 0, which means that no attempt is made to fail back to the primary server.
affinitylist	<list> elements with serverorder attributes. The serverorder attribute value specifies a list of servers, in the order that they should be tried during automatic client reroute with client affinities. The servers in <list> elements must also be defined in <server> elements in the <alternateserverlist>. You can specify multiple <list> elements, each of which has different server orders. The presence of the <affinitylist> element does not activate automatic client reroute.

Table 24. Settings to control automatic client reroute behavior (continued)

Element in the <acr> section of the db2dsdriver configuration file	Value
clientaffinitydefined	<client> elements that define the server order for automatic client reroute for each client. Each <client> element contains a listname attribute that associates a client with a <list> element from the <affinitylist> element.
clientaffinityroundrobin	<client> elements whose order in the <clientaffinityroundrobin> element defines the first server that is chosen for automatic client reroute. Each <client> element has an index. The first <client> element in the <clientaffinityroundrobin> element has index 0, the second <client> element has index 1, and so on. Suppose that the number of servers in the <alternateserverlist> element is *n* and the index in the <clientaffinityroundrobin> element of a <client> element is *i*. The first server to be tried is the server whose index in the <alternateserverlist> element is *i* mod *n*. The next server to be tried is the server whose index in the <alternateserverlist> element is (*i* +1) mod *n*), and so on.

You can use the IBM data server driver configuration file (`db2dsdriver.cfg`) to set the client information registers. You can set the client information register keywords in the <dsn>, <database>, or <parameters> section of the IBM data server driver configuration file. The list of supported IBM data server driver keywords for client information are listed in Table 3.

Table 25. Supported client information keywords

Keywords	Header
ClientAccountingString	Sets the client accounting string that is sent to a database.
ClientApplicationName	Sets the client application name that is sent to a database.
ClientCorrelationToken[1]	Sets the client correlation token that is sent to a Db2 for z/OS Version 11 server in new function mode (NFM).
ClientUserID	Sets the client user ID (accounting user ID) that is sent to a database. The **ClientUserID** keyword is for identification purposes only and is not used for any authentication.
ClientWorkstationName	Sets the client workstation name that is sent to a database server.
enableDefaultClientInfo[1]	Specifies whether the default client information register values, which are set on the Db2 for z/OS server, are returned to the application.

[1] You can set the specified keyword only when you are connecting to the Db2 for z/OS server.

The embedded SQL application cannot perform seamless failover. Also, the ability to resolve the data source name (DSN) with <dsncollection> section entry in the `db2dsdriver.cfg` file is only supported with IBM Data Server Driver Package.

IBM data server clients supports the use of the <dsncollection> section entry in the `db2dsdriver.cfg` file to resolve DSN entry.

The following steps outline the process involved with database alias resolution:

1. The embedded SQL application requests to CONNECT to the database alias.
2. The embedded SQL application looks up the catalog database directory to see if the specified database alias name exists.
   - If information is found, the embedded application uses the database name, host name, and port number information from the catalog. Proceed to step 4.

- If information is not found, the <dsncollection> sections in the `db2dsdriver.cfg` file is used to resolve the database alias name to the database name, host name, and port number information.

3. The application looks for database alias information in the `db2dsdriver.cfg` file:
   - If database alias information is not found, a database connection error is returned to the embedded SQL application.
   - If database alias information is found, the database name, host name, port number, and data server driver parameters that are specified in the <dsn> section are used.
4. Using the database name, host name, and port number, the <databases> section for matching entry is searched.
5. If a matching entry for the database name, host name, and port number is found in the <databases> section, the parameters specified under the matching <database> section is applied to the connection.
6. The database connection is attempted with information that is specified in the catalog and `db2dsdriver.cfg` file.

In Db2 Version 9.7 Fix Pack 6 and later fix packs, the embedded SQL application can use following timeout values and connection parameters in the `db2dsdriver.cfg` file:

- **MemberConnectTimeout**
- **ReceiveTimeout**
- **TcpipConnectTimeout**
- **keepAliveTimeOut**
- **ConnectionTimeout**
- **CommProtocol**
- **Authentication**
- **SSLClientLabel**
- **TargetPrincipal**
- **SecurityTransportMode**
- **SSLclientkeystoredb**
- **SSLclientkeystash**
- **SSLClientKeystoredbPassword**

Any unrecognized data server keywords are ignored silently by the embedded SQL application.

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. These libraries are 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.

Compatibility features for migration

The Db2 database manager provides features that facilitate the migration of embedded SQL C applications from other database systems.

You can enable these compatibility features by setting the precompiler option COMPATIBILITY_MODE to ORA. For example, the following command enables the compatibility features when you compile the file named `tbsel.sqc`:

```
$ db2 PRECOMPILE tbsel.sqc BINDFILE COMPATIBILITY_MODE ORA
```

The following features are supported when the `COMPATIBILITY_MODE ORA` precompile option is specified:

- Ability to specify the RELEASE option with EXEC SQL ROLLBACK and EXEC SQL COMMIT statements.
- Enhanced CONNECT statement syntax.
- Simple type definition for the VARCHAR type.
- Suppression of unspecified indicator variable error when the required NULL indicator is not specified and the `UNSAFENULL YES` option of the **PRECOMPILE** command is not set.
- Use of anonymous block to call a stored procedure.
- Use of C and C++ host variable arrays for FETCH INTO, INSERT, UPDATE, and DELETE statements that are non-dynamic.
- Use of double quotation marks to specify file names with the INCLUDE statement.
- Use of INDICATOR variable arrays for FETCH INTO, INSERT, UPDATE, and DELETE statements that are non-dynamic.
- Use of structure type, structure arrays, and structure indicator arrays.

Additionally, the following features are supported for C or C++ embedded SQL applications even if you do not issue the **PRECOMPILE** command with the `COMPATIBILITY_MODE ORA` option:

- Use of the `STATICASDYNAMIC` string for the **GENERIC** parameter of the **BIND** command to provide true dynamic SQL behavior for the package that is bound in a session.
- Use of a string literal with the PREPARE statement.
- Use of three-part name to call a stored procedure.
- Use of WHENEVER *condition* DO *action* statement.

C and C++ host variable arrays

You can use C and C++ host variable arrays for FETCH INTO, INSERT, UPDATE, and DELETE statements that are non-dynamic, when you set the precompiler option `COMPATIBILITY_MODE` to `ORA`.

For a host variable array that is declared for an INSERT, UPDATE, or DELETE statement, you must ensure that entire array elements are initialized with a value. Otherwise, unexpected data can get introduced or removed from the table.

When you specify multiple host variable arrays for one database object in an INSERT, UPDATE, or DELETE statement, you must declare the same cardinality for those arrays. Otherwise, the smallest cardinality that is declared among the arrays is used.

The total number of rows that are successfully processed is stored in the sqlca.sqlerrd[3] field. However, the sqlca.sqlerrd[3] field does not represent the number of rows that are committed successfully in the case of INSERT, UPDATE, or DELETE operations.

The total number of rows that are affected by the INSERT, UPDATE, or DELETE operation is stored in the sqlca.sqlerrd[2] field.

In the following example, host variable arrays *arr_in1* and *arr_in2* demonstrate the use of the sqlca.sqlerrd[2] and sqlca.sqlerrd[3] fields:

```
// Declaring host variables with cardinality of 5.
EXEC SQL BEGIN DECLARE SECTION;
  sqlint32 arr_in1[5];
  char arr_in2[5][11];
EXEC SQL END DECLARE SECTION;
...
// Populating the arrays.
for ( i = 0; i < 5; i++)
{
    arr_in1[i] = i + 1;
    sprintf(arr_in2[i], "hello%d", i + 1);
}

// A duplicate value is introduced for arr_in1 array.
// arr_in1[0]==arr_in1[4]
arr_in1[4] = 1;

// The C1 column in the table tbl1 requires an unique key
// and doesn't allow duplicate values.

EXEC SQL INSERT into tbl1 values (:arr_in1, :arr_in2);
printf("sqlca.sqlcode = %d\n", sqlca.sqlcode ); // -803

// Since arr_in1[0] and arr_in1[4] have identicle values,
// the INSERT operation fails when arr_in1[4] element is
// processed for the INSERT operation (which is 5th row
// insert attempt).
// The INSERT operation successfully processed 4 rows (not committed).
printf("sqlca.sqlerrd[3] = %d\n", sqlca.sqlerrd[3] ); //Prints 4

// The INSERT operation failed and 0 rows are impacted.
printf("sqlca.sqlerrd[2] = %d\n", sqlca.sqlerrd[2] ); //Prints 0

// No rows are present in tbl1 as the INSERT operation failed.
// C1          C2
//----------- -----------
// 0 record(s) selected.
```

Use of C or C++ host variable arrays in FETCH INTO statements

You can declare a cursor and do a bulk fetch into a variable array until the end of the row is reached. Host variable arrays that are used in the same FETCH INTO statement must have same cardinality. Otherwise, the smallest declared cardinality is used for the array.

In one FETCH INTO statement, the maximum number of records that can be retrieved is the cardinality of the array that is declared. If more rows are available after the first fetch, you can repeat the FETCH INTO statement to obtain the next set of rows.

In the following example, two host variable arrays are declared; *empno* and *lastname*. Each can hold up to 100 elements. Because there is only one FETCH INTO statement, this example retrieves 100 rows, or less.

```
// Declaring host variables
EXEC SQL BEGIN DECLARE SECTION;
    char   empno[100][8];
    char   lastname[100][15];
EXEC SQL END DECLARE SECTION;

EXEC SQL DECLARE empcr CURSOR FOR
 SELECT empno, lastname FROM employee;

EXEC SQL OPEN empcr;

EXEC SQL WHENEVER NOT FOUND GOTO end_fetch;

while (1) {
 EXEC SQL FETCH empcr INTO :empno :lastname;  /* bulk fetch       */
 ...                                          /* 100 or less rows */
 ...
}
end_fetch:
EXEC SQL CLOSE empcr;
```

Use of C or C++ host variable arrays in INSERT statements

In the following example, host variable arrays *arr_in1* and *arr_in2* are used for an INSERT statement:

```
// Declaring host variables.
EXEC SQL BEGIN DECLARE SECTION;
  sqlint32 arr_in1[3];
  char arr_in2[3][11];
EXEC SQL END DECLARE SECTION;

...
// Populating the arrays.
for ( i = 0; i < 3; i++)
{
    arr_in1[i] = 100 + i;
    sprintf(arr_in2[i], "hello%d", arr_in1[i]);
}

// The 'arr_in1' & 'arr_in2' are host variable arrays.
EXEC SQL INSERT into tbl1 values (:arr_in1, :arr_in2);
printf("sqlca.sqlcode = %d\n", sqlca.sqlcode ); // 0

// The INSERT operation inserted 3 rows without encounting an error.
printf("sqlca.sqlerrd[3] = %d\n", sqlca.sqlerrd[3] ); // 3

// The INSERT operation was successful and 3 rows has been stored in database.
printf("sqlca.sqlerrd[2] = %d\n", sqlca.sqlerrd[2] ); // 3

// The tb11 table now contains the following rows:
//C1          C2
//----------- -----------
//        100 hello1
//        101 hello2
//        102 hello3
```

Use of C or C++ host variable arrays in UPDATE statements

In the following example, host variable arrays *arr_in1* and *arr_in2* are used for an UPDATE statement:

```
// Declaring host variables
EXEC SQL BEGIN DECLARE SECTION;
  sqlint32 arr_in1[3];
  sqlint32 arr_in2[2];
EXEC SQL END DECLARE SECTION;

...
// Populating the arrays.
for ( i = 0; i < 3; i++)
{
   arr_in1[i] = 100 + i;
```

```
}

arr_in2[0] = 1000;
arr_in2[1] = 1001;

// Table tbl2 consists of following rows before an update statement is issued.
//C1          C2
//----------- -----------
//        100        500
//        101        501
//        102        502

// The 'arr_in1' array is declared with cardinality of 3 for use in the
// SET clause of an UPDATE statement.
// The 'arr_in2' array is declared with cardinality of 2 for use in the
// WHERE clause of an UPDATE statement.
// The tbl2 table contains 3 rows.
// The following UPDATE statement will affect only 2 rows as per arr_in2
// for column c2 and remaining need to be untouched.

// The 'arr_in1' array in the following update statement is treated as
// having cardinality of 2.
EXEC SQL UPDATE tbl2 SET c2 = :arr_in2 + c2 where c1 = :arr_in1;
printf("sqlca.sqlcode = %d\n", sqlca.sqlcode ); // 0

// As there is no error in update statement, sqlca.sqlerrd[3]
// contains rows which are updated successfully.
printf("sqlca.sqlerrd[3] = %d\n", sqlca.sqlerrd[3] ); // 2

// update successful and 2 rows has been updated in database.
printf("sqlca.sqlerrd[2] = %d\n", sqlca.sqlerrd[2] ); // 2

// The tb12 table now contains the following rows:
//C1          C2
//----------- -----------
//        100        1500
//        101        1502
//        102         503
```

Use of C or C++ host variable arrays in DELETE statements

In the following example, host variable arrays *arr_in1* and *arr_in2* are used for a DELETE statement:

```
// Declaring host variables
EXEC SQL BEGIN DECLARE SECTION;
  sqlint32 arr_in1[3];
EXEC SQL END DECLARE SECTION;
...
// Populating the arrays.
for ( i = 0; i < 3; i++)
{
   arr_in1[i] = 101 + i;
}

// Initial data in the tbl2 table:
// C1          C2
// ----------- -----------
//         100         500
//         101         501
//         102         502
//         103         503
//         104         504
// using array host while executing delete statement in where clause
// The'arr_in1' host variable array is used in the WHERE clause of
// an DELETE statement.

EXEC SQL DELETE FROM tbl2 where c1 = :arr_in1;
printf("sqlca.sqlcode = %d\n", sqlca.sqlcode ); // 0

// delete successful attempted rows are 3
printf("sqlca.sqlerrd[3] = %d\n", sqlca.sqlerrd[3] ); // 3

// delete successful and 3 rows has been deleted in database.
printf("sqlca.sqlerrd[2] = %d\n", sqlca.sqlerrd[2] ); // 3

// The tb12 table now contains the following rows:
// C1          C2
// ----------- -----------
//         100         500
//         104         504
```

Restrictions with C or C++ host variable array support

The use of a C or C++ host variable array in embedded SQL applications is subject to the following restrictions:

- Host variables arrays are supported by C or C++ embedded SQL applications that connect to Db2 servers.
- Host variable arrays must be declared in the `DECLARE SECTION` with exact size of the array elements (cardinality).
- Specific array element cannot be specified in a SQL statement.
- The INSERT, UPDATE, or DELETE operation with host variable arrays is run as an atomic operation on the database server. If any array element causes an SQL_ERROR, current transaction is rolled back.
- Use of host variable arrays are not supported by dynamically prepared INSERT, UPDATE, or DELETE statements.
- Maximum size of array element (cardinality) is 32672.
- The following C and C++ data types are not supported for use with host variable arrays:
  - Another host variable array (nesting)
  - BLOB
  - BLOB file reference
  - BLOB locator variable
  - CLOB
  - CLOB file reference
  - CLOB locator variable
  - User-defined data type
  - XML
- From Db2 V11.1 M3 FP3 onwards, FOR N ROWS clause can be used to specify the cardinality for INSERT and MERGE statement, where N can be an integer or a host variable of type int or short. If array host variables are used, it will take the minimum cardinality value among all the host variables that are used in the SQL.
- Host variable array support is not provided for Db2 for z/OS and Db2 for i servers.

Call to a stored procedure with anonymous block

C and C++ embedded SQL applications can call a stored procedure with use of the anonymous block when the PRECOMPILE option **`COMPATIBILITY_MODE`** is set to `ORA`.

The following example calls a stored procedure, INOUT_PARAM, with use of the anonymous block:

```
EXEC SQL EXECUTE BEGIN INOUT_PARAM(:inout_median:medianind,
   :out_sqlcode:codeind, :out_buffer:bufferind); END; END-EXEC;
```

The *inout_median*, *out_sqlcode*, and *out_buffer* are host variables and *medianind*, *codeind*, and *bufferind* are null indicator variables.

Remember: The embedded SQL applications do not support returning the values from a stored procedure.

Call to a stored procedure with a three-part name

C and C++ embedded SQL applications can call a stored procedure with a three-part name when the connected database server supports use of three-part names.

A three-part name consists of *schema_name.module_name.procedure_name*, or *schema_name.package_name.procedure_name*. The Db2 Version 9.7 and later server supports use of three-part names, containing a package name, only if the **DB2_COMPATIBILITY_VECTOR** registry variable is set to ORA.

```
EXEC SQL CALL schema_name.module_name.procedure_name(:parm);
EXEC SQL CALL schema_name.package_name.procedure_name(:parm);
```

CONNECT statement syntax enhancements

You can specify additional CONNECT statement syntax, when you set the precompiler option COMPATIBILITY_MODE to ORA.

In your C and C++ embedded SQL applications, a database connection can be established with one of the following CONNECT statements:

- `EXEC SQL CONNECT TO dbname;`
- `EXEC SQL CONNECT TO dbname USER username USING password;`

When you set the precompiler option COMPATIBILITY_MODE to ORA, you can also specify the following CONNECT statement syntax:

```
EXEC SQL CONNECT [ username IDENTIFIED BY password ][ USING  dbname ] ;
```

The parameters are described in the following table:

Parameter	Description
username	Either a host variable or a string that specifies the database user name
password	Either a host variable or a string that specifies the password
dbname	Either a host variable or a string that specifies the database name

Declaration of the VARCHAR data type

You can declare host variables with VARCHAR data type, when you set the precompiler option COMPATIBILITY_MODE to ORA.

The following declaration of the VARCHAR data type is supported. The precompiler expands the VARCHAR data type into the equivalent C struct type:

```
EXEC SQL BEGIN DECLARE SECTION;
   VARCHAR var_name [n+1];
EXEC SQL END DECLARE SECTION;
```

Include-file names with double quotation marks

You can use double quotation marks to specify include-file names in the INCLUDE directives, when you set the precompiler option COMPATIBILITY_MODE to ORA.

In the following example, an INCLUDE directive was used with double quotation marks:

```
EXEC SQL INCLUDE "abc.h";
```

You can use only single quotation marks without setting the precompiler option COMPATIBILITY_MODE to ORA.

Indicator variable arrays

You can use indicator arrays for FETCH INTO, INSERT, UPDATE, and DELETE statements that are non-dynamic, when you set the precompiler option COMPATIBILITY_MODE to ORA.

An indicator variable array is a short data type variable that is associated with a specific host variable array or a structure array. Each indicator variable element in the indicator variable array can contain 0 or -1 value that indicates whether an associated host variable or structure contains a null value. If an indicator variable value is less than zero, it identifies the corresponding array value as NULL.

In FETCH INTO statements, you can use indicator variable arrays to determine whether any elements of array variables are null.

You can use the keyword **INDICATOR** to identify an indicator variable, as shown in the example.

In the following example, the indicator variable array that is called *bonus_ind* is declared. The *bonus_ind* indicator variable array can have up to 100 elements, the same cardinality as the *bonus* array variable. When the data is being fetched, if the value of *bonus* is NULL, the value in *bonus_ind* is negative.

```
EXEC SQL BEGIN DECLARE SECTION;
    char   empno[100][8];
    char   lastname[100][15];
    short  edlevel[100];
    double bonus[100];
    short  bonus_ind[100];
EXEC SQL END DECLARE SECTION;

EXEC SQL DECLARE empcr CURSOR FOR
 SELECT empno, lastname, edlevel, bonus
  FROM employee
  WHERE workdept = 'D21';

EXEC SQL OPEN empcr;

EXEC SQL WHENEVER NOT FOUND GOTO end_fetch;

while (1) {
 EXEC SQL FETCH empcr INTO :empno :lastname :edlevel,
         :bonus INDICATOR :bonus_ind
 ...
 ...
}
end_fetch:
EXEC SQL CLOSE empcr;
```

Instead of being identified by the **INDICATOR** keyword, an indicator variable can immediately follow its corresponding host variable, as shown in the following example:

```
EXEC SQL FETCH empcr INTO :empno :lastname :edlevel, :bonus:bonus_ind
```

In the following example, the indicator variable arrays *ind_in1* and *ind_in2* are declared. It can have up to three elements, the same cardinality as the *arr_in1* and *arr_in2* array variables. If the value of *ind_in1* or *ind_in2* is negative, the NULL value is inserted for the corresponding *arr_in1* or *arr_in2* value.

```
// Declare host & indicator variablesof array size 3
EXEC SQL BEGIN DECLARE SECTION;
  sqlint32 arr_in1[3];
  char arr_in2[3][11];
  short ind_in1[3]; // indicator array size is same as host
                    // variable's array size
  short ind_in2[3]; // note here indicator array size is greater
                    // than host variable's array size
EXEC SQL END DECLARE SECTION;

...
// Populating the arrays.
for ( i = 0; i < 3; i++)
{
    arr_in1[i] = i + 1;
    sprintf(arr_in2[i], "hello%d", arr_in1[i]);
}

ind_in1[0] = 0;
ind_in1[1] = SQL_NULL_DATA; // Mark it as a NULL data
ind_in1[2] = 0;

ind_in2[0] = 0;
ind_in2[1] = 0;
ind_in2[2] = SQL_NULL_DATA; // Mark it as a NULL data


// 'arr_in1' & 'arr_in2' are host variable arrays
// 'ind_in1' & 'ind_in2' are indicator variable arrays
EXEC SQL INSERT into tbl1 values (:arr_in1 :ind_in1, :arr_in2 :ind_in2);

// The tbl1 table now contains the following rows:
C1          C2
----------- -----------
          1 hello1
            hello2  // c1 is set to NULL as indicator is set
          3         // c2 is set to NULL as indicator is set
```

If the cardinality of indicator variable array does not match the cardinality of the corresponding host variable array, an error is returned.

In the following example, the indicator structure array *MyStructInd* is declared.

```
// declaring indicator structure array of size 3
EXEC SQL BEGIN DECLARE SECTION;
   ...

   struct MyStructInd
   {
       short c1_ind;
       short c2_ind;
   } MyStructVarInd[3];
EXEC SQL END DECLARE SECTION;

...

// using structure array host variables & indicators structure type
// array while executing FETCH statement
// 'MyStructVar' is structure array for host variables
// 'MyStructVarInd' is structure array for indicators
EXEC SQL FETCH cur INTO :MyStructVar :MyStructVarInd;
```

Important: The following conditions must be met when the indicator structure array is used.

- The cardinality of the indicator structure array must be equal to or greater than the cardinality of the structure array.
- All members in the indicator structure array must use the short data type.
- The number of members in the indicator structure array must match the number of members in the corresponding structure array.
- For INSERT, UPDATE and DELETE operations, application must ensure that all indicator variables are initialized with either 0 or `SQL_NULL_DATA` (`-1`).

The total number of rows that are successfully processed is stored in the `sqlca.sqlerrd[3]` field. However, the `sqlca.sqlerrd[3]` field does not represent successfully committed number of rows in the case of INSERT, UPDATE, or

DELETE operations. The total number of rows that are impacted by the INSERT, UPDATE, or DELETE operation is stored in the sqlca.sqlerrd[2] field.

RELEASE option in ROLLBACK statements and COMMIT statements

You can specify the ROLLBACK or COMMIT operation with the connection reset operation in a single statement, when you set the precompiler option COMPATIBILITY_MODE to ORA.

Whenever a new RELEASE option is used, an application must reestablish the connection before any statement is issued on the same connection.

The following example resets connection for ROLLBACK option:

```
EXEC SQL ROLLBACK RELEASE;
EXEC SQL ROLLBACK WORK RELEASE;
```

The following example resets connection for COMMIT option:

```
EXEC SQL COMMIT RELEASE;
EXEC SQL COMMIT WORK RELEASE;
```

Remember: The PRECOMPILE command returns the SQL1696N error if you use the new syntax without setting the **COMPATIBILITY_MODE** parameter to ORA.

Strings for the GENERIC option on the BIND command

Description

The command line process command BIND contains the GENERIC parameter. The GENERIC parameter specifies a string that contains any option-value pairs. Each option and value must be separated by one or more blank spaces. The syntax for this string is displayed as follows:

```
GENERC 'option1 value1 option2 value2 '
```

The option-values available for the Linux, Unix, and Windows (LUW) platforms are the following strings:

- HV_EXPANSION_FACTOR <1|2|3|4>
- STATICASDYNAMIC [YES|NO]

HV_EXPANSION_FACTOR <1|2|3|4>

This option applies a multiplier to CHAR and VARCHAR host variables. It is also expand character host variable lengths within database server to accomdate code page conversion expansion in unequal codepage environment.

You should use this string in order to resolve -302 errors occurring in static SQL applications running in unequal codepage environments even after table definitions increased to accommodate expansion. This is because static SQL by default uses the defined length of the host variable provided by the client during BIND to size the equivalent database server memory location for the variable.

Assume there's a 819 codepage application with string **"niño".** This requires a 4 byte character string variable in the application. However in unicode UTF-8 the string requires 5 bytes as **ñ** takes 2 bytes to represent. So if you had an application running codepage 819 connecting to a unicode (utf-8) database, issue

```
bind myapp GENERIC HV_EXPANSION_FACTOR 2
```

To force the database to allocate 2 times the defined application length for the server representation of the variable. So in this example the 4 byte variable becomes 8 bytes in the database server. This enables the string niño on expansion to 5 bytes in UTF-8, so it still fits in the memory available to the variable

STATICASDYNAMIC [YES|NO]

You can set the Db2 database manager to store all statements in the catalogs and marks them as incremental bind. To achieve this setting, the `STATICASDYNAMIC YES` string must be set for the `GENERIC` **`BIND`** command option.

At run time, when the package is first loaded, the database manager uses the current session environment (rather than the package) to set up the section entries and other entities (text is populated and the package cache is accessed).

Thereafter, the statements in the bound file behave the same as they would if you were using dynamic SQL. For example, sections are implicitly recompiled for database definition language invalidations, special register updates. The new syntax is defined as follows:

```
DB2 BIND filename GENERIC 'STATICASDYNAMIC [YES|NO]'
```

String literals with PREPARE statements

Embedded SQL applications use the PREPARE statement to dynamically prepare an SQL statement for execution. The PREPARE statement creates an executable SQL statement from a character string form of the statement, called a *statement string*.

C and C++ embedded SQL applications can prepare a statement from a host variable or a literal string (*statement string*) that is enclosed in a single quotation mark.

For example: `EXEC SQL PREPARE stmt_name FROM  'select empid from employee' ;`

Structure arrays

You can use structure arrays for FETCH INTO, INSERT, UPDATE, and DELETE statements that are non-dynamic, when you set the precompiler option `COMPATIBILITY_MODE` to `ORA`.

You can use structure arrays to store multiple column data in a structure form.

For a structure array that is declared for an INSERT, UPDATE, or DELETE statement, you must ensure that all array elements are initialized with a value. Otherwise, unexpected data can get introduced or removed from the table.

The total number of rows that are successfully processed is stored in the `sqlca.sqlerrd[3]` field. However, the `sqlca.sqlerrd[3]` field does not represent the number of rows that are committed successfully in the case of INSERT, UPDATE, or DELETE operations.

The total number of rows that are impacted by the INSERT, UPDATE, or DELETE operation is stored in the `sqlca.sqlerrd[2]` field.

In one FETCH INTO statement, the maximum number of records that can be retrieved is the cardinality of the array that is declared. If more rows are available after the first fetch, you can repeat the FETCH INTO statement to obtain the next set of rows.

A structure array can be used to store multiple column data in a structure form when a FETCH INTO statement is run. In the following example, a structure array is used for a FETCH INTO statement:

```
// Declare structure array with cardinality of 3.
EXEC SQL BEGIN DECLARE SECTION;
 struct MyStruct
 {
  int c1;
  char c2[11];
 } MyStructVar[3];
EXEC SQL DECLARE cur CURSOR FOR
 SELECT empno, lastname FROM employee;
EXEC SQL END DECLARE SECTION;
...
// MyStrutVar is a structure array for host variables
EXEC SQL FETCH cur INTO :MyStructVar;
```

You can use a structure array to store multiple rows for an INSERT statement. In the following example, a structure array is used for an INSERT statement:

```
// Declare structure array with cardinality of 3.
EXEC SQL BEGIN DECLARE SECTION;
   typedef struct _st_type {
       int   id;
       char  name[21];
   } st_type;

   st_type st[3];
EXEC SQL END DECLARE SECTION;

...
// Populating the array.
for( i=0; i<3; i++)
{
      memset( &st[i], 0x00, sizeof(st_type));
      if( i==0) { st[i].id = 100; strcpy(st[i].name, "hello1");}
      if( i==1) { st[i].id = 101; strcpy(st[i].name, "hello2");}
      if( i==2) { st[i].id = 102; strcpy(st[i].name, "hello3");}
}
// The structure elements must be in
// the same order as that of the table elements.
//
EXEC SQL INSERT INTO tbl values (:st);

// Check for SQLCODE.
printf("sqlca.sqlcode = %d\n", sqlca.sqlcode ); // 0
// The INSERT operation inserted 3 rows without encounting an error
printf("sqlca.sqlerrd[3] = %d\n", sqlca.sqlerrd[3] ); // 3

// The INSERT operation was successful and 3 rows has been stored in database.
printf("sqlca.sqlerrd[2] = %d\n", sqlca.sqlerrd[2] ); // 3

// The tb11 table now contains the following rows:
// C1          C2
// ----------- -----------
//         100 hello1
//         101 hello2
//         102 hello3
```

Restrictions with the structure array support

The use of the structure array in embedded SQL applications is subject to the following restrictions:

- Structure arrays are supported by C or C++ embedded SQL applications that connect to Db2 servers.
- Structure arrays must be declared in the `DECLARE SECTION` with exact size of the array elements (cardinality).

- Specific array element cannot be specified in a SQL statement.
- The INSERT, UPDATE, or DELETE operation with structure arrays is run as an atomic operation on the database server. If any array element causes an SQL_ERROR, current transaction is rolled back.
- Use of structure arrays are not supported by dynamically prepared INSERT, UPDATE, or DELETE statements.
- When structure array is specified, only one structure array can be declared in an embedded SQL application.
- You cannot create a structure array within another structure array (for example, nested structure arrays).
- Maximum size of array element (cardinality) is 32672.
- The following C and C++ data types are not supported for use with structure arrays:
  - BLOB
  - BLOB file reference
  - BLOB locator variable
  - CLOB
  - CLOB file reference
  - CLOB locator variable
  - User-defined data type
  - XML

UNSAFENULL PRECOMPILE option

You can suppress the unspecified indicator variable error by setting the `UNSAFENULL YES` option in the **`PRECOMPILE`** command, when you set the precompiler option `COMPATIBILITY_MODE` to `ORA`.

The unspecified indicator variable error is generated when the NULL value exists but the embedded SQL application failed to specify the NULL indicator.

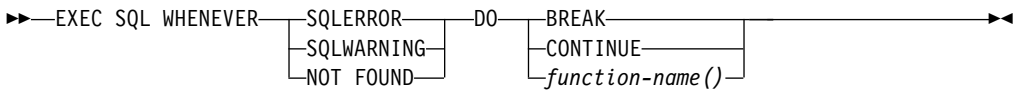
```
# Suppress SQL0305N error.
db2 prep test.sqc COMPATIBILITY_MODE ORA UNSAFENULL YES

# Default behavior if null value is retrieved, SQL0305N error
db2 prep test.sqc COMPATIBILITY_MODE ORA UNSAFENULL NO
db2 prep test.sqc COMPATIBILITY_MODE ORA
db2 prep test.sqc

# Below both PRECOMPILE option give error as COMPATIBILITY_MODE ORA is not set.
db2 prep test.sqc UNSAFENULL YES
db2 prep test.sqc UNSAFENULL NO
```

Remember: Even if you do not set the **`COMPATIBILITY_MODE`** parameter to `ORA` while precompiling, an application can check the `sqlca.sqlerrd[2]` structure to get the cumulative sum of the number of rows that were successfully populated until the last fetch in non-array host variables.

WHENEVER <condition> DO <action> statements

C and C++ embedded SQL applications can use the `WHENEVER` *`condition`* `DO` *`action`* statement to take a specified action when an exception condition occurs.

The WHENEVER statement specifies the action to be taken when a specified exception condition occurs. The following syntax diagram shows the `WHENEVER` *`condition`* `DO` *`action`* statement syntax:



The WHENEVER statement handles the following conditions:

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 WHENEVER statement conditions, the following DO *action* can take place:

DO Causes additional action in the form of a function call, break statement, or continue statement to take place.

BREAK
Specifies the C break statement. The C break statement exits the do, for, switch, or while statement block.

CONTINUE
Specifies the C continue statement. The C continue statement passes control to the next iteration of the do, for, switch, or while statement block.

function-name()
Specifies the C function that is to be called. The function must have a void return value and cannot accept any arguments. The function name must end with a set of parentheses "(" and ")". The name of the function is limited to 255 bytes.

The function name resolution takes place during the compilation of a C and C++ embedded SQL application. The Db2 precompiler does not resolve the function name.

The following C example uses the WHENEVER *condition* DO *action* statement:

```
void sqlError(void)
{
    switch (sqlca.sqlcode)
    {
        case -999: // some SQLCODE code
            printf ("Error related to -999 occurred\n");
            break;
        case -888: // some SQLCODE code
            printf ("Error related to -888 occurred\n");
            break;
        default:
            printf ("Unknown error occurred\n");
            break;
    }
    exit(-3);
}

int func1() // DO function-name

    ...
    EXEC SQL WHENEVER NOT FOUND DO sqlError();
    ...
    while(sqlca.sqlcode == SQL_RC_OK)
    {
        // some application logic
    }
    ...
}
```

```
int func2() // DO BREAK
{
    ...
    EXEC SQL WHENEVER SQLWARNING DO BREAK;
    ...
    while(...)
    {
        // some application logic
    }
    // Causes the next sequential instruction of the source program
    // to be executed. Basically voids effect of previous WHENEVER.
    EXEC SQL WHENEVER SQLWARNING CONTINUE;
    ...
}

int func3() // DO CONTINUE
{
    ...
    EXEC SQL WHENEVER NOT FOUND DO CONTINUE;
    ...
    while(...)
    {
        // some application logic
    }
    ...
}
```

Index

Special characters

Numerics

A

B

C

D

E

F

G

H

I

J

L

M

N

O

P

Q

R

S

T

U

V

W

X

Printed in USA