IBM® DB2 Universal Database™

Precompiler Customization

Version 8.2

IBM® DB2 Universal Database™

Precompiler Customization

Version 8.2

The latest version of this document (″Precompiler Services APIs″, prepapi.pdf) is available from the DB2 application development Web site (www.ibm.com/software/data/db2/udb/ad).

Contents

About this document

An SQL precompiler examines source code and processes SQL statements to generate modified source code for your application program. This document explains how you can create your own precompiler to support additional features or languages, using the Precompiler Services interface provided by DB2.

It is recommended that you read one or more of the following sections in the *Application Development Guide*: "Programming in C and C++", "Programming in FORTRAN", or "Programming in COBOL", to become familiar with the use of the existing DB2 precompilers. As well, you should study some of the sample programs provided with DB2, and compare the contents of the precompiler output with the original source. This will help you to understand what the precompiler must do in different circumstances.

Changed APIs (All versions)

Starting in Version 6, a new standard is being applied to some DB2 APIs. Implementation of the new API definitions is being carried out in a staged manner. Following is a brief overview of the changes:

- The new API names contain the prefix "db2", followed by a meaningful mixed case string (for example, db2CompileSql).
- Generic APIs have names that contain the prefix "db2g", followed by a string that matches the C API name.
- The first parameter into the function (*versionNumber*) represents the version, release, or PTF level to which the code is to be compiled. This version number is used to specify the level of the structure that is passed in as the second parameter.
- The second parameter into the function is a void pointer to the primary interface structure for the API. Each element in the structure is either an atomic type (for example, db2Uint32) or a pointer. Each parameter name adheres to the following naming conventions:

```
piCamelCase  - pointer to input data
poCamelCase  - pointer to output data
pioCamelCase - pointer to input or output data
iCamelCase   - integral input data
ioCamelCase  - integral input/output data
oCamelCase   - integral output data area
```

- The third parameter is a pointer to the SQLCA, and is mandatory.

Some Precompiler Services APIs have been replaced by new APIs conforming to this standard. The following table lists these APIs:

Table 1. Discontinued APIs

Descriptive Name	API (Version)	New API (Version)
Compile SQL Statement	sqlacmpl (V5)	db2CompileSql (V6)
Initialize Precompiler Services	sqlainit (V6)	db2Initialize[a] (V6 FixPak 4, V7)
Record Multiple Host Variable Addresses	sqlasetd (V6), sqlasetda (V7)	sqlasetdata (V8)
Record Host Variable Address		sqlastlva (V7 FixPak 1)

Note: [a] If a version number is specified when calling **db2Initialize**, the data types will adhere to the SQLTYPES and the SQLLEN for that version.

Other updated APIs are listed in the following table:

Table 2. Updated APIs

Descriptive Name	**API (Version)**
Parse Option String	sqlaoptions (V8, FixPak 2)

Chapter 1. Designing a precompiler

To create an effective precompiler, you need to understand the tasks of several database manager services. The following is an overview of the precompilation process. You can then examine each interrelated service.

The description of a processing model follows. Use this model to ensure that your precompiler is both efficient and complete.

Precompilation process

There are four participants in creating a successful application program:

- The application programmer
- The precompiler
- Precompiler Services
- Runtime Services

Each has a distinct list of responsibilities. During the development process, these duties complement each other. This division provides three distinct benefits for your application development:

- Programmers can make full use of the features of their preferred host language.
- Programmers do not need to learn a new interface to access the DB2 kernel.
- Additional languages can be added to the list of those supported by the database manager.

The following steps describe what happens to an SQL statement from its creation to its execution. You can see what is required at each step of the process.

1. The programmer creates an application and embeds an SQL statement in the source code.
2. The precompiler identifies SQL elements in the source code. It processes the statement to prepare it for Precompiler Services.
3. Precompiler Services compiles the processed SQL statement. It stores the processed statement in a bind file, and the compiled statement in a section of the package. Precompiler Services then defines the tasks required to successfully execute the SQL statement at run time.
4. The precompiler creates host language code, generally consisting of Runtime Services function calls to support these tasks, and inserts this code in the modified source file.
5. When the application executes, the Runtime Services function calls communicate with the database manager to process the SQL statements.

This procedure occurs for each SQL statement in the source being precompiled. During this process, the precompiler copies non-SQL code directly into the modified source file. The application programmer then compiles and links this modified file into the final form.

Although this sequence sounds simple, writing a precompiler requires a significant design effort and a solid understanding of text translation techniques. The following explains the responsibilities of each participant in greater detail.

Application programmer

The application programmer has two responsibilities:

- Construct correct SQL statements. For informaton about constructing SQL statements, see the *SQL Reference*.
- Choose appropriate precompilation options. The precompiler must collect a number of options, values, and names from the user. The precompiler passes this data as parameters to Precompiler Services. Each of the necessary parameters is discussed as Precompiler Services APIs and Runtime Services APIs are introduced.

Precompiler

The precompiler has the following responsibilities:

- Create the necessary data structures.
- Translate the application source file into a modified source file.
- Process host variable declarations.
- Process SQL statements.
- Construct Runtime Services function calls.

The precompiler is pivotal to simplifying application development. With a properly constructed precompiler, the application programmer does not need to create code for direct access to the database manager. The precompiler translates SQL requests to host language calls.

Precompiler Services

Precompiler Services has the following responsibilities:

- Validate and compile SQL statements.

  Precompiler Services calls the database manager for a full syntactic and semantic check of the SQL statement. The kernel compiles the statement and stores it in a section of the package.
- Identify how each host variable is used in the SQL statement.

  While compiling an SQL statement, Precompiler Services determines how all host variables are used (that is, for input or for output, or as indicator variables). It provides a usage code for each host variable.
- Identify tasks required to support the SQL statement.

  After processing SQL statements, Precompiler Services creates a list of tasks called a task array. The precompiler converts tasks in this array to function calls in the application program. The precompiler constructs host language calls to allocate an SQLDA, for example. It then inserts these calls into the modified source file. The application program completes these tasks to execute the statement.
- Create a bind file and package.

  Precompiler Services processes each executable SQL statement into a separate section of the package. The sections are collectively referred to as a package. Each precompiled source module has its own package.

  If the application programmer chooses to defer binding (the act of creating a package in the database), the processed SQL statements are stored in a bind file. The DB2 bind utility uses the bind file to create the package at a later time. This allows an application to use different databases without additional precompilation, since the bind file can be bound multiple times against different databases.

Runtime Services

Runtime Services APIs support communication between the application program and the database manager. Runtime Services has three responsibilities:

- Initialize and validate the SQL Communication Area (SQLCA) and the SQL Description Areas (SQLDA).
- Manipulate SQLDAs.
- Provide a functional interface to the database manager.

Runtime Services calls the database manager to process compiled SQL statements. It also modifies the input and output SQLDA, and passes dynamic SQL statements to the database manager.

Processing model

This section examines the responsibilities of the precompiler in detail. It also explores precompiler design and language considerations.

Precompiler design

There are several ways to structure a precompiler. Three approaches are discussed:

- The statement oriented model.

  You can design a precompiler to process source code one line at a time; each line is examined for host variable declarations and SQL statements.

  This design presents the precompilation process in a way that is easy to understand. It may work well with line-oriented languages, such as FORTRAN, but there are significant disadvantages. The design requires too much text rescanning to be efficient. Multi-line constructs and streaming languages, such as C, are difficult to handle.

- The compiler model.

  Another method is to create a precompiler that works like a compiler. Such a precompiler would use a parser and scanner to tokenize and process the input file. You could create production rules to identify and process host variable declarations and SQL statements.

  This model might require the precompiler to understand the complete host language syntax just for statement recognition. The precompiler would spend too much time processing non-SQL code. This is also inefficient.

- The hybrid model.

  The hybrid model is a compromise between the other two. Ideally, the precompiler should copy non-SQL code directly into the modified source file. One solution is a state-based scanner.

  The scanner could understand various text modes such as plain text, comments, and strings, and perform only minimal tokenization. It should only recognize SQL-related keywords in plain text areas (that is, not in the middle of comments or strings.)

  The precompiler also needs to process SQL statements as they are found. The entire statement must be processed at once. With a statement-oriented scanner, the entire statement can be identified. But additional semantic actions must occur before the statement is processed. An intelligent scanner or a simplified parser could discern the changes needed.

Depending on the host language, the precompiler may need to parse host variable declarations. Languages such as C have a complex syntax for variable declaration. You may have to process declarations and save contextual information as you go.

Although this model begins to address some important design considerations, it is not the only solution. Base your precompiler design on your understanding of the precompilation process and the requirements of your situation.

Language considerations

Your precompiler can be written in any language. Some languages offer more facilities to create an efficient precompiler. C is a good choice because it offers the following:

- String manipulation
- User-defined data structures
- Indirect addressing using pointers
- Dynamic allocation of storage.

The most important consideration is the ability of the language to call Precompiler Services. The precompiler should be able to pass structures that are arrays of pointers and other data objects. If the language does not support pointers, you may have difficulties. Precompiler Services also uses signed and unsigned data types. Consider these factors before you choose a language.

Precompiler responsibilities

The following outlines the tasks required of a successful precompiler. This is a generic list. You can add or subtract from it as necessary.

- Create necessary data structures.

  The precompiler uses the following structures:
  - Program identification string (PID)
  - Option array
  - SQLCA
  - Host variable name array
  - Token ID array
  - Task array
  - SQL flagger diagnostics structure.
- Translate source code into modified source code.

  The precompiler copies all non-SQL source code verbatim into the modified source file. It is important to maintain the integrity of the original application code.
- Process host variables.

  To process host variables, the precompiler:
  - Detects host variable declarations
  - Determines SQL data type, variable length, and other information
  - Assigns unique token IDs to each host variable
  - Maintains the host variables in a symbols table
  - Declares the variables to Precompiler Services.

  Your precompiler stores information about each host variable. You need this information to generate function calls in the modified source file.
- Process SQL statements.

To process SQL statements, the precompiler:

- Detects SQL statements
- Includes the statement as a comment in the modified source file
- Removes comments from the statement
- Replaces EXEC SQL keywords and the statement terminator with blanks
- Replaces non-blank white space with blanks
- Detects host variables in the SQL statement
- Places host variable IDs in the Token ID Array
- Replaces SQL statement host variable names with blanks
- Passes preprocessed SQL statements to Precompiler Services for compilation.

By processing SQL statements into this form, the precompiler turns language-specific statements into language-independent statements. Precompiler Services can then process each statement without knowing what host language is being used.

- Construct host language function calls.

  Once Precompiler Services compiles the SQL statement, it returns a sequence of functions and values to the precompiler in the task array. These values define the required calls. The precompiler inserts necessary Runtime Services function calls in the modified source file, based on the contents of the task array.

Processing model

Chapter 2. Writing a precompiler

The sequence of tasks a precompiler must perform is described in this section. The user interface is not discussed; you can design an interface that is appropriate to your needs.

Initialization

Before the precompiler reads the first byte of program input, it should perform the following initialization tasks:

- Initialize the precompiler.

  Allocate the SQLCA, set the precompiler break handler, process command line arguments, open files, and set up the option array. A database connection must be established before calling Precompiler Services initialization (**db2Initialize**).
- Initialize Precompiler Services.

  Call **db2Initialize** with initialization data and the option array.
- Process return data from **db2Initialize**.

  Check the SQLCA, generate program ID data in modified source file.

Figure 1 on page 8 shows the tasks performed by Precompiler Services.

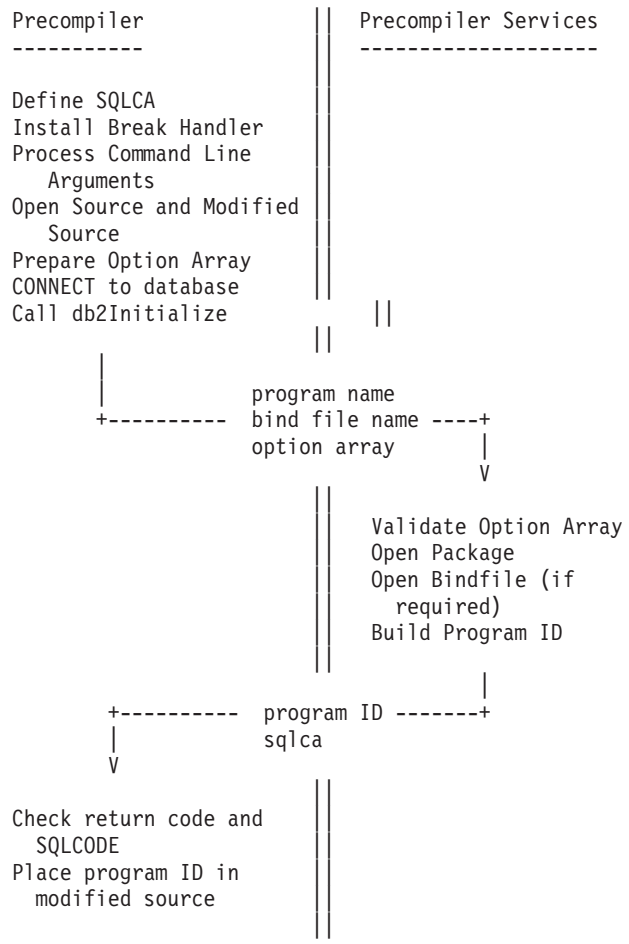

Figure 1. Initialization Tasks

Defining an SQLCA

All Precompiler Services APIs use the SQLCA to report completion codes and other diagnostic information to the calling program. Define and allocate an SQLCA before calling Precompiler Services.

For detailed information about the SQLCA structure, see the *SQL Reference*. You should not alter the SQLCA structure or member names, because programmers may wish to access the SQLCA directly, according to instructions found in other IBM manuals.

Handling interrupts

Precompiling can take a significant amount of time. The application programmer may decide to terminate the process by interrupting the precompiler. Precompiler Services detects user-initiated interrupts and returns the SQLCODE `SQLA_RC_CTRL_BREAK` (`-4994`) to the precompiler when these interrupts occur.

You may want to provide your own interrupt routine by installing a signal handler in the precompiler. Precompiler Services acquires control when an interrupt occurs. If the precompiler has an installed signal handler, the Precompiler Services handler invokes it before terminating.

Note: Install the signal handler before the first **db2Initialize** call occurs. If the precompiler installs its handler at any time after the first **db2Initialize** call, the results of an interrupt are unpredictable.

Although Precompiler Services maintains its own interrupt handler, the precompiler must still call Precompiler Services after an interrupt has occurred to properly terminate the precompilation session. Call **sqlafini** with the termination option set to discard the package or bind file.

After an interrupt, Precompiler Services rejects all calls except **sqlafini**. After **sqlafini** has completed, call **db2Initialize** to initiate a new precompilation session.

Processing command line arguments

A precompiler requires the following information:

source file name
The name of the file containing the source program being precompiled. This is not sent to **db2Initialize**, but is required by any precompiler.

modified source file name
The name of the file that will be created by the precompiler. This is not sent to **db2Initialize**, but is required by any precompiler.

bind file name
The name of the bind file to be produced, if any. This is generally based on the source file name, but can be any valid file name with an extension of `.bnd`.

database name
A short identifier specifying the alias of the database against which this program is to be precompiled. The precompiler connects to this database before calling **db2Initialize**.

package name
The name of the package to be created. This is generally based on the source file name, but can be any valid short identifier.

options
These specify the date and time format, isolation level, and record blocking behavior, among others. See "Preparing the option array."

You can allow users to specify these items on the command line or through some other method. In any case, the precompiler is responsible for validating these items.

Opening files

The precompiler opens the input source file and the output modified source file. The precompiler should not attempt to open the bind file. Precompiler Services performs this function.

Preparing the option array

The precompiler uses the option array to pass options to Precompiler Services. The array contains a header, followed by pairs of 4-byte integers.

The header consists of two 4-byte integers. The first integer gives the number of option pairs allocated. The second gives the actual number of options used.

If *M* is the required number of options, you need to allocate at least 8 * (*M*+1) bytes of storage for the option array.

The remaining pairs of integers specify options and option values. Each pair represents one option specification from the precompiler.

With DB2 UDB Version 5 and higher, Precompiler Services provides an API which parses an option string and produces the corresponding option structure to be passed to **db2Initialize**. This enables Precompiler Services clients to simply obtain a precompilation option string from the user with little regard to its contents, pass it to **sqlaoptions**, and then use the resulting option structure with **db2Initialize**. For details, see "sqlaoptions - Parse Option String" on page 51.

For a list of the options supported by the DB2 precompilers, see the description of **sqlaprep** in the *Administrative API Reference*. Precompiler Services supports all of the options listed there, except for the following, which are implemented by the various precompilers:

```
BINDFILE (*)              SQLCA
LONGERROR                 SQLERROR
MESSAGES                  TARGET
NOLINEMACRO               WCHARTYPE (**)
OPTLEVEL
PACKAGE (*)
```

Notes:

1. (*) BINDFILE and PACKAGE are implemented with different option values in Precompiler Services than in sqlaprep. See below.
2. (**) WCHARTYPE is used only for C applications, and affects the contents of the sqla_runtime_info structure. See "Runtime information structure" on page 49.

The following two options interact to control the creation of bind files and packages:

- SQLA_BIND_FILE (3) with:
  - Value `SQLA_CREATE_BIND_FILE` (1). A bind file is created with the name specified in the bind file argument to **db2Initialize**.
  - Value `SQLA_NO_BIND_FILE` (0). No bind file is created; the bind file argument to **db2Initialize** is ignored.
  - Value `SQLA_SQLERROR_CONTINUE` (2). Similar to `SQLA_CREATE_BIND_FILE`, but should be used if the user has also requested SQLERROR CONTINUE behavior (see the PRECOMPILE PROGRAM command in the *Command Reference* for details).
- SQLA_ACCESS_PLAN (2) with:
  - Value `SQLA_CREATE_PLAN` (1). A package is created with the name specified in the program name argument to **db2Initialize**.
  - Value `SQLA_NO_PLAN` (0). No package is created; the program name argument to **db2Initialize** is ignored.
  - Value `SQLA_SQLERROR_CONTINUE` (2). Similar to `SQLA_CREATE_PLAN`, but should be used if the user has also requested SQLERROR CONTINUE behavior (see the PRECOMPILE PROGRAM command in the *Command Reference* for details).
  - Value `SQLA_NO_PLAN_SYNTAX` (3). Similar to `SQLA_NO_PLAN`, but should be used if the user requires a check of statement syntax, but no package or bind file creation. See the SYNTAX option under PRECOMPILE PROGRAM in the *Command Reference*.

Note that the behavior of the SQL flagger (see the SQLA_FLAG_OPT option under **sqlaprep** in the *Administrative API Reference*) depends on the setting of the SQLA_BIND_FILE and SQLA_ACCESS_PLAN options. From the DB2 precompiler interface (that is, the PRECOMPILE PROGRAM command, or the **sqlaprep** API), specifying one of the variants of the SQLFLAG option alone will suppress bind file and package creation, and SQL statements will only be verified against the chosen syntax (DB2 for OS/390 or SQL92E), not against DB2 Universal Database (UDB). This corresponds to SQLA_ACCESS_PLAN with SQLA_NO_PLAN, and SQLA_BIND_FILE with SQLA_NO_BIND_FILE, in Precompiler Services. (If desired, a package and/or bind file can be created during SQL flagging, simply by specifying the SQLA_ACCESS_PLAN and SQLA_BIND_FILE options with the appropriate values.)

If, on the other hand, you require syntax checking against both DB2 for MVS or SQL92E, and DB2 UDB, but without package or bind file creation, this can be achieved by combining the SQLA_FLAG_OPT option and the SQLA_NO_PLAN_SYNTAX value of the SQLA_ACCESS_PLAN option.

Initializing Precompiler Services using db2Initialize

The precompiler initializes Precompiler Services by calling **db2Initialize**. See "db2Initialize - Initialize Precompiler Services" on page 62 for a description of this API.

Testing the return code from db2Initialize

Precompiler Services passes any errors or warnings returned from **db2Initialize** through the SQLCA; however, Precompiler Services first attempts to validate the SQLCA itself. The return code from **db2Initialize** shows whether the SQLCA address was valid, and whether the structure is long enough to contain an SQLCA. The return code may be set to the following values:

SQLA_CHECK_SQLCA (0)
Check the SQLCA.SQLCODE element for the completion code.

SQLA_SQLCA_BAD (-1)
The address of the SQLCA passed to the function was not valid; the command was not processed.

Processing the program ID

The program ID is a string of alphanumeric data used to identify the program being precompiled, the user ID performing the precompilation, the date and time when precompilation occurred, and so on. On initialization, Precompiler Services creates a program ID and returns it to the precompiler. The precompiler generates the appropriate host language code to declare a character array in the modified source file and initialize it with the program ID. At run time, the variable that contains this data is used as an input parameter to the **sqlastrt** function.

Note: For maximum portability, the program ID should be restricted to uppercase A-Z, 0-9, and the underscore (_).

Errors that require reinitialization

If **db2Initialize** is successful, Precompiler Services returns 0 to SQLCODE in the SQLCA. If it is not successful, an error code is returned.

If a fatal error occurs while executing any Precompiler Services API, terminate Precompiler Services with a call to **sqlafini**. Reinitialize with an **db2Initialize** call before continuing. To see which errors are considered fatal, refer to Chapter 9, "Error messages and codes," on page 81.

Source processing

The following describes tasks related to processing the input source file and generating the modified source file.

Copying non-SQL code

Your precompiler should copy all non-SQL code directly into the modified source file. It is important to maintain the integrity of the application program.

While copying the non-SQL code, the precompiler searches for the keywords `EXEC SQL`. The precompiler only recognizes these keywords if they appear on the same line, separated by one blank. You may want to relax these restrictions.

These keywords are part of the ANSI standard; however, you may want to use some other way to identify SQL statements.

Note that if the keyword pair `EXEC SQL` appears in a comment or a string, it should be ignored. Your precompiler should not recognize comments or string characters as valid keywords. Ensure that your logical scanning rules properly enter and exit all comment and string variations.

Precompiler tasks for host variables

When Precompiler Services compiles a BEGIN DECLARE SECTION statement, it tells the precompiler to begin processing host variables. The precompiler should copy syntactically valid host variable declarations directly into the modified source file. SQL-based "pseudo declarations", such as for large object types, should be translated into proper host language syntax before they are written to the output file. The precompiler also needs to record the host variable in a symbols table and register it with Precompiler Services.

When the precompiler detects a host variable, it does the following:

- Determines if the host variable is SQL compatible
- Determines the SQL type
- Adds information to the host variable symbols table
- Calls the Precompiler Services function **sqlaalhv**
- Checks for successful completion.

The precompiler continues processing host variables until an END DECLARE SECTION statement is detected.

Acceptable host variables

Only certain host variable data types are compatible with SQL columns. To be recognized as an SQL host variable, the variable must conform with the SQL variable declaration syntax of each host language. See the *Application Development Guide* for examples of language-specific variable formats that are compatible with SQL.

The precompiler determines if the declared host variable is acceptable. If not, the precompiler can return an error, or ignore the declaration.

It is up to the implementer to decide what host variable declaration syntax will be accepted. Typically, the precompiler accepts declarations that are valid in the target host language (for example, C declarations in a C precompiler). However, the precompiler may accept non-host language declarations and map them to proper host language in the modified source file. This may simplify parsing, or may provide a way for the user to express a semantic difference between host variables whose declarations would otherwise be syntactically identical in the host language.

For example, C does not provide a way to indicate at declaration time that a character array does not contain a NULL terminator, and that it should be treated as a fixed-length character string. A C precompiler could recognize some extra syntax and assign it to the SQL type 452 (fixed-length string) instead of type 460 (C NULL-terminated string). This hypothetical precompiler might successfully process the following:

```
exec sql begin declare section;
         FIXED_CHAR a[10]; /* type 452  */
         char       b[10]; /* type 460 */
exec sql end declare section;
```

and then give the modified source:

```
/* exec sql begin declare section; */
          /*FIXED_CHAR*/ char a[10]; /* type 452 */
          char       b[10]; /* type 460 */
/* exec sql end declare section; */
```

The "pseudo-declaration" technique is used extensively for large object declarations. For details, see "Large objects" on page 14.

Following are some examples of host variable declarations. In these examples, all three declarations are legal in C. The variables *number* and *mystruct1* would be valid with a basic C precompiler, since they are each recognized as an atomic SQL type. The variable *mystruct2* would only be valid to a more sophisticated precompiler implementing structure support. This feature allows the precompiler to recognize declarations of structures which are not themselves atomic SQL types, but which are composed of them. For more detailed information about structure support, see "Support for structure host variables" on page 41.

```
EXEC SQL BEGIN DECLARE SECTION;

short number;      /* "number" is recognized as a  */
                   /*  SMALLINT host variable by   */
                   /*  the precompiler.            */

struct {           /* "mystruct1" is recognized as */
  short number;    /*  a VARCHAR host variable by  */
  char mydata[30]; /*  the precompiler.            */
} mystruct1;

struct {           /* "mystruct2" is NOT recognized  */
  char mydata[30]; /*  as a host variable by the     */
  short number;    /*  precompiler, unless structure */
} mystruct2;       /*  support is implemented.       */
                   /*  This will cause a precompilation error. */

EXEC SQL END DECLARE SECTION;
```

Determining SQL type

Each acceptable host variable has a corresponding SQL data type. The writer of a precompiler must determine what host language declarations are equivalent to each SQL type. The precompiler then recognizes those declarations, and maps

them to the appropriate SQL types. The *SQL Reference* lists all SQL types, and the *Application Development Guide* shows how they are declared in the languages currently supported by the precompilers provided by DB2.

Large objects

DB2 supports large object (LOB) SQL types, which are essentially large capacity character and byte strings. These are typically declared with the "pseudo-declaration" mechanism mentioned earlier. The syntax `SQL TYPE IS` *type-name* is used in these declarations, where *type-name* is one of:

- BLOB(size)
- CLOB(size)
- DBCLOB(size)
- BLOB_LOCATOR
- CLOB_LOCATOR
- DBCLOB_LOCATOR
- BLOB_FILE
- CLOB_FILE
- DBCLOB_FILE.

The variable *size* is the size in bytes for BLOBs and CLOBs, and in double-byte characters for DBCLOBs. For detailed information about the nature and use of LOB SQL types, see the *SQL Reference*.

These types are of particular interest to the writer of a precompiler, because the pseudo declarations in the input source file must be replaced by equivalent declarations in the host language. For information about how these declarations map to their equivalents in C, COBOL, and FORTRAN, see the *Application Development Guide*. Note that the LOB and LOB file declarations map to structure declarations in the host language. The names of the structure members are generated by the precompiler, but must be predictable to the user, since the application must be able to reference them. For example, in C:

```
exec sql begin declare section;
   static sql type is clob(100) foo;
exec sql end declare section;
```

maps to:

```
/* exec sql begin declare section; */
   static /* sql type is clob(100) */
   struct foo_t {
       sqluint32 length;
       char      data[100];
   } foo;
/* exec sql end declare section; */
```

In this example, *foo* becomes a structure name, with members *length* and *data*. The precompiler could have named the members however it liked, but the application programmer must know what name to use, so that `foo.length`, for example, can be referred to. Note that in languages like COBOL, where structure members are part of the global name space, the structure members should have names that make them uniquely identifiable; for example, `FOO-LENGTH` and `FOO-DATA`. The LOB file structure has more fields, but the principle is the same. The LOB locator declarations become simple 4-byte integer declarations in the host language.

Structured types

DB2 supports structured types. They can be declared using the `SQL TYPE IS` *type-name* AS PREDEINFED_TYPE *var-name*. Refer to the *SQL Reference* for details on structured types. The predefined PREDEFINED_TYPE can be any of the following:

- SMALLINT
- INTEGER
- BIGINT
- REAL
- DECIMAL
- DOUBLE
- CHAR
- GRAPHIC
- VARCHAR(size)
- BLOB(size)
- CLOB(size)
- DBCLOB(size)
- BLOB_LOCATOR
- CLOB_LOCATOR
- DBCLOB_LOCATOR
- BLOB_FILE
- CLOB_FILE
- DBCLOB_FILE

Just as in a large objects declaration, the precompiler can replace these declarations with equivalent declarations in the host language. For example, in C:

```
EXEC SQL BEGIN DECLARE SECTION;
          SQL TYPE IS Person_1 AS VARCHAR(30) person1;
          SQL TYPE IS Person_2 AS SMALLINT person2;
EXEC SQL END DECLARE SECTION;
```

would map to:

```
struct {
  short length;
  char data[30];
} person1;

short person2;
```

The structured type information must be added to the precompiler host variable table and the structured type name can be passed to Precompiler Services using the **sqlaalhv** call.

Recording host variables: The precompiler assigns a unique 4-byte token ID to each declared host variable. The token ID may actually be a pointer or an array index, or anything else. For Precompiler Services, the only consideration is that the token ID be unique for each host variable. Precompiler Services refers to variables only by token ID; however, host variable names passed to Precompiler Services must also be unique. Variable names may be up to thirty characters in length.

Precompiler Services uses host variable information to set up SQLVAR elements in SQLDA structures; because of this, the precompiler must be able to provide the following information:

- Token ID
- Variable name length
- Variable name
- SQL type
- Variable data length
- Location code.

You can use various techniques to create a unique token ID and determine variable name length. The precompiler should maintain a host variable symbols table to provide a cross-reference between token IDs and variable names.

Storing the variable name itself causes some additional problems. The recorded name must not contain any operators. If the variable is declared as `short *number` (as in C), the name should be recorded as `number`. Store the operator separately with the token ID as the key. The operators used with host variables must be retrievable by the precompiler when the calls to Runtime Services are generated by the precompiler.

Note: The name can contain characters from the database manager's extended character set. A host language can allow characters that are not part of that extended set. Remove those characters, and replace them with valid characters before sending the name to Precompiler Services.

The precompiler should check the SQLCA before retrieving messages. Replace the original characters before displaying error or warning messages. Otherwise, variable names with replacement characters may appear in your messages. The SQLCODEs which currently contain host variable names are: SQL0104N, SQL0303N, SQL0307N, SQL0312N, SQL0324N, and SQL4942N. For more information, see Chapter 9, "Error messages and codes," on page 81.

Determine SQL data type and variable length from the host variable declaration. For non-graphic data types, variable length is the actual length of the host variable in bytes. For example, the length of a short integer is 2. For graphic data types, such as `PIC G(xx)` in COBOL, the length is the number of double-byte characters, not the number of bytes.

For the DECIMAL data type, length is determined by placing the declared precision of the variable in the lower-address byte of the length field, and the scale in the higher-address byte. In COBOL, a declaration like `PIC S9(7)V99 COMP-3` has precision 9 and scale 2. Assembled in a temporary short integer for calculating the length, this becomes:

```
Offset  0   +----------------+
            | precision = 9  |
        1   +----------------+
            | scale = 2      |
            +----------------+
```

If the above is again viewed as a short integer, it becomes a value of 9*256+2=2306 on "Big Endian" (UNIX based) operating systems, and 2*256+9=521 on "Little Endian" (Windows) operating systems.

The location code tells Precompiler Services where the host variable was declared. Host variables found in an SQL statement, not previously declared in an SQL declare section, are assumed to be SQLDA structures for dynamic SQL statements. Ensure that SQLDAs have their location marked correctly.

Reporting host variables through sqlaalhv: Once the above information has been determined, the precompiler calls Precompiler Services through the **sqlaalhv** API. This API provides Precompiler Services with the information it uses to compile SQL statements with host variables. For a description of this API, its arguments, and valid completion codes, see "sqlaalhv - Add Host Variable" on page 57.

The return code from **sqlaalhv** reports the validity of the SQLCA structure. If it reports that the SQLCA is valid, check SQLCODE for the completion status of the **sqlaalhv** API.

Processing host variables outside the declare section: The precompiler usually calls **sqlaalhv** when processing a host variable declaration. There is a special circumstance in which the precompiler calls sqlaalhv while processing an SQL statement.

When the precompiler finds an undeclared host variable identifier in an SQL statement, the precompiler assumes that the identifier represents an SQLDA structure. It should call **sqlaalhv** with location set to `SQLA_SQL_STMT` (1), and the *sqltype* and *sql_length* pointers set to NULL. If a second undeclared host variable is found in the statement, the precompiler must consider this an undeclared host variable error, because only one SQLDA name can occur in a statement.

Processing SQL statements

A statement consists of all tokens found between the `EXEC SQL` keywords and the SQL statement terminator. This is exclusive of the begin and end tokens themselves, and any line continuation tokens, comments, or other host language artifacts.

Identifying SQL statements

The precompiler identifies statements by recognizing the `EXEC SQL` keyword pair at the beginning of an SQL statement. It resumes scanning text after the statement terminator has been found, and the statement has been completely processed.

SQL statement terminators may be language and product specific. As a guideline, the ANSI standard proposes the following terminators:

C/C++ semicolon (;)

Pascal semicolon (;)

COBOL
END-EXEC keyword

FORTRAN
End of a line with no continuation.

Using these rules, SQL statements embedded in C programs have the following syntax:

```
EXEC SQL <statement>;
```

IN COBOL, SQL statements have the following syntax:

```
EXEC SQL <statement> END-EXEC
```

Statement termination processing is affected somewhat by compound SQL. For more information, see “Compound SQL” on page 37.

SQL statements can span lines using continuation tokens available in the host language. In the C precompiler, statements span lines until they are terminated, according to usual C convention. Host language rules regarding token continuation should apply as well.

Copying SQL statements to modified source

The precompiler can copy SQL statements into the modified source as comments, in a format appropriate to the host language. Copying the statements enhances the readability of the modified source. It helps the application programmer determine which changes were made to the source file.

For example, the C precompiler copies statements such as:

```
/*
<SQL statement>
*/
```

Preprocessing SQL statements

The precompiler preprocesses each SQL statement before sending it to Precompiler Services for compilation. The precompiler removes non-SQL constructs such as comments, host variable operators and names, and non-blank white space characters. The precompiler replaces this material with blanks, on a character-for-character basis. In this way, the position of statement tokens does not change, and the precompiler may be able to construct more meaningful diagnostic messages.

Recognizing quoted strings: During the scan of an SQL statement string, the precompiler must recognize quoted strings. Quoted strings in SQL are delimited by apostrophes or quotation marks. If a string delimiter appears within a string, it must be doubled; that is, '' or "". SQL syntax does not allow nesting of strings or mixing of string delimiters, ' or ". During statement preprocessing, ignore all characters appearing within quoted strings, and do not alter material there.

Replacing statement comments: There are two methods for embedding comments within SQL statements. The first method uses comment delimiters native to the host language. For example, C uses:

```
/* */
```

as comment delimiters.

Note: Some types of host language comment introducers, such as // in C/C++ and ! in FORTRAN (which can both start in mid-line), can conflict with SQL syntax. You should be very cautious about supporting such host language comment introducers within SQL statements. Supporting them elsewhere within the input source file is less risky.

Besides language-specific comment delimiters, ANSI SQL also provides comment delimiters for use within SQL statements. The format for ANSI SQL style is a double dash (--). It is followed by a string of 0 or more characters, and terminated by an end-of-line character. To assist portability, precompilers should support ANSI SQL style comments appearing within an SQL statement. This is the preferred method of including comments in SQL statements.

Blank out all comments (including their delimiters) appearing within the SQL statement text before issuing a compile request for that statement. Precompiler Services treats comment delimiters as invalid syntax.

Removing host variable identifiers and operators: Legal syntax for host variables varies across different languages. For example, the form :A-B has one meaning in C, and a different meaning in COBOL. In C, the form suggests a host variable A minus column B (the hyphen is an operator). In COBOL, the form might identify a single host variable A-B (the hyphen is part of the identifier). Note that in the COBOL case, the precompiler would translate the embedded hyphen to another character (for example, an underscore) before calling **sqlaalhv**. In this way, Precompiler Services does not need to know the host language conventions regarding hyphens.

When the precompiler finds a host variable in an SQL statement, it blanks out the host variable in the statement string, looks up the host variable's token ID in the host variable symbols table, and places the token ID in the token ID array. The colon that precedes a host variable is left in the SQL statement. This shows Precompiler Services that a host variable occupies that space. There is a one-to-one correspondence between colons in the SQL statement and entries in the token array ID prepared by the precompiler.

For example, assume that the precompiler is processing the following SQL statement in C. The slash (/) represents a new line character, and the period (.) represents a blank:

```
EXEC SQL
   SELECT A,B,C
   INTO :VAR_A:IND_A,:VAR_B:IND_B,:VAR_C
   FROM T
   WHERE A > :HV1 AND B < :HV2 AND C = :HV3;
```

Removing the statement delimiters gives:

```
"...SELECT.A,B,C/
 ...INTO.:VAR_A:IND_A,:VAR_B:IND_B,:VAR_C/
 ...FROM.T/
 ...WHERE.A.>.:HV1.AND.B.<.:HV2.AND.C.=.:HV3"
```

Removing host variables gives:

```
"...SELECT.A,B,C/
 ...INTO.:.....:.....,:.....:.....,:...../
 ...FROM.T/
 ...WHERE.A.>.:....AND.B.<.:....AND.C.=.:..."
```

Keep a list of all operators associated with each host variable found in an SQL statement. The precompiler restores the operators before using the host variables in calls to Runtime Services. For example, the C precompiler allows the * operator for host variables declared as pointers. If an * operator prefixes the host variable in the SQL statement, it must also be used in the generated run time calls.

If a DECLARE *cname* CURSOR FOR *select stmt* statement contains host variables with operators, those operators must be stored in a table, with the section number used as the key to the operator list. Precompiler Services returns the section number from a **db2CompileSql** call for a DECLARE statement.

When an OPEN CURSOR statement is processed for that cursor, **db2CompileSql** returns the token IDs of the host variables in the DECLARE statement. It also returns a section number that matches the declared cursor. The precompiler checks

whether the section number matches a section number in the operator list table. If so, the precompiler should restore the operators to the host variable names. The precompiler then uses the restored host variable names and their operators in **sqlastlv** or **sqlasetdata** calls.

Removing white space and leaving a spare byte: Replace each non-blank white space character with a blank (X'20') before making a compile request for that statement. White space characters include:

- Carriage return (X'0D')
- Line feed (X'0A')
- Tab (X'09').

Finally, the precompiler must include at least one extra byte beyond the last character in the SQL statement. Precompiler Services uses this byte when compiling the statement.

Following is the completely preprocessed example statement. The white space characters are replaced by blanks, and the extra byte (?) appears at the end of the statement:

```
"...SELECT.A,B,C....INTO...:.....:.....,.
 :.....:.....,.:.........FROM...T....
 WHERE..A.>.:....AND.B.<.:....AND.C.=.
 :....?"
```

Preparing the token array

The token array is used to pass host variable and literal information between the precompiler and Precompiler Services.

The precompiler fills in the token array with the token IDs of all the host variables in the SQL statement, in the order in which they occur. During the **db2CompileSql** call, Precompiler Services updates this array with usage information to indicate to the precompiler how each host variable in the statement is used.

Precompiler Services may also add entries to the array, representing significant string literals in the SQL statement. An example of such a literal would be the database name in a CONNECT TO statement. The precompiler does not ″understand″ SQL, so it doesn't ″know″ the nature of the statement, much less where the database name is located. As a result, Precompiler Services inserts an entry in the array at the appropriate location, pushing down any other entries which might come behind it. On return from **db2CompileSql**, the precompiler interrogates the token ID array to find the value of literals it needs to generate the Runtime Services calls, in a similar way to how it generates calls using host variables.

The token array is an array of logically paired 4-byte integers. The first integer is the number of pairs available for tokens in the array. The precompiler initializes this array size before any compile calls to Precompiler Services. It is necessary that the token array be large enough to allow for the literals that may be inserted by Precompiler Services. If Precompiler Services finds that the token array is too small, it sets the second field of the header equal to the required number of entries. It then returns error `SQLA_RC_TOKEN_ARRAY_LIMIT` (`-4920`) in the SQLCODE field of the SQLCA. The precompiler can then reallocate the token array to the correct size and call Precompiler Services again, or return an error.

The second integer is the number of pairs that actually contain host variable and/or literal information. The precompiler provides this value when it calls Precompiler Services. However, Precompiler Services can modify the second integer if it requires more token IDs, or if it needs to insert literals into the token array. An example of Precompiler Services requiring more token IDs is an OPEN statement in which host variables appear in the SELECT clause of the corresponding DECLARE CURSOR statement.

All other pairs contain information about each host variable or literal found in the SQL statement. For host variables, the first element of each pair is the token ID. For literals, Precompiler Services divides the first element (a 4-byte integer) into two adjacent 2-byte integers, collectively referred to as a ″return token″ structure. The first 2-byte integer gives the offset in bytes from the beginning of the SQL statement to the beginning of the literal. The second 2-byte integer gives the length of the literal in bytes.

Precompiler Services fills in the second element of the pair with a usage code. These usage codes are as follows:

SQLA_INPUT_HVAR (0)
Input host variable

SQLA_INPUT_WITH_IND (1)
Input host variable with indicator variable

SQLA_OUTPUT_HVAR (2)
Output host variable

SQLA_OUTPUT_WITH_IND (3)
Output host variable with indicator variable

SQLA_INDICATOR (4)
Indicator variable

SQLA_INVALID_USE (5)
Host variable does not match use

SQLA_USER_SQLDA (6)
User-defined SQLDA name

SQLA_INVALID_ID (7)
Host variable token ID is not valid

SQLA_LITERAL (8)
Literal string.

The following graphic represents the token array. Each cell contains a 4-byte integer.

- A = Number of token pairs allocated in the array
- N = Number of token pairs needed for this statement
- T = Token ID or instance of return token structure
- U = Usage code.

Your precompiler should determine some practical initial allocation size. If the precompiler encounters an SQL statement that contains more host variables or literals, it can allocate a larger size array and call **db2CompileSql** again, if necessary.

Example 1

Assume that the precompiler has processed the following SQL statement, and that a single new line character follows the last visible character on each line:

```
EXEC SQL
   SELECT A,B,C
   INTO   :VAR_A:IND_A, :VAR_B:IND_B, :VAR_C
   FROM   T
   WHERE  A > :HV1 AND B < :HV2 AND C = :HV3;
```

Assume that your precompiler assigned the following token IDs to the host variables:

Name	Token ID
HV1	6
HV2	7
HV3	8
VAR_A	2
VAR_B	4
VAR_C	5
IND_A	10
IND_B	11

The following token array has a capacity of 25 host variables or literals. The precompiler constructs this array before the compilation request:

Token ID	Usage
25 (A)	8 (N)
2	-
10	-
4	-
11	-
5	-
6	-
7	-
8	-

After compilation, Precompiler Services returns this array:

Token ID	Usage
25 (A)	8 (N)
2	SQLA_OUTPUT_WITH_INDICATOR
10	SQLA_INDICATOR
4	SQLA_OUTPUT_WITH_INDICATOR
11	SQLA_INDICATOR
5	SQLA_OUTPUT_HVAR
6	SQLA_INPUT_HVAR

Token ID	Usage
7	SQLA_INPUT_HVAR
8	SQLA_INPUT_HVAR

Example 2

Assume the precompiler has processed the following SQL statement:

```
EXEC SQL CONNECT TO :dbname USER 'FRED' USING :pwd;
```

Assume that your precompiler assigned the following token IDs to the host variables:

Name	Token ID
dbname	2
pwd	3

The following token array has a capacity of 50 host variables or literals. The precompiler constructs this array before the compilation request:

Token ID/Literal	Usage
50 (A)	2 (N)
2	-
3	-

After compiling, Precompiler Services returns this array:

Token ID/Literal	Usage
50 (A)	3 (N)
2	SQLA_INPUT_HVAR
Literal 1	SQLA_LITERAL
3	SQLA_INPUT_HVAR

Literal 1 is an instance of the return token structure, where:

```
offset = offset of F in FRED in the SQL statement (24)
length = length of FRED (4)
```

Compiling an SQL statement through db2CompileSql

The **db2CompileSql** API compiles an SQL statement. During this call, Precompiler Services:

- Parses the statement
- Assigns a section number, if needed
- Stores the statement in a bind file, if one is being created
- Completes the task array
- Completes the token ID array
- Provides other output parameters.

For a description of this API, its arguments, and valid completion codes, see "db2CompileSql - Compile SQL Statement" on page 59.

As with **sqlaalhv**, check the return code to determine the validity of your SQLCA structure. If the SQLCA is valid, check SQLCODE in the SQLCA for completion status of the **db2CompileSql** function call.

Generating code

If the Precompiler Services compilation request was executed successfully, the precompiler interrogates the task array. The task array defines further actions to be taken by the precompiler.

The task array is an array of logically paired 4-byte integers, very similar to the token ID array. The pairs specify which run time functions or data structures should be generated in the modified source file.

The first logical pair of 4-byte integers is the header. The first integer is the number of pairs available in the task array for the task codes. The precompiler initializes this value before calling **db2CompileSql**.

The second integer is the number of pairs that contain valid data on return from **db2CompileSql**. Precompiler Services provides this value.

If Precompiler Services finds that the task array is too small, it sets the second header integer equal to the required number. Precompiler Services then returns error `SQLA_RC_TASK_ARRAY_LIMIT (-4919)` through SQLCODE in the SQLCA. Your precompiler can then reallocate the task array to the correct size, and call **db2CompileSql** again.

The remaining pairs contain the task array function flag and the function value. The function flag F represents a function the precompiler must perform. The function value V is associated with the previous flag. It contains data necessary to perform the function.

This graphic represents the task array. Each cell contains a 4-byte integer.

- A = Number of pairs allocated
- U = Number of pairs used
- F = Function flag
- V = Function value.

```
+---+---+---+---+---+---+---+---+--
| A | U | F | V | F | V | F | V |   ...
+---+---+---+---+---+---+---+---+--
```

Table 3 lists all possible function flags and function values. It also shows the tasks required of your precompiler.

Table 3. Function Flags and Function Values

Function	Value	Precompiler Action
SQLA_START (0)	(not used)	Generate host language call to **sqlastrt**.
SQLA_DECLARE (1)	SQLA_BEGIN (0)	Begin processing host variables.
SQLA_DECLARE (1)	SQLA_END (1)	Terminate processing host variables.
SQLA_INCLUDE (2)	SQLA_SQLCA (10)	Generate code for a standard SQLCA template.

Table 3. Function Flags and Function Values (continued)

Function	Value	Precompiler Action
SQLA_INCLUDE (2)	SQLA_SQLDA (11)	Generate code for a standard SQLDA template.
SQLA_INC_TEXTFILE (14)	Instance ofsqla_return_token	Suspend reading current file; start reading include file.
SQLA_ALLOC_INPUT (3)	Number of SQLVAR elements	Generate host language call for an input **sqlaaloc**, and then generate **sqlastlv** calls for all host variables with an ″input″ usage in the token ID array.
SQLA_ALLOC_OUTPUT (4)	Number of SQLVAR elements	Generate host language call for an output **sqlaaloc**, and then generate **sqlastlv** calls for all host variables with an ″output″ usage in the token ID array.
SQLA_USDA_INPUT (6)	Token ID of the user-specified input SQLDA	Generate host language call to **sqlausda**.
SQLA_USDA_OUTPUT (7)	Token ID of the user-specified output SQLDA	Generate host language call to **sqlausda**.
SQLA_SETS (5)	Token ID of the host variable containing the dynamic SQL statement	Generate host language call to **sqlastls**.
SQLA_CALL (8)	SQLA_CONNECT (29), SQLA_DUOW (40)	Generate call to **sqlacall** with call_type = the task value and section_number = poSqlStmtType returned from **db2CompileSql**.
SQLA_CALL (8)	Various, not CONNECT or DUOW	Generate call to **sqlacall** with call_type = the task value.
SQLA_DEALLOC (9)	(not used)	Generate call to **sqladloc**.
SQLA_STOP (10)	(not used)	Generate call to **sqlastop**.
SQLA_SQLERROR (11)	Length of the host-label name in poBuffer1 of **db2CompileSql**	Generate code for WHENEVER SQLERROR.
SQLA_SQLWARNING (12)	Length of the host-label name in poBuffer2 of **db2CompileSql**	Generate code for WHENEVER SQLWARNING.
SQLA_NOT_FOUND (13)	Length of the host-label name in poBuffer3 of **db2CompileSql**	Generate code for WHENEVER NOT FOUND.
SQLA_BEGIN_ COMPOUND (15)	Non-zero flag means that a STOP AFTER FIRST clause was present	Begin processing Compound SQL Block (see “Compound SQL” on page 37).
SQLA_CMPD (16)	SQLA_COMMIT (21)	Generate call to **sqlacmpd** with call_type = SQLA_COMMIT.
SQLA_CMPD (16)	SQLA_EXECUTE (24)	Generate call to **sqlacmpd** with call_type = SQLA_EXECUTE.

Table 3. Function Flags and Function Values (continued)

Function	Value	Precompiler Action
SQLA_CMPD_TEST (17)	Token ID of the controlling host variable in a STOP AFTER FIRST clause in compound SQL	Generate code to test the STOP AFTER FIRST variable.
SQLA_CMPD_MARK (18)	(not used)	Generate target label for STOP AFTER FIRST processing in compound SQL.
SQLA_NEXT_ SUBSTATEMENT (19)	(not used)	Send the next available substatement to Precompiler Services (see "The CREATE TRIGGER statement" on page 39).
SQLA_SQLCODE_COPY (20)	(not used)	Update the stand-alone SQLCODE and SQLSTATE variables (see "Support for stand-alone SQLCODE/SQLSTATE" on page 42).

Null task array

If the second integer in the task array (the number of pairs used) is set to 0, no further processing of the task array is required.

The precompiler may need to act on other output parameters of the **db2CompileSql** API. One instance of this is after compiling a DECLARE CURSOR statement. If the cursor was declared for a SELECT statement — indicated by the *poSqlStmtType* output parameter from **db2CompileSql** being set to `SQLA_TYPE_DECLARE_SELECT (0)` — that contained host variables with operators, the precompiler should save the list of operators used in the SELECT statement, keyed on the section number also returned from **db2CompileSql**.

Inserting an SQLCA data structure into modified source

When Precompiler Services detects an SQL INCLUDE SQLCA statement, it returns the function pair SQLA_INCLUDE and SQLA_SQLCA in the task array. The precompiler then embeds an SQLCA data structure declaration directly into the modified source program. If appropriate for the host language, the precompiler also declares an instance of that structure in the modified source. Both the SQLCA definition and the declaration are host language-specific.

Inserting an SQLDA data structure into modified source

When Precompiler Services detects an SQL INCLUDE SQLDA statement, it returns the function pair SQLA_INCLUDE and SQLA_SQLDA in the task array. The precompiler then embeds an SQLDA data structure declaration directly into the modified source program. The precompiler does not declare an instance of the SQLDA, as it does with the SQLCA. The SQLDA is inserted into the modified source only as a template. The application program assumes full responsibility for allocating and manipulating SQLDAs.

The INCLUDE SQLDA statement usually cannot be embedded in FORTRAN, which does not generally support templates. Other languages may share this problem.

Processing an embedded source file

When Precompiler Services detects an SQL INCLUDE text file statement, it returns the function flag SQLA_INC_TEXTFILE in the task array. The corresponding function value is a packed pair of 2-byte integers providing the offset and length of the file name within the SQL statement. The precompiler then suspends processing of the current source file and begins processing the included source file. Nesting of include files should be permitted to a reasonable depth, and care should be taken to detect cyclic includes. The messages SQL0062W and SQL0063W can be used to announce the start and end of include file processing to the user.

Inserting runtime function calls

SQL statements other than DECLARE, INCLUDE, WHENEVER, BEGIN DECLARE SECTION, and END DECLARE SECTION must be executed by calling Runtime Services functions. There are ten Runtime Services functions. The precompiler may need to insert several function call combinations into the modified source module after a single compilation request. The Runtime Services functions are:

sqlastrt
: Starts run time SQL statement execution.

sqlaaloc
: Allocates an input or output SQLDA large enough to contain a specified number of SQLVAR elements. The SQLDA storage is allocated from the system storage and is managed internally by Runtime Services.

sqlastlv
: Sets the fields of an SQLDA SQLVAR element to the type, length, and address of a host variable or literal used in an SQL statement.

sqlastlva
: Sets the fields of an SQLDA SQLVAR element to the type, length, and address of a structured host variable used in an SQL statement.

sqlasetdata
: Sets the fields of several SQLDA SQLVARs with a single call. Equivalent to multiple calls to **sqlastlv**.

sqlausda
: Sets an internal database manager structure pointer to the address of an input or output SQLDA created by the user, rather than by Runtime Services functions.

sqlastls
: Sets an internal database manager structure pointer to the length and address of a host variable used to store the text of a dynamic SQL statement.

sqlacall
: Calls the database manager to execute a specific package section. Any host variable data associated with the call must have already been set up through previous Runtime Services function calls, such as **sqlaaloc** and **sqlastlv**, or **sqlausda**.

sqlacmpd
: Adds a compound SQL substatement to the list of substatements to be executed on the next call to **sqlacall**.

sqladloc
: Deallocates an SQLDA.

sqlastop
Stops run time SQL statement execution.

The calls that result from a single SQL statement should always be considered as a group. If an error occurs in one of them, this error is propagated through later calls. The SQLCODE of the SQLCA need not be tested until after the call to **sqlastop**.

There is an exception to this rule for **sqlaaloc**. For more information, see "sqlaaloc - Allocate SQLDA" on page 67.

Following is the typical order of Runtime Services calls for an executable SQL statement.

Note: No real SQL statement would require an SQLDA set up with **sqlaaloc**/**sqlastlv**, *and* an input SQLDA set up with **sqlausda**, *and* an output SQLDA set up with **sqlausda**, *and* a dynamic SQL statement set up with **sqlastls**.

```
sqlastrt(...);           /* Always                      */
sqlaaloc(...);           /* If statement contains host  */
                         /*  variables.                 */
if (sqlca.sqlcode==      /* The "if test" is an         */
    SQLA_RC_SQLVARS_SET) /*  optimization technique.    */
{                        /*  There are input parms      */
  sqlastlv(...);         /*  to sqlaaloc that cause it  */
  sqlastlv(...);         /*  to return a non-zero value */
  sqlastlv(...);         /*  if the sqlastlv calls have */
}                        /*  already been made by a     */
                         /*  previous call to this      */
                         /*  statement.                 */
sqlausda(...);           /* Input SQLDA used with OPEN  */
                         /*  or EXECUTE.                */
sqlausda(...);           /* Output SQLDA used with      */
                         /*  FETCH, DESCRIBE, or        */
                         /*  PREPARE.                   */
sqlastls(...);           /* Character host var used     */
                         /*  with PREPARE or EXECUTE    */
                         /*  IMMEDIATE.                 */
sqlacmpd(...);           /* Substatement from a         */
                         /*  compound SQL statement.    */
sqlacall(...);           /* Always                      */
sqladloc(...);           /* Optional after CLOSE,       */
                         /*  COMMIT, or ROLLBACK.       */

SQLCODE = sqlca.sqlcade; /* If STDS_LEVEL STND_SQL92E   */
                         /* is set.                     */

/* If WHENEVER SQLERROR active */
if (sqlca.sqlcode < 0)
{
   sqlastop(...);
   goto error_label;
}
/* If WHENEVER SQLWARNING active */
if (((sqlca.sqlcode > 0) AND
     (sqlca.sqlcode <> 100))
    OR
    ((sqlca.sqlcode == 0) AND
     (sqlca.sqlwarn[0] == 'W')))
{
   sqlastop(...);
   goto warning_label;
}
```

```
/* If WHENEVER NOT FOUND active */
if (sqlca.sqlcode == 100)
{
   sqlastop(...);
   goto notfound_label;
}
sqlastop(...);           /* Always                      */
```

For detailed information about the syntax, arguments, and completion codes for the run time APIs, see Chapter 8, "Runtime Services APIs," on page 67.

Starting the statement

Precompiler Services places the flag SQLA_START in the task array when **sqlastrt** is required. The **sqlastrt** API records the address of the SQLCA and the program ID, which identifies the package. It also obtains a semaphore to serialize access to DB2 structures between different threads of a single process.

Allocating input and output SQLDAs

The precompiler generates an **sqlaaloc** call when a task array flag SQLA_ALLOC_INPUT or SQLA_ALLOC_OUTPUT is returned. The function value field is the number of SQLVAR elements to be allocated (the SQLD value). This cannot be the number of host variables or literals found in the statement, since the statement may contain both input and output host variables. As well, indicator variables do not get a separate SQLVAR entry.

The precompiler decides whether to create a new SQLDA, or use an existing one. If the *sqlda_id* it passes to Runtime Services matches an existing SQLDA, and the input *sqld* parameter is less than or equal to the current number of SQLVAR elements in the SQLDA, Precompiler Services leaves the size of the existing SQLDA unchanged, but updates its SQLD field to the new value. If the *sqld* parameter is greater than the current number of SQLVAR elements in the SQLDA, Precompiler Services reallocates the SQLDA and updates its SQLN and SQLD fields.

The precompiler assigns a unique *stmt_id* to each SQL statement. This identifies the statement for which the SQLDA is allocated. Use any unique value (source line, for example) for each SQL statement within the module.

After examining the *stmt_id* and the *sqlda_id*, Runtime Services may return an SQLCODE reporting that the SQLDA has already been allocated and initialized for that particular statement. If this is the case, subsequent calls to **sqlastlv** or **sqlasetdata** for the current SQL statement can be skipped. This occurs when a fetch statement contains host variables and is performed in a loop — there may be no need to repeat the calls to initialize the SQLDA.

When **sqlaaloc** allocates or reallocates a dynamic SQLDA, Runtime Services returns SQLA_RC_OK in SQLCODE. If the **sqlastlv** or **sqlasetdata** calls have already been performed for the SQLDA for the current SQL statement, SQLCODE is `SQLA_SQLVARS_SET (4959)`.

Describing host variables and literals

The **sqlastlv** and **sqlasetdata** functions are used with **sqlaaloc**. They initialize the fields of an SQLDA SQLVAR element to the type, length, and address of a host variable or literal found in an SQL statement. The difference between the two is that **sqlastlv** initializes one SQLVAR element at a time, requiring multiple calls to set up an entire SQLDA. The **sqlasetdata** API was introduced as a way to speed this process up. It can initialize many SQLVARs in a single call, avoiding the

performance cost of repeatedly validating parameters which don't change from one host variable to the next. However, some extra effort is required to set up the parameters for **sqlasetdata**. Initially, **sqlastlv** will be discussed, followed by differences between it and **sqlasetdata**.

If either of the task flags SQLA_ALLOC_INPUT or SQLA_ALLOC_OUTPUT is present and has a non-zero value, the precompiler must process the token array. The task value indicates the number of SQLVARs which must be allocated in the SQLDA.

To set up an input SQLDA, scan the token ID array for tokens with input-related usage codes: SQLA_INPUT_HVAR, SQLA_INPUT_WITH_IND, and SQLA_LITERAL. To set up an output SQLDA, scan the token ID array for tokens with output-related usage codes: SQLA_OUTPUT_HVAR and SQLA_OUTPUT_WITH_IND. Indicator variables do not have to be separately allocated. Usage types SQLA_INPUT_WITH_IND and SQLA_OUTPUT_WITH_IND imply indicator variables.

```
EXEC SQL
   SELECT A, B, C, D, E
   INTO :VA:IA, :VB:IB, :VC, :VD, :VE
   FROM T;
```

This statement would result in the generation of five function calls to **sqlastlv**. VA, VB, VC, VD, and VE are host variables. Each requires an SQLVAR structure. IA and IB are indicator variables, and as such do not require separate SQLVAR structures. Instead, they are passed to **sqlastlv** with their host variables, VA and VB, respectively.

When generating the **sqlastlv** call, add one (1) to the SQL type of any host variable used with an indicator variable. For example, a NULL-terminated character string would normally use SQL type 460 as the sqltype parameter; however, with an indicator variable, the *sqltype* parameter is set to 461. If we assume that VA and VB are 10-byte fixed-length character strings, and VC, VD and VE are small integers, the **sqlastlv** calls for the above statement might look like the following:

```
sqlastlv( 2,0,453,10,A,&IA,NULL );
sqlastlv( 2,1,453,10,B,&IB,NULL );
sqlastlv( 2,2,500,2,&C,NULL,NULL );
sqlastlv( 2,3,500,2,&D,NULL,NULL );
sqlastlv( 2,4,500,2,&E,NULL,NULL );
```

The precompiler retrieves all necessary host variable information (name, SQL type, and size) from its symbols table, based on the ID from the token array. For literals, an SQL type of 460 should be used if the target host language supports NULL-terminated strings; otherwise, SQL type 452 should be used.

Alternatively, the five calls to **sqlastlv** in the above example could be replaced by a single call to **sqlasetdata**. This function takes a pointer to an array of *sqla_setd_list* structures, which have been initialized at run time to the type, address, and length information of the host variables being added to the SQLDA.

```
{
  struct sqla_setdata_list sqla_setd_list[5];
  sqla_setd_list[0].sqltype = 453;
  sqla_setd_list[0].sqllen  = 10;
  sqla_setd_list[0].sqldata = A;
  sqla_setd_list[0].sqlind  = &IA;

  sqla_setd_list[1].sqltype = 453;
  sqla_setd_list[1].sqllen  = 10;
```

```
  sqla_setd_list[1].sqldata = B;
  sqla_setd_list[1].sqlind  = &IB;

  sqla_setd_list[2].sqltype = 500;
  sqla_setd_list[2].sqllen  = 2;
  sqla_setd_list[2].sqldata = &C;
  sqla_setd_list[2].sqlind  = NULL;

  sqla_setd_list[3].sqltype = 500;
  sqla_setd_list[3].sqllen  = 2;
  sqla_setd_list[3].sqldata = &D;
  sqla_setd_list[3].sqlind  = NULL;

  sqla_setd_list[4].sqltype = 500;
  sqla_setd_list[4].sqllen  = 2;
  sqla_setd_list[4].sqldata = &E;
  sqla_setd_list[4].sqlind  = NULL;

  sqlasetdata( 2,0,5,sqla_setd_list,NULL );
}
```

In this case, the *sqla_setd_list* structure is dynamically allocated on the stack and then discarded after the call. Since **sqlasetdata** internally records the information in the structure, a single structure with *N* entries can be used and re-used for all **sqlasetdata** calls in an application. If an SQL statement is processed which contains more than *N* host variables, multiple calls to **sqlasetdata** can be made, each providing the information on up to *N* host variables at a time. The *start_index* and *elements* parameters to **sqlasetdata** tell the API which SQLVAR elements are to be initialized on each call.

Designating a user-defined SQLDA

When Precompiler Services sets task array flags SQLA_USDA_INPUT or SQLA_USDA_OUTPUT, the precompiler generates an **sqlausda** call. This call sets a pointer in an internal database manager data structure to the address of the user-specified input or output SQLDA. That flag's function value is the token ID of the user-specified SQLDA.

The precompiler assigns an *sqlda_id* to these SQLDAs, just as it does for dynamically allocated SQLDAs. The **sqlacall** function uses the *sqlda_id* to identify the particular input or output SQLDA used in the SQL statement to be executed.

Passing a statement

Dynamic PREPARE and EXECUTE IMMEDIATE statements specify the name of a host variable used to store the dynamic SQL statement text. The precompiler generates a statement assignment call. This provides the address and the length of that host variable to Runtime Services.

Insert an **sqlastls** call when Precompiler Services returns flag SQLA_SETS in the task array. The function value specifies the token ID of the host variable containing the SQL statement text.

The length parameter of **sqlastls** must be set to the length of the SQL statement at run time, since the length is not known during precompilation. A run time string length function, or something similar, must be used to determine the correct statement length. If the statement is NULL-terminated, as in C, a length of 0 may be passed to **sqlastls**. The **sqlastls** function will then calculate the length of the SQL statement itself. An alternative is to use a fixed-length variable. Pad shorter SQL statements with blank spaces.

Executing the section

Precompiler Services compiles SQL statements, generating sections in the package. Calls to **sqlacall** execute these sections at run time. The precompiler inserts a call to **sqlacall** when the flag SQLA_CALL occurs in the task array. The function value is passed to **sqlacall** as the *call_type* parameter.

The SQLDA IDs of the input and output SQLDAs are assigned by the precompiler. They should match IDs used in calls to **sqlaaloc** or **sqlausda** generated for this particular statement.

The section number is returned from **db2CompileSql**. The value is zero for statements that do not have a section, such as COMMIT and ROLLBACK. A special case occurs if the function value is SQLA_CONNECT or SQLA_DUOW: the statement type passed back by **db2CompileSql** should be passed to **sqlacall** in place of the section number.

The **sqlacall** function is the only Runtime Services call that actually communicates with the database manager during execution of the application program.

Deallocating an SQLDA

The **sqladloc** call deallocates SQLDAs that have been previously allocated through **sqlaaloc**. The SQLA_DEALLOC flag is currently never set (that is, Precompiler Services never tells the precompiler to deallocate an SQLDA). The function value is not used.

You can choose to have your precompiler deallocate SQLDAs on its own. You may want to deallocate all SQLDAs after each COMMIT or ROLLBACK statement. You can also deallocate SQLDAs associated with a cursor after a CLOSE statement. This would optimize in favor of storage over speed. Internal SQLDAs not deallocated at run time are freed at process end or, if appropriate, when the application library is unloaded.

Updating stand-alone SQLCODE and SQLSTATE

If SQL92E stand-alone SQLCODE and SQLSTATE support is in effect (that is, if the SQLA_STDS_LEVEL option with value SQLA_STND_SQL92E was passed to **db2Initialize**), the SQLCODE resulting from statement execution is copied into a stand-alone SQLCODE variable by the precompiler before error handling and statement termination. This is signaled by the SQLA_SQLCODE_COPY task returned in the task array. For more information, see “Support for stand-alone SQLCODE/SQLSTATE” on page 42.

Error handling

There are three SQLCA error conditions:

SQLERROR
Flow is affected if the SQLCA return code is negative.

SQLWARNING
Flow is affected if the SQLCA return code is one of the following:

- Greater than 0, and not equal to 100
- Equal to 0, and an SQLWARN0 value of W.

NOT FOUND
Flow is affected if the SQLCA return code is equal to a value of 100 after a FETCH or SELECT statement.

The precompiler can set one of two flags to redirect program control if an SQL error is detected. They are:

CONTINUE
Continue with the next instruction in the program.

GOTO host-label
Pass control to the host label when the specified condition exists. Host labels need not be prefixed with a colon (:).

Precompiler Services maintains information about active WHENEVER statements. Each of three "WHENEVER flags" inside Precompiler Services is initialized to FALSE when **db2Initialize** is called.

If Precompiler Services detects a WHENEVER SQL statement with a GOTO action, it sets the corresponding internal flag to `TRUE`. If the WHENEVER SQL statement specifies CONTINUE, the internal flag is set to `FALSE`.

Associated with each flag is a 256-byte character array that contains the label in the application program to which control may be transferred. These labels are passed back to the precompiler in the *poBuffer1*, *poBuffer2*, and *poBuffer3* parameters of the **db2CompileSql** call when a WHENEVER condition is active. The task array function flags SQLA_SQLERROR, SQLA_SQLWARNING, or SQLA_NOT_FOUND will occur in the task array, depending on which error conditions are active when an SQL statement is compiled. The task array function value is the length of the host label name.

When the precompiler encounters one of these function flags, it generates a language-specific set of instructions to test the SQLCA, and possibly transfer control to the host label. Since the call to **sqlastop** marks the end of the SQL statement's use of the SQLCA structure (possibly releasing it for use in other SQL statements by other threads), the SQLCODE must be tested before **sqlastop** is called. Precompiler Services generates the tasks in the correct order.

For example, suppose the following WHENEVER statements were found in a C source file:

```
EXEC SQL WHENEVER SQLERROR GOTO label1;
EXEC SQL WHENEVER SQLWARNING GOTO label2;
EXEC SQL WHENEVER NOT FOUND GOTO label3;
```

The following code would need to be generated by the precompiler after every call that might access the database:

```
if (sqlca.sqlcode < 0)
{
  sqlastop(NULL);
  goto label1;
}

if (((sqlca.sqlcode >  0) &&
     (sqlca.sqlcode != 100))
   ||
    ((sqlca.sqlcode == 0) &&
     (sqlca.sqlwarn[0] == 'W')))
{
  sqlastop(NULL);
  goto label2;
}

if (sqlca.sqlcode == 100)
```

```
{
  sqlastop(NULL);
  goto label3;
}
```

After your precompiler inserts the correct error handling instruction, it inserts the final **sqlastop** call.

Terminating SQL statement processing

The precompiler inserts an **sqlastop** function call when it finds an SQLA_STOP function flag in the task array.

The **sqlastop** function terminates run time statement execution. It releases the semaphore obtained by **sqlastrt**, and updates the application SQLCA with the final status of the SQL statement. Note that, except for the very rare event of problems being encountered in releasing the semaphore, the SQLCODE portion of the SQLCA is already set before **sqlastop** is called. This enables the correct SQLCODE value to be copied to a stand-alone SQLCODE, before **sqlastop** is called, in the event that stand-alone SQLCODE processing is selected by the user.

Reporting results from the SQL flagger

If SQL flagging is desired, the SQL_FLAG_OPT option must have been passed to **db2Initialize**. On each call to **db2CompileSql**, a pointer to an instance of the *sqla_flaginfo* structure is passed in. This structure contains a counted array of SQLCAs, which is used to store messages regarding the SQL statement being compiled. The **db2CompileSql** function fills in this structure.

If *sqla_flaginfo.msgs.count* is non-zero on return from **db2CompileSql**, that number of SQLCAs in the *sqlq_flaginfo.msgs.sqlca* array contain flagger-related diagnostics about the SQL statement. In this case, the precompiler should issue these messages to the user. Since the SQL flagger only returns informational messages, no change in the precompiler's other behavior is required when flagging is enabled. For more information about the *sqla_flaginfo* structure, see Chapter 4, "Precompiler data structures," on page 45.

Termination

After the last token of the source program, or after a fatal error, the precompiler terminates Precompiler Services.

Figure 2 on page 35 shows the termination tasks.

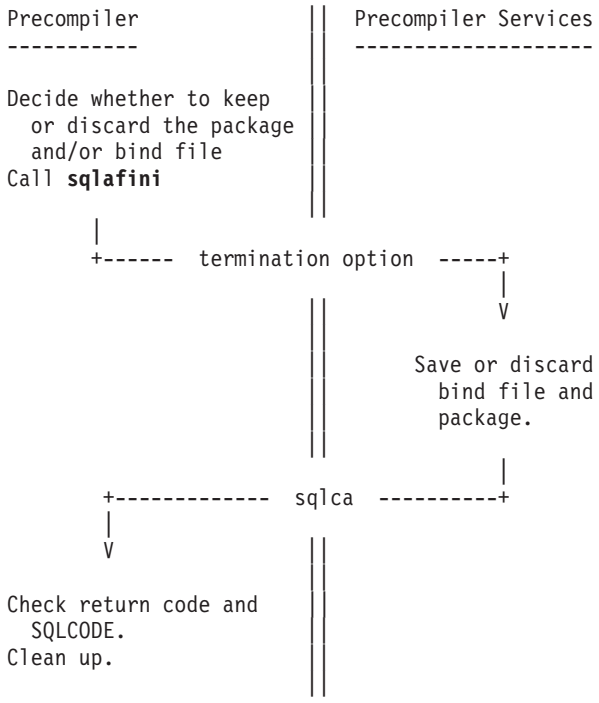

Figure 2. Terminating a Precompilation

Saving precompilation results

Depending on the kind of errors and warnings you receive while processing SQL statements, the precompiler determines whether to save both the package and the bind file, or to discard them.

Terminating Precompiler Services through sqlafini

The **sqlafini** function completes the precompilation process, and saves or discards the package or bind file as directed by the *term_option* parameter. The **sqlafini** function is the final call to Precompiler Services from the precompiler. Once this call has been issued, all other calls to Precompiler Services are rejected until a new **db2Initialize** call is successfully completed.

The **sqlafini** function returns data to two SQLCA fields. The fields indicate whether or not the package or bind file was saved. The fields are SQLWARN6 and SQLWARN7. They are always set, regardless of the termination option or any error condition that may have occurred. These single byte fields are used as follows:

- SQLWARN6 — If set to 1, the package was saved successfully. Otherwise, it was discarded.
- SQLWARN7 — If set to 1, the bind file was saved successfully. Otherwise, it was either deleted, or a disk error occurred, depending on the type of error.

Check these fields, even if an error is returned in SQLCODE. The package or bind file may actually have been created successfully before the error condition occurred. This is particularly important when handling break signals.

Cleaning up

Close all open files, release the break handler, and inform the application programmer of the precompile completion status.

Chapter 3. Advanced precompiler design

Compound SQL

Compound SQL (see the *SQL Reference*) is somewhat more difficult to precompile than traditional atomic SQL statements. A compound SQL statement is composed of substatements which are essentially individual SQL statements executed in ″batch mode″. Substatements are separated by semicolons (;), may have their own host variables, and so on, and are sent to **db2CompileSql** individually. The statement itself begins with the keywords ″BEGIN COMPOUND ...″, which forces Precompiler Services into a compound SQL ″mode″, wherein substatements are processed appropriately until the statement ends (marked with the ″END COMPOUND″ keywords.)

For each compound SQL statement, there is only one **sqlastrt** call at the beginning, and one **sqlastop** call at the end. In between, each substatement is handled with its own calls to Runtime Services APIs, such as **sqlaaloc**, **sqlastlv**, **sqlasetdata**, or **sqlausda**. Substatements are not individually executed with calls to **sqlacall**, like ordinary SQL. Instead, a call to **sqlacmpd** is generated, which adds the substatement to an internal list. At the end of the compound SQL statement as a whole, a call to **sqlacall** is generated, which initiates the execution of the list of substatements built with the **sqlacmpd** calls.

Since each substatement must be handled individually, the precompiler must recognize the semicolon substatement separator as a type of statement terminator. During ordinary (non-compound) SQL processing, a semicolon encountered in an SQL statement should cause the statement (as accumulated to that point) to be sent to Precompiler Services for compilation. This is required, since the semicolon may indicate a compound SQL statement. Since the semicolon is not a valid statement terminator in every language, the precompiler should then check the results of **db2CompileSql** to ensure that the semicolon is indeed part of a compound SQL statement.

If, for example, the following code were encountered in a COBOL application processed by the DB2 COBOL precompiler:

```
EXEC SQL
    FETCH C1 INTO :MY-VAR;
```

the precompiler would send the FETCH statement (up to, but not including, the semicolon) to **db2CompileSql**. This API would then return information indicating that this was not the beginning of a compound SQL statement, invalidating the use of the semicolon terminator. An error message would then be issued by the precompiler.

If, on the other hand, the statement were as follows:

```
EXEC SQL BEGIN COMPOUND ATOMIC STATIC
    EXECUTE STMT_1 USING :HV-1, :HV-2;
    EXECUTE STMT_2 USING DESCRIPTOR :MY-SQLDA;
END COMPOUND END-EXEC
```

the precompiler would send everything from `BEGIN` to `HV-2` to **db2CompileSql**, which would return two tasks in the task array: SQLA_START and SQLA_BEGIN_COMPOUND. SQLA_START is processed normally, and

SQLA_BEGIN_COMPOUND signals the precompiler to enter its own ″compound SQL mode″. While in this mode, it does not need to search for the EXEC SQL keyword pair to find new statements. Instead, it repeatedly accumulates substatements, passes them to **db2CompileSql**, and performs the resulting tasks.

Note that the first substatement is appended to the BEGIN COMPOUND phrase by the precompiler. It has not been processed by Precompiler Services, however. Instead, as part of BEGIN COMPOUND handling, Precompiler Services has moved it to the beginning of the SQL statement buffer, in preparation for immediate resubmission to **db2CompileSql**. During this second call, the substatement will be processed normally. After handling the resulting task array, the precompiler can proceed to the second and subsequent substatements.

As mentioned above, substatements in compound SQL are not executed individually with **sqlacall**. Instead, one of the function flags returned when a substatement is processed is SQLA_CMPD, which instructs the precompiler to generate a call to **sqlacmpd**. The function value for SQLA_CMPD provides a *call_type* to be passed to **sqlacmpd**, much like **sqlacall**.

When Precompiler Services processes the END COMPOUND clause, it returns a normal SQLA_CALL function flag and value pair to trigger execution of the statement, followed by SQLA_STOP. At this point, the precompiler can exit its compound SQL mode.

If the precompiler is processing a compound SQL statement, and finds that it cannot obtain the next substatement (due to, for example, an end-of-file, or a ″real″ statement terminator being found), it can force Precompiler Services out of compound SQL mode by passing a zero-length string to **db2CompileSql**.

The compound SQL syntax is further extended by the optional STOP AFTER FIRST :*n* STATEMENTS clause. This allows the application to limit execution to the first *n* substatements, where *n* is a SMALLINT host variable containing the number of substatements to be executed.

This is implemented through simple code generated by the precompiler. The compound SQL statement:

```
exec sql begin compound atomic static
         stop after first :n statements
    execute stmt_1 using :hv_1, :hv_2;
    execute stmt_2 using descriptor :my_sqlda;
end compound;
```

could be precompiled in two ways, depending on the mechanisms supported by the host language:

```
/* Method 1 */

sqlastrt(...);

if( n >= 1 )
{
  /* 1st substmnt */
  sqlaaloc(...);
  sqlasetdata(...);
  sqlacmpd(...);
}

if( n >= 2 )
{
  /* 2nd substmnt */
```

```
/* Method 2 */

sqlastrt(...);

if( n < 1 )
  goto sql_label_XXX;
/* 1st substmnt */
sqlaaloc(...);
sqlasetdata(...);
sqlacmpd(...);

if( n < 2 )
  goto sql_label_XXX;
/* 2nd substmnt */
```

```
  sqlausda(...);          |    sqlausda(...);
  sqlacmpd(...);          |    sqlacmpd(...);
}                         |
                          |  sql_label_XXX:
sqlacall(...);            |    sqlacall(...);
sqlastop(...);            |    sqlastop(...);
```

Method 2 is slightly more efficient than method 1, but may be slightly more difficult to generate. The task functions and values returned from Precompiler Services support both methods.

When a STOP AFTER FIRST clause is used at the beginning of a compound SQL statement, the tasks for each substatement will include the SQLA_TEST function flag, with the function value containing the token ID of the control variable. In the above example, this would be *n*. The precompiler then generates a test of the control variable, to determine whether to execute the substatement's run time API calls. The task function flag SQLA_CMPD_MARK is returned by **db2CompileSql** near the end of the compound SQL statement, when it is time for the precompiler to generate the target label used in method 2. If method 1 is being used, the SQLA_CMPD_MARK function flag can be ignored.

Note that when a STOP AFTER FIRST clause is used, the token array passed into the first call to **db2CompileSql** contains both the token ID of the control variable, *and* the token IDs of any host variables used in the first substatement. Precompiler Services alters the contents of the token ID array so that it contains only the host variable IDs of the first substatement; this is similar to the way in which it moves the first substatement to the beginning of the SQL statement buffer before returning to the precompiler. This is in preparation for the first substatement being resubmitted to **db2CompileSql** for compilation.

The CREATE TRIGGER statement

The CREATE TRIGGER statement is syntactically similar to compound SQL, because it can have substatements separated by semicolons. These substatements cannot stand alone as individual SQL statements, however. The entire statement must be processed at once by Precompiler Services.

If the precompiler supports compound SQL, it will likely see the semicolons, and send the substatements to **db2CompileSql** one by one. If this happens, Precompiler Services returns the SQLA_NEXT_SUBSTATEMENT function flag as the only entry in the task array. When the precompiler sees this flag, it should send the next substatement (or just the remainder of the SQL statement, if possible) to **db2CompileSql**. This process repeats until the entire CREATE TRIGGER statement has been accumulated by Precompiler Services, whereupon the task array will then be returned to the precompiler with the tasks for executing the entire CREATE TRIGGER statement.

Note that the precompiler does not have to send in the CREATE TRIGGER statement in this piecemeal fashion. An entire statement, semicolons and all, is perfectly acceptable. Any substatements that are passed in, however, should not include the trailing semicolon.

If the precompiler receives the SQLA_NEXT_SUBSTATEMENT function flag, but there are clearly no more substatements to be found (for example, an end-of-file has been read), the CREATE TRIGGER statement is in error. In this case, the precompiler should send an empty statement to **db2CompileSql** to indicate the error condition.

Optimizing function calls

Your precompiler should generate the correct Runtime Services function calls for each entry in the task array, but you can create a precompiler that will optimize the modified source file. Two examples of how to increase application performance follow:

- Avoid redundant initializations
- Use multiple dynamic SQLDAs.

These two techniques can have a dramatic effect on the run time environment of precompiled applications.

Avoid redundant initializations

The tasks required for a FETCH statement include allocating a dynamic SQLDA with **sqlaaloc**, and storing host variable data with as many **sqlastlv** or **sqlasetdata** calls as necessary. Your precompiler generates this code before it generates the **sqlacall** call for the FETCH request.

If the same FETCH request occurs a number of times (in a loop, for example), there may be no need to initialize the SQLDA repeatedly. Your precompiler can insert code to test if the SQLDA needs to be initialized.

The first time the FETCH is executed, the SQLDA is initialized so that **sqlaaloc** sets the SQLCODE element in the SQLCA to zero. During further calls to the same statement, **sqlaaloc** finds that the specified SQLDA is already initialized. When this happens, **sqlaaloc** sets SQLCODE to SQLA_RC_SQLVARS_SET (+4959).

After inserting **sqlaaloc** in the modified source, the precompiler can insert a test for SQLCODE equal to SQLA_RC_SQLVARS_SET. If this is the case during execution, the application does not need to call the **sqlastlv** or **sqlasetdata** functions that follow. Here is a sample of the standard and optimized code in C:

```
/* Standard code for EXEC SQL FETCH C1 INTO :name; */
{
sqlastrt(...)
sqlaaloc(...)
sqlastlv(...)
sqlacall(...)
sqlastop(...)
}

/* Optimized code for EXEC SQL FETCH C1 INTO :name; */
{
sqlastrt(...)
sqlaaloc(...)
if (sqlca.sqlcode != SQLA_RC_SQLVARS_SET) /* 4959 */
{
sqlastlv(...)
}
sqlacall(...)
sqlastop(...)
}
```

Note: This optimization is possible only if the host variable addresses remain constant. The **sqlastlv** cannot be skipped if the host variables are pointers or temporary variables, because the addresses may change from one iteration to the next.

Use multiple dynamic SQLDAs

Suppose you are optimizing with the previous method. It would be of little benefit to use the same SQLDA for two different FETCH statements, even if they use the same host variables. Although the **sqlaaloc** finds that the SQLDA has been initialized, it was not initialized for the same statement. Consequently, the SQLCODE will not equal zero, and the application will initialize the SQLDA.

To avoid this overhead, determine which statements warrant their own SQLDAs. The easy answer is to use an SQLDA ID for each SQL statement in the source file. This would allocate new memory for each statement, instead of reusing existing allocated memory.

Perhaps a better approach is to use unique SQLDA IDs for each FETCH statement. Use two other SQLDA IDs for all other SQL statements (one for input host variables, and one for output host variables).

As you develop your precompiler, you can determine the best means to optimize the code it generates for the modified source file.

Support for structure host variables

The DB2 COBOL and C/C++ precompiler supports declaration and use of host structures. These are composite data items whose member parts are themselves host variables. The members can be used individually, but more importantly, the entire composite data item can be used in SQL statements as a type of ″shorthand″ for the members it contains. Typically, such structures would contain members corresponding to the columns in a database table. In such cases, a structure host variable could be used as the only host variable in a FETCH statement, for example. This would be treated as equivalent to listing out the column host variables one by one. References to individual members can be qualified, as in *struct.member*, but they do not have to be. For more information, see the description of host structures in COBOL and C/C++ in the *Application Development Guide*. Structure support is a very popular feature, and any writer of a full-featured custom precompiler should consider implementing it.

Structure support is implemented almost totally by the precompiler. Additional responsibilities include:

- Accepting and parsing structure and indicator array host variable definitions.
- Maintaining a hierarchy of host variable symbols tables.
- If partially or totally unqualified structure member references are permitted, being able to complete the qualification to avoid ambiguity of reference at compile time.
- Recognizing structure references in SQL statements, and expanding them to the equivalent list of member variables, before sending the statement to Precompiler Services.
- Ensuring structure members are uniquely named. This is required, because all structure members must be passed to Precompiler Services through the **sqlaalhv** interface, just like ″ordinary″ host variables. As stated earlier, the names of such variables must be unique. Since most host languages permit duplicate names of fields in different structures, the precompiler must provide a mechanism to ″rename″ the host variables to ensure uniqueness. Note that if host variables are declared to Precompiler Services with aliases, tokens for error messages which mention host variable names must be scrutinized, to map the alias reported from

Precompiler Services back to the original host variable name for the user. See messages SQL0104N, SQL0303N, SQL0307N, SQL0312N, SQL0324N, and SQL4942N.

Another issue relates to the expansion of structure host variables during statement processing. The use of a structure name in an SQL statement is equivalent to using a comma delimited list of host variables. If used inappropriately (for example, in a CONNECT statement: `EXEC SQL CONNECT TO :a,:b,:c`), such a list generates a syntax error. If the user had entered the statement this way, the syntax error complaining about the unexpected comma following "a" would be reasonable. If, however, *a*, *b*, and *c* were part of a structure *s*, and the user had instead coded `EXEC SQL CONNECT TO :s`, the error message would still refer to the errant comma. This might be quite a difficult error to fix, unless the user realized that structure *s* had been expanded by the precompiler.

To fix this problem, Precompiler Services will return error SQL0087N if a multi-member structure has been expanded when it should not have been. To enable this, the precompiler must do two things. First, when it calls **db2Initialize**, it must pass the option `SQLA_TOKEN_USE_INITIALIZED_OPT (1000)`, set to some non-zero value. This indicates that the usage fields in the token ID array have been initialized by the precompiler before each **db2CompileSql** call. Secondly, the precompiler must initialize these usage fields to one of two values: `SQLA_MULTIPLE_STRUCT_FIELD (9)` indicates that the corresponding token was expanded from a structure host variable, and `SQLA_ATOMIC_FIELD (10)` indicates that the token was *not* expanded.

Support for 255-byte host variable names and labels

With DB2 UDB Version 5 and higher, Precompiler Services supports 255-byte host variable and label names. This will have little impact on Precompiler Services clients, except for the fact that the three buffers passed to the **db2CompileSql** API need to be 256 (not 128) bytes long.

In order for Precompiler Services to distinguish between down-level clients passing existing 128-byte buffers, and newer clients passing 256-byte buffers, an extra option must be passed to **db2Initialize** to enable full 255-byte label support. If the internal option `SQLA_USE_LONG_LABELS (1001)` is passed in the option array to **db2Initialize** with a value of 1, Precompiler Services will assume the buffers passed to **db2CompileSql** can store up to 256 bytes. If this option is not passed, labels will be limited to 128 bytes, since the **db2CompileSql** buffers will be assumed to be only 128 bytes long.

Note that this has no effect on the support for long host variable names. They can be up to 255 bytes long, even with down-level Precompiler Services clients.

Support for stand-alone SQLCODE/SQLSTATE

Stand-alone SQLCODE and SQLSTATE variables, as defined in ISO/ANSI SQL92, are supported through the SQLA_STDS_LEVEL **db2Initialize** option, with value `SQLA_STND_SQL92E`. This means that precompiled applications do not have to define an SQLCA structure in their application. In fact, if they do attempt to define one through EXEC SQL INCLUDE SQLCA, in an application precompiled with `SQLA_STND_SQL92E`, Precompiler Services will return an SQL0143W warning, and will not insert an SQLCA in the modified source code.

When SQLA_STDS_LEVEL or SQLA_STND_SQL92E is specified, the following points apply:

- The precompiler should generate a declaration for an SQLCA-like structure in the modified source code.
- When the precompiler generates code to call **sqlastrt** and to perform error handling, it will do so using this precompiler-generated SQLCA. Here, SQLCA-like means that the SQLCODE and SQLSTATE fields may need to be given different names, so that they do not collide with the application program's own stand-alone variables. For example, the DB2 C precompiler uses **sqlcade** and **sqlstat**. Since the SQLCA is not intended to be manipulated by the application code, it does not have to be called SQLCA.
- Precompiler Services will return the task SQLA_SQLCODE_COPY (20) in the task array resulting from **db2CompileSql**. The precompiler should generate code as follows, based on whether declarations for SQLCODE and SQLSTATE have been made:

```
SQLCODE declared?              N   Y   N   Y
SQLSTATE declared?             N   N   Y   Y

generate SQLCODE assignment    Y   Y   N   Y
generate SQLSTATE assignment   N   N   Y   Y
```

  That is, just as the SQLCA structure is presumed to have been declared by the application in non-SQLA_STND_SQL92E processing, here it is assumed that a 4-byte integer SQLCODE has been declared. The precompiler generates an assignment as follows:

```
:
sqlacall( ... );
SQLCODE = sqlca.sqlcade;
:
```

  If a declaration for SQLCODE has been made in the DECLARE SECTION, the precompiler should still generate the code, as above. If a DECLARE SECTION contains declarations for both SQLCODE and SQLSTATE, assignments should be made for both:

```
:
sqlacall( ... );
SQLCODE = sqlca.sqlcade;
strncpy(SQLSTATE, sqlca.sqlstat, sizeof(SQLSTATE));
SQLSTATE[sizeof(SQLSTATE)-1] = '\0';
:
```

  If only an SQLSTATE declaration is made, the SQLCODE assignment should be omitted:

```
:
sqlacall( ... );
strncpy(SQLSTATE, sqlca.sqlstat, sizeof(SQLSTATE));
SQLSTATE[sizeof(SQLSTATE)-1] = '\0';
:
```

- In the case of an error being returned from **db2CompileSql**, the precompiler should generate code to set SQLCODE and SQLSTATE to the precompile-time SQLCODE and SQLSTATE values. For example, the following statement:

```
exec sql foobar;
```

  would cause the following to be generated in the modified source file by the DB2 C precompiler:

```
/*
SQL0104N  An unexpected token "END-OF-STATEMENT" was found
following "foobar".  Expected tokens may include:  "JOIN
<joined_table>".  SQLSTATE=42601
*/
```

```
{
  SQLCODE = -104;
  strncpy(SQLSTATE, "42601", sizeof(SQLSTATE));
  SQLSTATE[sizeof(SQLSTATE)-1] = '\0';
}
```

Note that no task is returned from Precompiler Services to trigger this assignment of an error code. It is up to the precompiler to generate the code when an error is returned from **db2CompileSql**, if `SQLA_STND_SQL92E` has been specified.

To summarize the responsibilities of the precompiler with respect to stand-alone SQLCODE/SQLSTATE support, the precompiler must:

- Automatically generate an SQLCA declaration
- Determine and generate the appropriate assignments (based on whether SQLCODE or SQLSTATE has been declared) when the SQLA_SQLCODE_COPY task is received
- Determine and generate the appropriate assignments (based on whether SQLCODE or SQLSTATE has been declared) when an error is returned from **db2CompileSql**.

The SET CURRENT PACKAGE PATH statement

Special processing of the token ID array is required for the SET CURRENT PACKAGE PATH statement, support for which has been added as of DB2 UDB Version 8.2. An additional SQLVAR must be generated for each input item (there are no output items for this statement). Note that the number of input variables returned in the task array for the SQLA_ALLOC_INPUT task will have already been doubled by Precompiler Services. The extra SQLVAR generated for each input item will indicate whether the SQLVAR that follows is a literal or an input host variable. The extra SQLVAR must be initialized with a SMALLINT type and length, as well as the address of one of two 2-byte variables that have had declarations generated for them in the modified source file by the precompiler. One of these variables must be assigned a value of 1 to represent a literal, and the second variable must be assigned a value of 2 to represent an input host variable. These specific values are required for the Runtime Services functions to process the input items correctly.

The header file (`sqladef.h`) that is shipped with DB2 UDB includes the following defined constants:

```
#define  SQL_IS_LITERAL         1
#define  SQL_IS_INPUT_HVAR      2
```

The modified source file would include declarations similar to the following:

```
short  sqlIsLiteral   = SQL_IS_LITERAL;
short  sqlIsInputHvar = SQL_IS_INPUT_HVAR;
```

Chapter 4. Precompiler data structures

There are six data structures used between the precompiler and Precompiler Services. The precompiler allocates each of the following:

- Precompiler option array
- Program identifier string
- Token identifier array
- Task array
- Return token structure
- Flagger diagnostics structure.

Precompiler option array

The precompiler option array is an input parameter used by **db2Initialize**. This array of logically paired 4-byte integers provides information required to initialize Precompiler Services.

Figure 3 shows the precompiler option array. Each cell contains a 4-byte integer.

```
Offset   0  +------------------+-------------------+
            | header.allocated | header.used       |
         8  +------------------+-------------------+
            | option(0).type   | option(0).val     |
        16  +------------------+-------------------+
            | option(1).type   | option(1).val     |
        24  +------------------+-------------------+
            | option(2).type   | option(2).val     |
        32  +------------------+-------------------+
            .
            .
            .
```

Figure 3. Precompiler Option Array

The first logical pair is the header. It specifies the number of pairs allocated for the option data, and the number of pairs actually used. The header itself is not included in the count of pairs allocated and used.

The following options are always specified:

- Package creation (SQLA_ACCESS_PLAN)
- Bind file creation (SQLA_BIND_FILE).

Each option is represented by a pair of 4-byte integers. The first integer contains the type of option, and the second integer contains the actual value for the option.

Program identifier string

The program identifier string (PID) is an output parameter of **db2Initialize**. After DB2 Version 7, the PID is a character array of a maximum of 162 bytes that is used to uniquely identify the modified source file and associate it with its package. The PID for DB2 Version 6.1 is a 40-byte character array. The precompiler generates a variable definition in the modified source file and initializes it with the alphanumeric contents of the PID.

Runtime Services uses the PID to execute sections in the package.

Token identifier array

The token identifier array is both an input and an output parameter of **db2CompileSql**. It has the same basic structure as the option array and is used to pass host variable and literal information between the precompiler and Precompiler Services.

Figure 4 shows the token identifier array. Each cell contains a 4-byte integer.

```
Offset   0  +------------------+-------------------+
            | header.allocated | header.used       |
          8  +------------------+-------------------+
            | token(0).id      | token(0).use      |
         16  +------------------+-------------------+
            | token(1).id      | token(1).use      |
         24  +------------------+-------------------+
            | token(2).id      | token(2).use      |
         32  +------------------+-------------------+
            .
            .
            .
```

Figure 4. Token Identifier Array

The first pair in the token identifier array is the header. The first 4-byte integer is the number of pairs allocated. The second 4-byte integer is the actual number of pairs used. The remaining logical pairs identify host variables and literals and how they were used in an SQL statement.

For host variables, the first element of a token pair contains a non-zero, 4-byte integer that identifies a specific host variable. The precompiler fills in this value before calling Precompiler Services to compile the SQL statement. There is one entry for each occurrence of a host variable found in the SQL statement.

Note: An SQLDA in a dynamic statement is considered a host variable even though it does not appear in a host variable declaration section.

Precompiler Services returns a usage code for each token ID in the second integer of each token pair. The code specifies how the host variable was used within the SQL statement.

For literals, the first element of a token pair contains an instance of the return token structure. This is returned by Precompiler Services. The second element of the pair contains a usage code indicating that this entry represents a literal found in the SQL statement.

There can be thousands of array entries, although most SQL statements never contain that many host variables or literals. The precompiler can determine some practical size for normal usage and then, if necessary, allocate a larger size array if it finds an SQL statement that contains more host variables or literals than are currently allocated.

Task array

The task array is an output parameter of **db2CompileSql**. Upon completion of a call to **db2CompileSql**, the task array specifies the run time function calls and data to be used in the modified source of an application program.

Figure 5 shows the task array. Each cell contains a 4-byte integer.

```
Offset   0  +------------------+-------------------+
            | header.allocated | header.used       |
         8  +------------------+-------------------+
            | task(0).func     | task(0).val       |
        16  +------------------+-------------------+
            | task(1).func     | task(1).val       |
        24  +------------------+-------------------+
            | task(2).func     | task(2).val       |
        32  +------------------+-------------------+
            .
            .
            .
```

Figure 5. Task Array

The task array has the same structure as the precompiler option array and the token ID array. The first integer of the header is the number of pairs allocated for task information. The precompiler supplies this number.

The second integer of the header specifies the number of task pairs returned. Precompiler Services calculates that figure.

If the number of tasks returned is greater than the number allocated in the array, Precompiler Services sets the second header integer to the required size, and returns an error (SQL4919N) in the SQLCA. The precompiler can then reallocate the task array to the size required and attempt the compile request again.

The remaining pairs are function flags and function values. Each function flag represents a particular task the precompiler must perform when that function flag is active. The function value contains data or a value needed to complete the task.

Return token structure

This structure is used by the SQLA_INC_TEXTFILE tasks. The structure is used in place of the 4-byte integer function value, in the task array returned by **db2CompileSql**. It is also used when passing literal information in the token ID array. It is used in place of the 4-byte integer ID value in the token ID array returned by **db2CompileSql**.

The offset field contains the offset of the appropriate literal string within the SQL statement. Length contains the length of the string. Figure 6 on page 48 shows the structure.

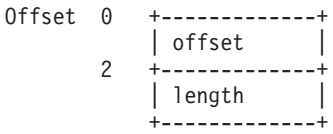

Figure 6. Return Token Structure

Flagger diagnostics structure

This structure is passed to the **db2CompileSql** API when SQL flagging is requested. The precompiler must initialize the version field to SQLA_FLAG_VERSION (currently 1; the value of this constant should be obtained from the **sqlaprep** header file by the precompiler.) Upon completion of the API, *count* SQLCAs within the structure have been set up with diagnostic information from the flagger. The precompiler should then display these messages to the user. Figure 7 shows the structure.

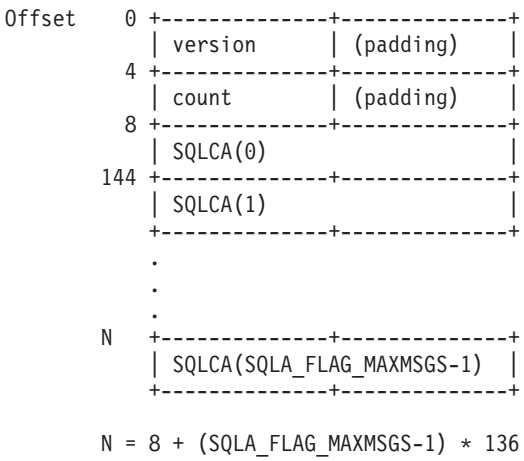

Figure 7. Flagger Diagnostics Structure

Chapter 5. Runtime data structures

Multiple variable SQLDA initialization structure

This structure can be used with **sqlasetdata**, an alternative to **sqlastlv**. Typically, the precompiler generates an array of *sqla_setd_list* structures in the modified source files. If there are any structured types, the *sqla_setds_list* structure is also generated. Code is also generated to initialize this array with information describing the host variables used in an SQL statement. The array is then passed to **sqlasetdata** to initialize the internal SQLDA. Figure 8 shows the structure.

```
Offset   0  +--------------+--------------+
            | sqltype      | (padding)    |
          4  +--------------+--------------+
            | sqllen                      |
          8  +--------------+--------------+
            | sqldata                     |
         12  +--------------+--------------+
            | sqlind                      |
            +--------------+--------------+
```

Figure 8. Multiple Variable SQLDA Initialization Structure

Runtime information structure

A pointer to this structure is passed to the second argument of the **sqlastrt** API at run time. It is currently only used to indicate how data stored in and retrieved from wchar_t host variables in C applications should be treated. The wc_size field provides the size (in bytes) of the wchar_t C data type as implemented by the application compiler. It should be initialized to sizeof(wchar_t). The wc_type field should be set to `SQL_WCHAR_NOCONVERT (0)` if wchar_t host variables in the application contain graphic data in DBCS format. If such host variables contain data in native wchar_t format, the wc_type field should be set to `SQL_WCHAR_CONVERT (1)`. The ID field should be set to `SQLARTIN`, and the unused field should be set to blank spaces.

In the DB2 C precompiler, the WCHARTYPE option controls the setting of this flag, allowing the application programmer to indicate the desired behavior. If WCHARTYPE is inappropriate or unsupported, the precompiler can pass a NULL pointer to **sqlastrt**. Figure 9 shows the structure.

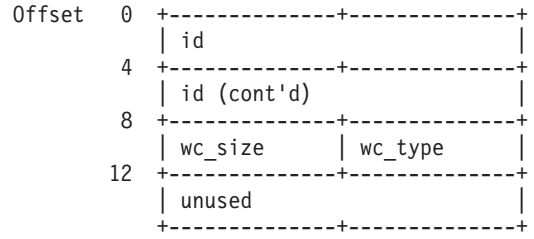

Figure 9. Runtime Information Structure

Chapter 6. Option APIs

sqlaoptions - Parse Option String

This API parses an input string into an option array or other parameters for the Precompiler Services initialization API (**db2Initialize**). Storage for the option array must be allocated by the caller. However, storage for database name (*ppszDBName*), user ID (*ppszUserid*), password (*ppszPassword*), package name (*ppszPackageName*), and bind file (*ppszBindFile*), as well as text-based option values such as COLLECTION, is allocated by the API on behalf of the caller. This storage must be freed after **db2Initialize** is called, by a call to **sqlaoptions_free**, a function that should be called even if **sqlaoptions** returns an unsuccessful SQLCODE.

Valid option string contents for SQLAO_PREP_SVCS_API are as follows:

```
ACTION {ADD | REPLACE [RETAIN {YES | NO}] [REPLVER version-id]}
AS400NAMING {SYSTEM | SQL}
BINDFILE [ USING  bind-file ]
BLOCKING {UNAMBIG | ALL | NO}
CALL_RESOLUTION {IMMEDIATE | DEFERRED}
CCSIDG double-ccsid
CCSIDM mixed-ccsid
CCSIDS sbcs-ccsid
CHARSUB {DEFAULT | BIT | SBCS | MIXED}
CNULREQD {YES | NO}
COLLECTION collection-id
CONNECT {1 | 2}
DATABASE dbname
DATETIME {DEF | USA | EUR | ISO | JIS | LOC}
DBPROTOCOL {DRDA | PRIVATE}
DEC {31 | 15}
DECDEL {PERIOD | COMMA}
DEFERRED_PREP {YES | NO | ALL}
DEGREE {1 | degree-of-I/O-parallelism | ANY}
DISCONNECT {EXPLICIT | CONDITIONAL | AUTOMATIC}
DYNAMICRULES {BIND | RUN}
ENCODING {ASCII | EBCDIC | UNICODE | CCSID}
EXPLAIN {NO | YES | ALL}
EXPLSNAP {NO | YES | ALL}
FEDERATED {NO | YES}
FUNCPATH schema-name [ {,schema-name} ... ]
GENERIC string
IMMEDWRITE {NO | YES | PH1}
INSERT {DEF | BUF}
ISOLATION {CS | RR | UR | RS | NC}
KEEPDYNAMIC {YES | NO}
LANGLEVEL {SAA1 | MIA | SQL92E}
LEVEL consistency-token
OPTHINT number
OS400NAMING {SYSTEM | SQL}
OWNER authorization-id
PACKAGE [ USING package-name ]
PATH schema-name [ {,schema-name} ... ]
QUALIFIER qualifier-name
QUERYOPT optimization-level
RELEASE {COMMIT | DEALLOCATE}
{REOPT VARS | NOREOPT VARS}
SORTSEQ {JOBRUN | HEX}
SQLERROR {NOPACKAGE | CHECK | CONTINUE}
SQLFLAG {MVSDB2V23 | MVSDB2V31 | MVSDB2V41 | SQL92E} SYNTAX
SQLRULES {DB2 | STD}
SQLWARN {YES | NO}
```

```
STRDEL {APOSTROPHE | QUOTE}
SYNCPOINT {ONEPHASE | TWOPHASE | NONE}
TEXT label
TRANSFORMGROUP string
USER userid
USING password
VALIDATE {RUN | BIND}
VERSION version-id
```

Note: The FEDERATED option is supported in Version 7, FixPak 2.

In general, keywords and values in the option string are the same as the parts of the DB2 PRECOMPILE PROGRAM command that are applicable to Precompiler Services. For example, the DB2 PREP NOLINEMACRO option is not accepted by **sqlaoptions**, because it does not apply to Precompiler Services. It is used exclusively by the DB2 C precompiler. A database name, user ID and password can also be included, for use in connecting to the database prior to calling **db2Initialize**. All options are returned through the option structure, with the following exceptions:

- *package-name* and *bind-file*, if supplied, are returned through their own output parameters (*ppszPackage* and *ppszBindFile*, respectively). The caller passes these on as parameters to **db2Initialize**.
- *dbname*, *userid*, and *password*, if supplied, are returned through their own output parameters (*ppszDBName*, *ppszUserid*, and *ppszPassword*, respectively.) The caller would use these in a CONNECT statement, prior to calling **db2Initialize**.

Valid option string contents for SQLAO_BIND_API are as follows:

```
ACTION {ADD | REPLACE [RETAIN {YES | NO}] [REPLVER version-id]}
AS400NAMING {SYSTEM | SQL}
BLOCKING {UNAMBIG | ALL | NO}
CCSIDG double-ccsid
CCSIDM mixed-ccsid
CCSIDS sbcs-ccsid
CHARSUB {DEFAULT | BIT | SBCS | MIXED}
CLIPKG number
CNULREQD {YES | NO}
COLLECTION collection-id
DATETIME {DEF | USA | EUR | ISO | JIS | LOC}
DBPROTOCOL {DRDA | PRIVATE}
DEC {31 | 15}
DECDEL {PERIOD | COMMA}
DEGREE {1 | degree-of-I/O-parallelism | ANY}
DYNAMICRULES {BIND | RUN}
ENCODING {ASCII | EBCDIC | UNICODE | CCSID}
EXPLAIN {NO | YES | ALL}
EXPLSNAP {NO | YES | ALL}
FEDERATED {NO | YES}
FUNCPATH schema-name [ {,schema-name} ... ]
GENERIC string
IMMEDWRITE {NO | YES | PH1}
INSERT {DEF | BUF}
ISOLATION {CS | RR | UR | RS | NC}
KEEPDYNAMIC {YES | NO}
OPTHINT number
OS400NAMING {SYSTEM | SQL}
OWNER authorization-id
PATH schema-name [ {,schema-name} ... ]
QUALIFIER qualifier-name
QUERYOPT optimization-level
RELEASE {COMMIT | DEALLOCATE}
{REOPT VARS | NOREOPT VARS}
SORTSEQ {JOBRUN | HEX}
```

```
SQLERROR {NOPACKAGE | CHECK | CONTINUE}
SQLWARN {YES | NO}
STRDEL {APOSTROPHE | QUOTE}
TEXT label
TRANSFORMGROUP string
VALIDATE {RUN | BIND}
```

Note: The FEDERATED option is supported in Version 7, FixPak 2.

API Include File:

sqlaprep.h

C API Syntax:

```
SQL_API_RC SQL_API_FN
      sqlaoptions(void           *pscInputString,
                  struct sqlopt  *pSqlOptStruct,
                  short          *psiDBNameLength,
                  char          **ppszDBName,
                  short          *psiUseridLength,
                  char          **ppszUserid,
                  short          *psiPasswordLength,
                  char          **ppszPassword,
                  short          *psiMsgFileLength,
                  char          **ppszMsgFile,
                  short          *psiPackageLength,
                  char          **ppszPackage,
                  short          *psiBindFileLength,
                  char          **ppszBindFile,
                  long            lTarget,
                  void           *pvMemList,
                  struct sqlca   *pSqlca )
```

Generic API Syntax:

```
SQL_API_RC SQL_API_FN
      sqlgoptions(void           *pscInputString,
                  struct sqlopt  *pSqlOptStruct,
                  short          *psiDBNameLength,
                  char          **ppszDBName,
                  short          *psiUseridLength,
                  char          **ppszUserid,
                  short          *psiPasswordLength,
                  char          **ppszPassword,
                  short          *psiMsgFileLength,
                  char          **ppszMsgFile,
                  short          *psiPackageLength,
                  char          **ppszPackage,
                  short          *psiBindFileLength,
                  char          **ppszBindFile,
                  long            lTarget,
                  void           *pvMemList,
                  struct sqlca   *pSqlca )
```

API Parameters:

pscInputString
The input string containing the option data to be parsed. This is a VARCHAR-like string, consisting of a 2-byte length field followed by an array of characters.

pSqlOptStruct
A pointer to an *sqlopt* option structure allocated by the caller. The format of

this structure is defined in sql.h. When **sqlaoptions** is called, the allocated field in the structure header must be set to indicate the number of entries allocated. On successful return, the used field in the structure header indicates how many fields were used.

Note: If the amount used exceeds the amount allocated on return, too few option spaces were available, and some options present in the input string were not transferred to the option array. This condition should be checked for *even if the SQLCODE indicates success*. To prevent this problem in DB2 Version 5 or higher, allocate 50 option entries in the structure.

This parameter will always contain at least two entries, one for package and one for bindfile. See below for details on how these two entries are interpreted.

psiDBNameLength
Address of a short integer which is updated with the length of the database name found in the option string. A zero length indicates that no database name was found.

ppszDBName
Address of a pointer to character storage. This pointer will be updated by the API to indicate dynamic storage containing the NULL-terminated database name. If the *psiDBNameLength* parameter has a non-zero value, the database name can be extracted from the buffer indicated by the address returned through *ppszDBName*.

psiUseridLength
Address of a short integer, which is updated with the length of the user ID found in the option string. A zero length indicates that no user ID was found.

ppszUserid
Address of a pointer to character storage. This pointer will be updated by the API to indicate dynamic storage containing the NULL-terminated user ID. If the *psiUseridLength* parameter has a non-zero value, the user ID can be extracted from the buffer indicated by the address returned through *ppszUserid*.

psiPasswordLength
Address of a short integer, which is updated with the length of the password found in the option string. A zero length indicates that no password was found.

ppszPassword
Address of a pointer to character storage. This pointer will be updated by the API to indicate dynamic storage containing the NULL-terminated password. If the *psiPasswordLength* parameter has a non-zero value, the password can be extracted from the buffer indicated by the address returned through *ppszPassword*.

psiMsgFileLength
Address of a short integer, which is updated with the length of the message file name found in the option string. A zero length indicates that no message file name was found.

ppszMsgFile
Address of a pointer to character storage. This pointer will be updated by the API to indicate dynamic storage containing the NULL-terminated

message file name. If the *psiMsgFileLength* parameter has a non-zero value, the message file name can be extracted from the buffer indicated by the address returned through *ppszMsgFile*.

psiPackageLength
Address of a short integer, which is updated with the length of the package name found in the option string. A zero length indicates that no package name was found.

ppszPackage
Address of a pointer to character storage. This pointer will be updated by the API to indicate dynamic storage containing the NULL-terminated package name. If the *psiPackageLength* parameter has a non-zero value, the package name can be extracted from the buffer indicated by the address returned through *ppszPackage*.

psiBindFileLength
Address of a short integer, which is updated with the length of the bind file name found in the option string. A zero length indicates that no bind file name was found.

ppszBindFile
Address of a pointer to character storage. This pointer will be updated by the API to indicate dynamic storage containing the NULL-terminated bind file name. If the *psiBindFileLength* parameter has a non-zero value, the bind file name can be extracted from the buffer indicated by the address returned through *ppszBindFile*.

lTarget
A long integer indicating the target API for which the option string should be parsed. Currently, the only supported target API is **db2Initialize**; therefore, this parameter can be set to `SQLAO_PREP_SVCS_API` (`0`) or `SQLAO_BIND_API` (`2`).

pvMemList
A pointer to 4 bytes of memory, in which **sqlaoptions** will store a pointer to storage allocated during option parsing. This storage should be freed by a call to **sqlaoptions_free** or **sqlgoptions_free** after the target API has been called.

pSqlca
A pointer to an SQLCA structure containing the return status of the API call.

Usage Notes:

Upon successful return from **sqlaoptions**, the caller should test the first two entries of the options array.

If `SQLAO_PREP_SVCS_API` has been specified, the following will be populated:

- The first entry indicates whether a package is to be created. If the option value is `SQLA_CREATE_PLAN` (`1`), the caller should verify that *pscPackage* contains a package name. If it does not, no package name was present in the option string, and the caller must provide a default name to **db2Initialize**.
- Similarly, the second entry indicates whether a bind file was requested. If the option value is `SQLA_CREATE_BIND_FILE` (`1`), the caller should verify that *pscBindFile* contains a bind file name. If it does not, no bind file name was present in the option string, and the caller must provide a default name to **db2Initialize**.

Note: These two options should be checked on any successful return, even if the input string was empty, or contained only other options.

Return Codes:

This API returns one of the following messages. Check the SQLCODE field in the SQLCA.

0 Successful execution.

-7 Invalid character.

-10 Unterminated string.

-83 Insufficient storage.

-104 Syntax error.

sqlaoptions_free - Free Option Parser Storage

This API releases any storage allocated by **sqlaoptions**. The **sqlaoptions** API allocates storage to hold any or all of: database name, user ID, password, package name, bind file name, or any string option, such as collection name. The **sqlaoptions_free** API should be called after each call to **sqlaoptions**.

API Include File:

sqlaprep.h

C API Syntax:

```
SQL_API_RC SQL_API_FN
      sqlaoptions_free( void *mem_list,
                        struct sqlca *sqlca )
```

Generic API Syntax:

```
SQL_API_RC SQL_API_FN
      sqlgoptions_free( void *mem_list,
                        struct sqlca *sqlca )
```

API Parameters:

mem_list
The value of the 4 bytes of memory updated by **sqlaoptions** through *pvMemList*. Note the one fewer levels of indirection here in comparison to **sqlaoptions**: *pvMemList* is updated by **sqlaoptions**, so it is passed by reference. In **sqlaoptions_free**, it is passed by value.

sqlca A pointer to an SQLCA structure containing the return status of the API call.

Return Codes:

This API returns one of the following messages. Check the SQLCODE field in the SQLCA.

0 Successful execution.

-83 Memory error.

Chapter 7. Precompiler Services APIs

The precompiler calls the following Precompiler Services APIs to analyze SQL statements:

sqlaalhv
Records a host variable.

db2CompileSql
Compiles an SQL statement and places it into a section in the package.

db2Initialize
Initializes the precompilation process.

sqlafini
Terminates the precompilation process.

Generic versions of these APIs are also provided for writing precompilers in host languages other than C.

SQLCA and return codes

All Precompiler Services and Runtime Services functions return one of the following 2-byte status codes:

SQLA_CHECK_SQLCA (0)
Check the SQLCA for the function call completion code.

SQLA_SQLCA_BAD (-1)
The SQLCA address passed as input is not valid. The command was not processed.

Precompiler Services and the database manager both return status codes in the SQLCA. Test the SQLCA after every Precompiler Services function. The SQLCA structure is consistent with other call interfaces to the database manager and with the SQL language itself.

For detailed information about the SQLCA, see the *SQL Reference*.

sqlaalhv - Add Host Variable

Use a call to this API to register a host variable with Precompiler Services. The precompiler detects a host variable, determines its type and length, and then assigns it a unique token ID. Precompiler Services uses this information to process SQL statements that reference host variables.

API Include File:

sqlaprep.h

C API Syntax:

```
SQL_API_RC  SQL_API_FN
     sqlaalhv   (unsigned short  *name_length,
                 char            *name,
                 unsigned short  *sqltype,
                 unsigned long   *sql_length,
                 unsigned long   *token_id,
                 unsigned short  *location,
                 void            *udtname,
                 struct sqlca    *sqlca);
```

Generic API Syntax:

```
SQL_API_RC  SQL_API_FN
     sqlgalhv   (unsigned short  *name_length,
                 char            *name,
                 unsigned short  *sqltype,
                 unsigned long   *sql_length,
                 unsigned long   *token_id,
                 unsigned short  *location,
                 void            *udtname,
                 struct sqlca    *sqlca);
```

API Parameters:

name_length
Pointer to the length of the host variable name.

name Pointer to the host variable name.

The name must be composed of characters taken from the database manager's extended character set. Some host languages allow characters that are not in the extended set. The precompiler should replace those with valid characters before they are sent to Precompiler Services.

sqltype
Pointer to the SQL data type of the host variable.

sql_length
Pointer to the length of the host variable.

token_id
Pointer to the 4-byte token ID of the host variable.

location
Pointer to the location value of the host variable. Possible values are:

- SQLA_DECLARE_SECT (0) — Host variable was found in declare section.
- SQLA_SQL_STMT (1) — Host variable without a token ID was found in an SQL statement. This is returned when a statement contains an SQLDA reference. SQLDA names are not declared within declare sections.

udtname
A pointer to the structured type name. It can also be set to NULL or to point to zero.

sqlca A pointer to an SQLCA structure containing the return status of the API call.

Return Codes:

This API returns one of the following messages. Check the SQLCODE field in the SQLCA.

0 Successful execution.

-83 Insufficient storage.

-307 Host variable already declared.

-308 Maximum number of host variables exceeded.

-4901 Reinitialization has not occurred since last fatal error.

-4902 Invalid characters in parameter.

-4903 Invalid parameter length.

-4904 Invalid pointer to parameter.

-4905 Parameter not within valid range.

-4911 SQL type for host variable is invalid.

-4912 Length of host variable is out of range.

-4913 Token ID has already been used.

-4914 Invalid token ID.

-4916 **db2Initialize** has not been invoked.

-4994 Interrupt key sequence detected.

-4999 Database manager error.

db2CompileSql - Compile SQL Statement

Compiles an SQL statement. This API parses the statement, assigns a section number, and possibly stores the statement in the bind file. It completes the task array, the token array, and any other required output parameters.

API Include File:

db2ApiDf.h

C API Syntax:

```
SQL_API_RC SQL_API_FN
  db2CompileSql (
   db2Uint32 versionNumber,
   void * pParmStruct,
   struct sqlca * pSqlca);

typedef struct db2CompileSqlStruct
{
   db2Uint32            *piSqlStmtLen;
   char                 *piSqlStmt;
   db2Uint32            *piLineNum;
   struct sqla_flaginfo *pioFlagInfo;
   struct sqla_tokens   *pioTokenIdArray;
   struct sqla_tasks    *poTaskArray;
   db2Uint16            *poSectionNum;
   db2Uint16            *poSqlStmtType;
   char                 *poBuffer1;
   char                 *poBuffer2;
   char                 *poBuffer3;
   void                 *pioReserved;
} db2CompileSqlStruct;
```

Generic API Syntax:

```
SQL_API_RC SQL_API_FN
  db2gCompileSql (
   db2Uint32 versionNumber,
   void * pParmStruct,
   struct sqlca * pSqlca);

typedef struct db2gCompileSqlStruct
{
   db2Uint32            *piSqlStmtLen;
   char                 *piSqlStmt;
   db2Uint32            *piLineNum;
   struct sqla_flaginfo *pioFlagInfo;
   struct sqla_tokens   *pioTokenIdArray;
   struct sqla_tasks    *poTaskArray;
   db2Uint16            *poSectionNum;
   db2Uint16            *poSqlStmtType;
   char                 *poBuffer1;
   char                 *poBuffer2;
   char                 *poBuffer3;
   void                 *pioReserved;
} db2gCompileSqlStruct;
```

API Parameters:

versionNumber
Input. Specifies the version and release level of the structure passed in as the second parameter, *pParmStruct*.

pParmStruct
Input. A pointer to the *db2CompileSqlStruct* structure.

pSqlca
A pointer to an SQLCA structure containing the return status of the API call.

piSqlStmtLen
Pointer to the text length (in bytes) of the SQL statement.

piSqlStmt
Pointer to the preprocessed SQL statement text.

The statement buffer must have one more trailing byte than is required for the statement text. This byte is not included in the length parameter. The contents of the statement buffer may change during the call to **db2CompileSql**.

piLineNum
Pointer to the source file line number where the SQL statement begins. If the application is being precompiled against a DRDA server such as DB2 for OS/390, DB2 for AS400, or DB2 for VM/VSE, this line number must be greater than or equal to 1.

pioFlagInfo
Pointer to an instance of the *sqla_flaginfo* structure. This is only required if SQL flagging is requested in the call to **db2Initialize**. If flagging is not desired, set the pointer to NULL.

pioTokenIdArray
Pointer to the start of the token identifier array.

poTaskArray
Pointer to the start of the task array.

poSectionNum
Pointer to the package section number assigned to this SQL statement. If the statement does not require a section number, Precompiler Services returns zero to this address.

poSqlStmtType
Pointer to the SQL statement type.

If an SQL CONNECT or SQL CALL statement is being processed, the value returned to this address should be passed to **sqlacall** in place of the section number. Valid values are defined in the **sqlaprep** include file (`sql.h`).

poBuffer1
Pointer to a 256-byte character buffer used to store string data. The use of this buffer (and the following two buffers) depends on the type of statement that was precompiled. Current use is for WHENEVER processing only.

poBuffer2
Pointer to a 256-byte character buffer used to store string data. Current use is for WHENEVER processing only.

poBuffer3
Pointer to a 256-byte character buffer used to store string data. Current use is for WHENEVER processing only.

pioReserved
Spare pointer, set to NULL or to point to zero.

Return Codes:

This API returns one of the following messages. Check the SQLCODE field in the SQLCA.

0 Successful execution.

143 SQL statement is not supported; syntax ignored.

513 SQL statement will modify an entire table.

4943 Number of host variables does not match number of items in SELECT clause.

-7 Invalid character in SQL statement.

-10 String begun but not terminated.

-32 Cannot access disk or file.

-51 Maximum number of package sections exceeded.

-83 Insufficient storage.

-85 Duplicate statement name declared.

-87 Invalid use of multi-member structure host variable.

-100 Invalid numeric literal.

-101 Statement too long or too complex.

-104 Incorrect statement syntax.

-107 Name of database object too long.

-108 Name has improper number of qualifiers.

-142 SQL statement is not supported.

-199 Invalid use of an SQL reserved word.

-306 Host variable has not been declared.

-310 Number of host variables in statement exceeds limit.

-324 Host variable may not be used in this context.

-505 Duplicate cursor name.

-751 Invalid trigger statement.

-968 File system full.

-4010 Recursive compound SQL is invalid.

-4011 Illegal statement type in compound SQL block.

-4012 Only one COMMIT allowed per compound SQL block.

-4013 Not in compound SQL block, so END COMPOUND is invalid.

-4901 Reinitialization has not occurred since last fatal error.

-4902 Invalid characters in parameter.

-4903 Invalid parameter length.

-4904 Invalid pointer to parameter.

-4905 Parameter not within valid range.

-4916 **db2Initialize** has not been invoked.

-4919 Task array too small.

-4920 Token ID array too small.

-4940 Illegal clause in statement.

-4941 Blank or empty SQL statement text.

-4942 Incompatible data type selected into host variable.

-4944 Attempt to store a NULL value in a NOT NULL column.

-4945 Invalid use of a parameter marker.

-4946 Cursor not declared.

-4994 Interrupt key sequence detected.

-4998 Database connection has been lost.

-4999 Internal error.

db2Initialize - Initialize Precompiler Services

This call initializes the Precompiler Services data structures, opens a bind file if necessary, and calls the DB2 kernel to initialize the package in the database. No other Precompiler Services calls are valid until **db2Initialize** has been successfully completed.

API Include File:

db2ApiDf.h

C API Syntax:

```
SQL_API_RC SQL_API_FN
  db2Initialize (
   db2Uint32 versionNumber,
   void * pParmStruct,
   struct sqlca * pSqlca);

typedef struct db2InitStruct
{
   db2Uint16       *piProgramNameLength;
   char            *piProgramName;
   db2Uint16       *piDbNameLength;
   char            *piDbName;
   db2Uint16       *piDbPasswordLength;
   char            *piDbPassword;
   db2Uint16       *piBindNameLength;
   char            *piBindName;
   struct sqla_options *piOptionsArray;
   db2Uint16       *piPidLength;

   struct sqla_program_id *poPrecompilerPid;
} db2InitStruct;
```

Generic API Syntax:

```
SQL_API_RC SQL_API_FN
  db2gInitialize (
   db2Uint32 versionNumber,
   void * pParmStruct,
   struct sqlca * pSqlca);

typedef struct db2gInitStruct
{
   db2Uint16       *piProgramNameLength;
   char            *piProgramName;
   db2Uint16       *piDbNameLength;
   char            *piDbName;
   db2Uint16       *piDbPasswordLength;
   char            *piDbPassword;
   db2Uint16       *piBindNameLength;
   char            *piBindName;
   struct sqla_options *piOptionsArray;
   db2Uint16       *piPidLength;

   struct sqla_program_id *poPrecompilerPid;
} db2gInitStruct;
```

API Parameters:

versionNumber
Input. Specifies the version and release level of the structure passed in as the second parameter, *pParmStruct*. The versionNumber allows the Precompiler Services to emulate the specific version level behaviour such as maping of SGL data types and sizes.

pParmStruct
Input. A pointer to the *db2InitStruct* structure.

pSqlca
A pointer to an SQLCA structure containing the return status of the API call.

piProgramNameLength
A pointer to a 2-byte unsigned integer representing the length of the program name.

piProgramName
The package name. If the application is being precompiled against a DRDA server such as DB2 for OS/390, DB2/400, or DB2 for VM/VSE, *program_name* should consist only of the following characters: A-Z, 0-9, and _.

spare1-4
Spare pointers, set to NULL or to point to zero.

piBindNameLength
A pointer to a 2-byte unsigned integer representing the length of the bind file name.

piBindName
The name of the bind file.

piOptionsArray
Pointer to the start of the precompiler option array.

poPrecompilerPid
Pointer to the start of the precompiler program ID.

piPidLength
Precompiler program ID buffer length.

Return Codes:

This API returns one of the following messages. Check the SQLCODE field in the SQLCA.

0 Successful execution.

20 Precompile options ignored.

-31 Cannot open disk or file.

-32 Cannot access disk or file.

-83 Insufficient storage.

-968 File system full.

-1024 No database connection exists.

-4902 Invalid characters in parameter.

-4903 Invalid parameter length.

-4904 Invalid pointer to parameter.

-4915 **db2Initialize** has already been invoked.

-4917 Unsupported option found in option array.

-4930 Invalid option found in option array.

-4994 Interrupt key sequence detected.

-4995 Cannot find COUNTRY.SYS.

-4997 Invalid authorization ID.

-4998 Database connection has been lost.

-4999 Precompiler Services error.

sqlafini - Terminate Precompiler Services

Terminates Precompiler Services. It is the final call to Precompiler Services from the precompiler.

Once this call has been issued, all other calls to Precompiler Services are rejected unless a new initialization call is successfully completed.

API Include File:

sqlaprep.h

C API Syntax:

```
SQL_API_RC  SQL_API_FN
     sqlafini   (unsigned short  *term_option,
                 void            *reserved,
                 struct sqlca    *sqlca);
```

Generic API Syntax:

```
SQL_API_RC  SQL_API_FN
     sqlgfini   (unsigned short  *term_option,
                 void            *reserved,
                 struct sqlca    *sqlca);
```

API Parameters:

term_option
Pointer to a 2-byte integer that indicates whether the package or bind file being created should be saved or discarded. Possible values are:

- SQLA_SAVE — Save the package or bind file.
- SQLA_DISCARD — Discard the package or bind file.

reserved
Spare pointer, set to NULL or to point to zero.

sqlca A pointer to an SQLCA structure containing the return status of the API call.

This function returns completion codes in SQLCODE. It also uses the SQLWARN6 and SQLWARN7 fields of the SQLCA as follows:

- SQLWARN6 — If set to 1, the package was successfully saved. Otherwise, it was discarded.
- SQLWARN7 — If set to 1, the bind file was successfully saved. Otherwise, it was deleted, or could not be successfully saved.

These warning fields are always set, regardless of the termination option. This allows the precompiler to determine whether either of the objects was saved if an error or interrupt occurs while **sqlafini** is being executed.

Return Codes:

This API returns one of the following messages. Check the SQLCODE field in the SQLCA.

0 Successful execution.

-32 Cannot access disk or file.

-83 Insufficient storage.

-968 File system full.

-4903 Invalid parameter length.

-4904 Invalid pointer to parameter.

-4916 **db2Initialize** has not been invoked.

-4918 Invalid termination option.

-4994 Interrupt key sequence detected.

-4998 Database connection has been lost.

-4999 Internal error.

Chapter 8. Runtime Services APIs

The application calls the following Runtime Services APIs to use the database manager:

sqlaaloc
Allocates a dynamic run time SQLDA.

sqlacall
Calls DB2 to process a package section or function.

sqlacmpd
Registers a compound SQL substatement for execution.

sqladloc
Deallocates a dynamic run time SQLDA.

sqlastls
Passes a statement string to be processed dynamically.

sqlastlv
Adds a variable to a dynamic SQLDA.

sqlastlva
Adds a structured type variable to a dynamic SQLDA.

sqlasetdata
Adds a group of variables to a dynamic SQLDA.

sqlastop
Terminates a sequence of calls to Runtime Services.

sqlastrt
Begins a sequence of calls to Runtime Services.

sqlausda
Records a pointer to a user-defined SQLDA.

Generic versions of these APIs are also provided for writing precompilers in host languages other than C.

Treat all Runtime Services APIs between **sqlastrt** and **sqlastop** as a unit. The generated run time code does not usually check SQLCA contents until final error processing. If an error occurs in one API, the error is passed along through the other APIs.

The **sqlaaloc** API is an exception, in which the SQLCODE of the SQLCA reports whether the SQLVAR fields of the allocated SQLDA need to be initialized.

sqlaaloc - Allocate SQLDA

Allocates an internal SQLDA. The SQLDA is not seen directly by the application programmer. Runtime Services uses this SQLDA to send host variable information to the database manager.

There are techniques to optimize usage of internal SQLDAs. For details, see "Optimizing function calls" on page 40.

API Include File:

sqlaprep.h

sqladef.h

C API Syntax:

```
SQL_API_RC  SQL_API_FN
     sqlaaloc   (unsigned short  sqlda_id,
                 unsigned short  sqlvar_num,
                 unsigned short  stmt_id,
                 void           *reserved);
```

Generic API Syntax:

```
SQL_API_RC  SQL_API_FN
     sqlgaloc   (unsigned short  sqlda_id,
                 unsigned short  sqlvar_num,
                 unsigned short  stmt_id,
                 void           *reserved);
```

API Parameters:

sqlda_id
A 2-byte SQLDA identifier that cannot be zero.

If the ID matches an existing dynamic SQLDA, the existing SQLDA is used; a new SQLDA is not allocated.

sqlvar_num
A 2-byte integer representing the number of SQLVAR elements to allocate in this dynamic SQLDA.

stmt_id
A 2-byte unique identifier of the current statement.

reserved
Spare pointer, set to NULL or to point to zero.

Return Codes:

This API returns one of the following messages. Check the SQLCODE field in the SQLCA.

0 SQLVAR elements must be initialized; call **sqlastlv**.

4959 No SQLVAR elements need to be initialized.

-4905 Parameter not within valid range.

-4951 Invalid *sqlda_id*.

-4999 Internal error.

sqladloc - Deallocate SQLDA

Deallocates an internal SQLDA previously allocated by **sqlaaloc**. It also removes the *sqlda_id* from the list of SQLDA IDs recognized by **sqlacall**.

API Include File:

sqlaprep.h

sqladef.h

C API Syntax:

```
SQL_API_RC  SQL_API_FN
     sqladloc   (unsigned short  sqlda_id,
                 void           *reserved);
```

Generic API Syntax:

```
SQL_API_RC  SQL_API_FN
     sqlgdloc   (unsigned short  sqlda_id,
                 void           *reserved);
```

API Parameters:

sqlda_id
A 2-byte identifier for the SQLDA to be deallocated.

reserved
Spare pointer, set to NULL or to point to zero.

Return Codes:

This API returns one of the following messages. Check the SQLCODE field in the SQLCA.

0 Successful execution.

-4951 Invalid *sqlda_id*.

-4999 Internal error.

sqlacall - Execute SQL Statement

Calls the database manager to execute an SQL statement. This is the only Runtime Services API that communicates with the database manager.

All execution parameters are set up prior to, or provided with, this function call. All validation of host variables is performed by this API.

API Include File:

sqlaprep.h

sqladef.h

C API Syntax:

```
SQL_API_RC  SQL_API_FN
     sqlacall   (unsigned short  call_type,
                 unsigned short  section_number,
                 unsigned short  input_sqlda,
                 unsigned short  output_sqlda,
                 void           *reserved);
```

Generic API Syntax:

```
SQL_API_RC  SQL_API_FN
     sqlgcall   (unsigned short  call_type,
                 unsigned short  section_number,
                 unsigned short  input_sqlda,
                 unsigned short  output_sqlda,
                 void           *reserved);
```

API Parameters:

call_type
A 2-byte integer identifying the type of call being made.

section_number
A 2-byte integer denoting the section of the package to be processed. This may be zero for statements that do not use section numbers, such as COMMIT and ROLLBACK. For the SQL CONNECT and CALL statements, this parameter is used to pass information about the statement type.

input_sqlda
A 2-byte identifier for the input SQLDA.

If the statement does not use an input SQLDA, set this parameter to zero.

output_sqlda
A 2-byte identifier for the output SQLDA.

If the statement does not use an output SQLDA, set this parameter to zero.

reserved
Spare pointer, set to NULL or to point to zero.

Return Codes:

This API returns one of the following messages. Check the SQLCODE field in the SQLCA.

0 Successful execution.

-311 Length of host variable is negative.

-804 Invalid *sqlda* contents (incorrect *sqltype*, *sqllen*, or other).

-822 Invalid address in *sqlda*.

-1216 Invalid use of graphic data.

-4951 Invalid *sqlda_id*.

-4953 Invalid *call_type*.

-4954 Section number is invalid.

-4999 Internal error.

sqlacmpd - Register Compound SQL Substatement

Calls the database manager to add a compound SQL substatement to the current list of substatements to be executed. The substatement will not actually be executed until the next call to **sqlacall**.

API Include File:

sqlaprep.h

sqladef.h

C API Syntax:

```
SQL_API_RC  SQL_API_FN
     sqlacmpd   (unsigned short  call_type,
                 unsigned short  section_number,
                 unsigned short  input_sqlda,
                 unsigned short  output_sqlda,
                 void           *reserved);
```

Generic API Syntax:

```
SQL_API_RC  SQL_API_FN
     sqlgcmpd   (unsigned short  call_type,
                 unsigned short  section_number,
                 unsigned short  input_sqlda,
                 unsigned short  output_sqlda,
                 void           *reserved);
```

API Parameters:

call_type
A 2-byte integer identifying the type of call to be made when the substatement is executed.

section_number
A 2-byte integer denoting the section of the package to be processed.

input_sqlda
A 2-byte identifier for the input SQLDA.

If the statement does not use an input SQLDA, set this parameter to zero.

output_sqlda
A 2-byte identifier for the output SQLDA.

If the statement does not use an output SQLDA, set this parameter to zero.

reserved
Spare pointer, set to NULL or to point to zero.

Return Codes:

This API returns one of the following messages. Check the SQLCODE field in the SQLCA.

0 Successful execution.

-83 Insufficient storage.

-4011 Invalid compound SQL substatement.

-4012 Invalid COMMIT within compound SQL.

-4951 Invalid *sqlda_id*.

-4953 Invalid *call_type*.

-4954 Section number is invalid.

-4999 Internal error.

sqlastlv - Record Host Variable Address

Use **sqlastlv** after the **sqlaaloc** allocation call to initialize the fields of an SQLDA SQLVAR element to the type, length, and address of a host variable that refers to a structured type found in an SQL statement.

API Include File:

sqlaprep.h

sqladef.h

C API Syntax:

```
SQL_API_RC  SQL_API_FN
     sqlastlv   (unsigned short  sqlda_id,
                 unsigned short  sqlvar_index,
                 unsigned short  sqltype,
                 unsigned long   var_length,
                 void           *host_var,
                 short          *ind_var,
                 void           *reserved);
```

Generic API Syntax:

```
SQL_API_RC  SQL_API_FN
     sqlgstlv   (unsigned short  sqlda_id,
                 unsigned short  sqlvar_index,
                 unsigned short  sqltype,
                 unsigned long   var_length,
                 void           *host_var,
                 short          *ind_var,
                 void           *reserved);
```

API Parameters:

sqlda_id
A 2-byte integer containing the ID of the SQLDA to be initialized. It must match an SQLDA ID passed to **sqlaaloc**.

sqlvar_index
A 2-byte integer containing the index of the SQLVAR element in the SQLDA to be initialized.

sqltype
A 2-byte integer containing the SQL data type of the host variable or literal. If the host variable uses an indicator variable, the data type must be odd; otherwise, without an indicator variable, it is even. This parameter is not verified when the API is called.

var_length
A 4-byte integer containing the declared length of the host variable or literal. This parameter is not verified when the API is called.

host_var
Address of the host variable or literal.

ind_var
Address of the indicator variable, if one was used with the host variable; otherwise NULL.

reserved
Spare pointer, set to NULL or to point to zero.

Return Codes:

This API returns one of the following messages. Check the SQLCODE field in the SQLCA.

0 Successful execution.

-4911 SQL type for host variable is invalid.

-4912 Length of host variable is out of range.

-4951 Invalid *sqlda_id*.

-4952 Invalid *sqlvar_index* parameter.

-4999 Internal error.

sqlastlva - Record Host Variable Address

Use **sqlastlva** after the **sqlaaloc** allocation call to initialize the fields of an SQLDA SQLVAR element to the type, length, and address of a host variable that refers to a structured type found in an SQL statement.

Note: Support for this function was added for DB2 Universal Database, Version 7 FixPak 1.

API Include File:

sqlaprep.h

sqladef.h

C API Syntax:

```
SQL_API_RC  SQL_API_FN
     sqlastlva  (unsigned short  sqlda_id,
                 unsigned short  sqlvar_index,
                 unsigned short  sqltype,
                 unsigned long   var_length,
                 void           *host_var,
                 sqluint32       sqldslen,
                 void           *sqldsname,
                 short          *ind_var,
                 void           *reserved);
```

Generic API Syntax:

```
SQL_API_RC  SQL_API_FN
     sqlgstlv   (unsigned short  sqlda_id,
                 unsigned short  sqlvar_index,
                 unsigned short  sqltype,
                 unsigned long   var_length,
                 void           *host_var,
                 sqluint32       sqldslen,
                 void           *sqldsname,
                 short          *ind_var,
                 void           *reserved);
```

API Parameters:

sqlda_id
A 2-byte integer containing the ID of the SQLDA to be initialized. It must match an SQLDA ID passed to **sqlaaloc**.

sqlvar_index
A 2-byte integer containing the index of the SQLVAR element in the SQLDA to be initialized.

sqltype
A 2-byte integer containing the SQL data type of the host variable or literal. If the host variable uses an indicator variable, the data type must be odd; otherwise, without an indicator variable, it is even. This parameter is not verified when the API is called.

var_length
A 4-byte integer containing the declared length of the host variable or literal. This parameter is not verified when the API is called.

host_var
Address of the host variable or literal.

sqldslen
A 4-byte integer containing the length of the structured type name.

sqldsname
Address of the structured type name.

ind_var
Address of the indicator variable, if one was used with the host variable; otherwise NULL.

reserved
Spare pointer, set to NULL or to point to zero.

Return Codes:

This API returns one of the following messages. Check the SQLCODE field in the SQLCA.

0 Successful execution.

-4911 SQL type for host variable is invalid.

-4912 Length of host variable is out of range.

-4951 Invalid *sqlda_id*.

-4952 Invalid *sqlvar_index* parameter.

-4999 Internal error.

sqlasetdata - Record Multiple Host Variable Addresses

Use **sqlasetdata** after the **sqlaaloc** allocation call to initialize the fields of several SQLDA SQLVAR elements to the types, lengths, and addresses of multiple host variables or literals found in an SQL statement. Similar to **sqlastlv**, except that it can process several host variables at once.

API Include File:

sqladef.h

C API Syntax:

```
SQL_API_RC  SQL_API_FN
     sqlasetdata (unsigned short             sqlda_id,
                  unsigned short             start_index,
                  unsigned short             elements,
                  struct sqla_setdata_list  *setdlist,
                  struct sqla_setds_list    *setdslist,
                  void                      *spare);
```

Generic API Syntax:

Not currently available. Use **sqlastlv** or **sqlastlva** instead.

API Parameters:

sqlda_id
A 2-byte integer containing the ID of the SQLDA to be initialized. It must match an SQLDA ID passed to **sqlaaloc**.

start_index
A 2-byte integer containing the starting index of the SQLVAR elements in the SQLDA to be initialized.

elements
A 2-byte integer containing the number of SQLVARs to be initialized.

setdlist
Pointer to an array of *sqla_setdata_list* structures containing host variable information to be stored in the SQLDA.

setdslist
Pointer to an array of *sqla_setds_list* structures containing structured type host variable information to be stored in the SQLDA.

spare Spare pointer, set to NULL or to point to zero.

Return Codes:

This API returns one of the following messages. Check the SQLCODE field in the SQLCA.

0 Successful execution.

-4911 SQL type for host variable is invalid.

-4912 Length of host variable is out of range.

-4951 Invalid *sqlda_id*.

-4952 Invalid *sqlvar_index* parameter.

-4999 Internal error.

sqlastls - Record SQL Statement Text

Generate this call when an SQL statement contains the name of a host variable used to store statement text, as in the dynamic SQL statements PREPARE and EXECUTE IMMEDIATE. This call sends the length and the address of a host variable containing the stored statement text to Runtime Services.

It is the responsibility of the calling program to determine the string length at execution time, unless the string is contained within a host variable of data type

460 (NULL-terminated string). In this case, a zero can be passed as the string length, and the API will calculate the length.

API Include File:

sqlaprep.h

sqladef.h

C API Syntax:

```
SQL_API_RC  SQL_API_FN
     sqlastls     (unsigned long   stmt_length,
                   const void     *stmt_text,
                   void           *reserved);
```

Generic API Syntax:

```
SQL_API_RC  SQL_API_FN
     sqlgstls     (unsigned long   stmt_length,
                   const void     *stmt_text,
                   void           *reserved);
```

API Parameters:

stmt_length
A 4-byte integer containing the length of the SQL statement, in bytes.

stmt_text
Address of an array of characters containing the SQL statement text.

reserved
Spare pointer, set to NULL or to point to zero.

Return Codes:

This API returns one of the following messages. Check the SQLCODE field in the SQLCA.

0 Successful execution.

-101 SQL statement is too long.

-4904 Invalid pointer to parameter.

-4905 Parameter not within valid range.

-4999 Internal error.

sqlausda - Register SQLDA

Stores the address of a user-specified input or output SQLDA in Runtime Services. It generates an SQLDA ID for this structure. Calling **sqlacall** with the generated SQLDA ID tells Runtime Services to use this user-defined SQLDA for the SQL statement.

API Include File:

sqlaprep.h

sqladef.h

C API Syntax:

```
SQL_API_RC  SQL_API_FN
     sqlausda     (unsigned short  sqlda_id,
                   struct sqlda   *sqlda,
                   void           *reserved);
```

Generic API Syntax:

```
SQL_API_RC  SQL_API_FN
     sqlgusda     (unsigned short  sqlda_id,
                   struct sqlda   *sqlda,
                   void           *reserved);
```

API Parameters:

sqlda_id
A 2-byte integer containing a unique identifier for the SQLDA data structure. This ID is passed in a subsequent call to **sqlacall**.

sqlda Pointer to a user-defined SQLDA data structure.

reserved
Spare pointer, set to NULL or to point to zero.

Return Codes:

This API returns one of the following messages. Check the SQLCODE field in the SQLCA.

0 Successful execution.

-83 Insufficient storage.

-4904 Invalid pointer to parameter.

-4951 Invalid *sqlda_id*.

-4999 Internal error.

sqlastrt - Start Serialized Execution

Initializes the SQLCA, and registers the address of the program ID that identifies the access plan.

API Include File:

sqlaprep.h

sqladef.h

C API Syntax:

```
SQL_API_RC  SQL_API_FN
     sqlastrt   (char                       runtime_pid[40],
                 struct sqla_runtime_info *sqla_rtinfo,
                 struct sqlca             *sqlca);
```

Generic API Syntax:

```
SQL_API_RC  SQL_API_FN
     sqlgstrt   (char                      runtime_pid[40],
                 struct sqla_runtime_info *sqla_rtinfo,
                 struct sqlca             *sqlca);
```

API Parameters:

runtime_pid
Address of the run time program ID. This ID is registered in the Runtime Services internal control block. It is returned from the **db2Initialize** function at precompile time.

sqla_rtinfo
Pointer to an instance of the *sqla_runtime_info* structure. This structure currently indicates only the program's preferred method of handling the C wchar_t data type. For non-C applications, this pointer should be set to NULL. See "Runtime information structure" on page 49.

sqlca A pointer to an SQLCA structure containing the return status of the API call.

Return Codes:

This API returns one of the following messages. Check the SQLCODE field in the SQLCA.

0 Successful execution.

-83 Insufficient storage.

-4903 Invalid parameter length.

-4904 Invalid pointer to parameter.

-4999 Internal error.

sqlastop - Stop Serialized Execution

Terminates processing of the SQL statement.

API Include File:

sqlaprep.h

sqladef.h

C API Syntax:

```
SQL_API_RC  SQL_API_FN
     sqlastop   (void *reserved);
```

Generic API Syntax:

```
SQL_API_RC  SQL_API_FN
     sqlgstop   (void  *reserved);
```

API Parameters:

reserved
Spare pointer, set to NULL or to point to zero.

Return Codes:

This API returns one of the following messages. Check the SQLCODE field in the SQLCA.

0 Successful execution.

-4999 Internal error.

sqlastop - Stop Serialized Execution

Chapter 9. Error messages and codes

Table 4 lists the error code values returned from Precompiler Services and Runtime Services. It also lists the database manager message code, the associated constant, and a brief description. The `sqlaprep.h` include file separates these codes into standard error codes, and those that are considered fatal. Another **db2Initialize** must be issued before you can continue precompiling after a fatal error.

Table 4. Error Messages and Codes

Value	Code	Constant	Fatal?	Description
0	SQL0000N	SQLA_RC_OK	N	Successful execution.
20	SQL0020W	SQLA_RC_OPTION_IGNORED	N	Precompiled option ignored.
143	SQL0143W	SQLA_RC_DDSIGN	N	SQL statement is not supported; invalid syntax ignored.
513	SQL0513W	SQLA_RC_STMT_MODIFY_ALL	N	Statement modifies entire table.
4943	SQL4943W	SQLA_RC_SELECT_LIST_BAD	N	Number of host variables does not match number of items in SELECT clause.
4959	SQL4959W	SQLA_RC_SQLVARS_SET	N	SQLVARS already initialized.
-7	SQL0007N	SQLA_RC_CHAR_BAD	N	Invalid character in SQL statement.
-10	SQL0010N	SQLA_RC_STRING_NOT_TERMINATED	N	String begun but not terminated.
-13	SQL0013N	SQLA_RC_EMPTY_DEL_IDENT	N	An empty string delimiter is not valid.
-31	SQL0031N	SQLA_RC_BFILE_OPEN_ERROR	Y	Cannot open disk or file.
-32	SQL0032N	SQLA_RC_BFILE_DISK_ERROR	Y	Cannot access disk or file.
-51	SQL0051N	SQLA_RC_SECTION_LIMIT	N	Too many sections.
-83	SQL0083N	SQLA_RC_MEMORY_BAD	Y	Insufficient storage.
-85	SQL0085N	SQLA_RC_SNAME_DUP	N	Duplicate statement name.
-87	SQL0087N	SQLA_RC_NO_STRUCT	N	Invalid use of host structure.
-88	SQL0088N	SQLA_RC_AMBIG_HOSTVAR	N	Ambiguous reference to host structure field.
-101	SQL0101N	SQLA_RC_STMT_LIMIT	N	Statement too long or too complex.
-103	SQL0103N	SQLA_RC_NUMBER_BAD	N	Invalid numeric literal.
-104	SQL0104N	SQLA_RC_STMT_SYNTAX_BAD	N	Incorrect statement syntax.
-105	SQL0105N	SQLA_RC_GSTRING_BAD	N	Invalid graphic string (DBCS environment only).
-107	SQL0107N	SQLA_RC_IDENTIFIER_LIMIT	N	Name of database object is too long.

Table 4. Error Messages and Codes (continued)

Value	Code	Constant	Fatal?	Description
-108	SQL0108N	SQLA_RC_QUALIFIER_BAD	N	Name has improper number of qualifiers.
-142	SQL0142N	SQLA_RC_DDSBAD	N	SQL statement is not supported.
-198	SQL0198N	SQLA_RC_EC	N	The statement is blank or empty.
-199	SQL0199N	SQLA_RC_KEYWORD_BAD	N	Invalid use of reserved word.
-306	SQL0306N	SQLA_RC_HVAR_NOT_DEC	N	Host variable used but not declared.
-307	SQL0307N	SQLA_RC_HVAR_DUP_NAME	N	Host variable already declared.
-308	SQL0308N	SQLA_RC_HVAR_LIMIT	N	Maximum number of host variables exceeded.
-310	SQL0310N	SQLA_RC_STMT_HVAR_LIMIT	N	Number of host variables in statement exceeds limit.
-311	SQL0311N	none	N	Length of host variable is negative.
-324	SQL0324N	SQLA_RC_HVAR_USE_BAD	N	Host variable may not be used in this context.
-505	SQL0505N	SQLA_RC_CURSOR_DUP	N	Duplicate cursor name.
-751	SQL0751N	SQLA_RC_INVALID_TRIGGER_STMT	N	Statement unsupported within trigger.
-803	SQL0803N	SQLA_RC_INV_INSERT	Y	Insert causes duplicate row in table with unique index.
-804	SQL0804N	SQLA_RC_SQLDA_SQLD_ERR, SQLA_RC_SQLVAR_TYPE_ERR	Y	Invalid sqlda contents (incorrect sqltype, sqllen, or other).
-822	SQL0822N	SQLA_RC_E822	Y	Invalid address in SQLDA.
-902	SQL0902C	SQLA_RC_SYS_ERROR.	Y	System error.
-911	SQL0911N	SQLA_RC_DEADLOCK_ERR	N	Transaction has been rolled back because of deadlock.
-912	SQL0912N	SQLA_RC_TOO_MANY_LKS	Y	Maximum number of lock requests exceeded.
-930	SQL0930N	SQLA_RC_FAT_SYS_ERR	Y	Fatal system error.
-954	SQL0954C	SQLA_RC_STORAGE_ERR	Y	Not enough storage to process statement.
-956	SQL0956C	SQLA_RC_DB_HEAP_ERR	Y	Not enough storage to process statement.
-958	SQL0958C	SQLA_RC_TOOMANY_OFLS	Y	Maximum number of open files exceeded.
-960	SQL0960C	SQLA_RC_TOOMANY_FILES	Y	Maximum number of files in database exceeded.
-964	SQL0964C	SQLA_RC_LOG_FULL	Y	Transaction log full.
-968	SQL0968C	SQLA_RC_DISK_FULL	Y	File system full.

Table 4. Error Messages and Codes (continued)

Value	Code	Constant	Fatal?	Description
-970	SQL0970N	SQLA_RC_READ_ONLY_FIL	Y	Attempt to write to a read-only file.
-972	SQL0972N	SQLA_RC_INCORRECT_DSK	Y	Database drive does not contain correct diskette.
-974	SQL0974N	SQLA_RC_DB_DRV_LOCKED	Y	Database drive locked.
-976	SQL0976N	SQLA_RC_DRV_DOOR_OPEN	Y	Database drive door open.
-978	SQL0978N	SQLA_RC_DISK_WRT_PRO	Y	Database drive diskette write protected.
-980	SQL0980C	SQLA_RC_DISK_ERROR	Y	Disk error.
-982	SQL0982N	SQLA_RC_RDS_DISK_ERR	Y	Disk error.
-984	SQL0984C	SQLA_RC_COMM_RB_ERR	Y	Unsuccessful COMMIT or ROLLBACK.
-985	SQL0985C	SQLA_RC_CAT_FILE_ERR	Y	File error in catalog.
-986	SQL0986N	SQLA_RC_TAB_FILE_ERR	Y	File error in user table.
-990	SQL0990C	SQLA_RC_INDEX_ERR	Y	Index error.
-992	SQL0992C	SQLA_RC_REL_NUM_BAD	Y	Release number of precompiled program invalid.
-1024	SQL1024	none	Y	No database connection exists.
-1216	SQL1216N	none	N	Invalid use of graphic data.
-1224	SQL1224N	SQLA_RC_AGENT_GONE	Y	Database agent could not be started.
-4010	SQL4010N	SQLA_RC_CMPD_NESTED	N	Illegal nesting of compound SQL statements.
-4011	SQL4011N	SQLA_RC_CMPD_INVALID_STMT	N	Invalid substatement in a compound SQL statement.
-4012	SQL4012N	SQLA_RC_CMPD_INVALID_COMMIT	N	Invalid use of COMMIT within a compound SQL statement.
-4013	SQL4013N	SQLA_RC_CMPD_INVALID_END	N	END COMPOUND found without previous BEGIN COMPOUND.
-4901	SQL4901N	SQLA_RC_FATAL_ERROR	Y	Reinitialization has not occurred since last fatal error.
-4902	SQL4902N	SQLA_RC_PARM_CHARS_BAD	N	Invalid characters in parameter.
-4903	SQL4903N	SQLA_RC_PARM_LENGTH_BAD	N	Invalid parameter length.
-4904	SQL4904N	SQLA_RC_PARM_POINTER_BAD	N	Invalid pointer to parameter.
-4905	SQL4905N	SQLA_RC_PARM_RANGE_BAD	N	Parameter not within valid range.
-4911	SQL4911N	SQLA_RC_HVAR_SQLTYPE_BAD	N	SQL type for host variable is invalid.

Table 4. Error Messages and Codes (continued)

Value	Code	Constant	Fatal?	Description
-4912	SQL4912N	SQLA_RC_HVAR_SQLLEN_BAD	N	Length of host variable is out of range.
-4913	SQL4913N	SQLA_RC_VAR_TOKEN_ID_DUP	N	Token ID has already been used.
-4914	SQL4914N	SQLA_RC_HVAR_TOKEN_ID_BAD	N	Invalid token ID.
-4915	SQL4915N	SQLA_RC_INIT_DUP	Y	db2Initialize has already been invoked.
-4916	SQL4916N	SQLA_RC_INIT_REQUIRED	Y	sqlaalhv has not been invoked.
-4917	SQL4917N	SQLA_RC_OPTION_BAD	Y	Unsupported option found in option array.
-4918	SQL4918N	SQLA_RC_TERM_OPTION_BAD	N	Invalid termination option.
-4919	SQL4919N	SQLA_RC_TASK_ARRAY_LIMIT	N	Task array too small.
-4920	SQL4920N	SQLA_RC_TOKEN_ARRAY_LIMIT	N	Token array too small.
-4940	SQL4940N	SQLA_RC_STMT_CLAUSE_BAD	N	Illegal clause in statement.
-4941	SQL4941N	SQLA_RC_STMT_BLANK	N	Blank or empty SQL statement text.
-4942	SQL4942N	SQLA_RC_SELECT_HVAR_TYPE_BAD	N	Incompatible data type selected into host variable.
-4944	SQL4944N	SQLA_RC_COLUMN_NOT_NULLABLE	N	Attempt to store a NULL value in a NOT NULL column.
-4945	SQL4945N	SQLA_RC_STMT_MARKER_BAD	N	Invalid use of a parameter marker.
-4946	SQL4946N	SQLA_RC_CURSOR_NOT_DECLARED	N	Cursor not declared.
-4951	SQL4951N	SQLA_RC_SQLDA_ID_BAD	N	Invalid SQLDA ID.
-4952	SQL4952N	SQLA_RC_SQLVAR_INDEX_BAD	N	Invalid sqlvar_index parameter.
-4953	SQL4953N	SQLA_RC_CALL_TYPE_BAD	N	Invalid call type.
-4954	SQL4954N	SQLA_RC_SECTION_BAD	N	Section number is invalid.
-4994	SQL4994N	SQLA_RC_CTRL_BREAK	Y	Interrupt key sequence detected.
-4995	SQL4995C	SQLA_RC_CODEPAGE_BAD	Y	Cannot find COUNTRY.SYS.
-4997	SQL4997N	SQLA_RC_SQLUSER_BAD	Y	Invalid authorization ID.
-4998	SQL4998C	SQLA_RC_DB_DISCONNECTED	Y	Database connection has been lost.
-4999	SQL4999C	SQLA_RC_INTERNAL_ERR	Y	Internal error.

Index

S

T

Printed in USA