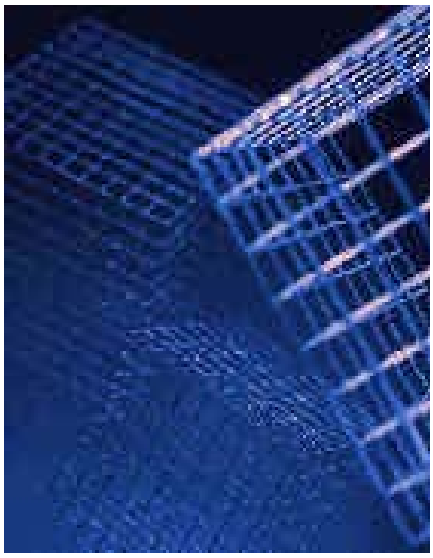
C/C++-Precompiler User Manual



Version 7.4

Copyright

Icons

Icon	Meaning
	Caution
	Example
	Note
	Recommendation
	Syntax

Typographic Conventions

Type Style	Description
Example text	Words or characters that appear on the screen. These include field names, screen titles, pushbuttons as well as menu names, paths and options. Cross-references to other documentation.
Example text	Emphasized words or phrases in body text, titles of graphics and tables.
EXAMPLE TEXT	Names of elements in the system. These include report names, program names, transaction codes, table names, and individual key words of a programming language, when surrounded by body text, for example, SELECT and INCLUDE.
`Example text`	Screen output. This includes file and directory names and their paths, messages, source code, names of variables and parameters as well as names of installation, upgrade and database tools.
`EXAMPLE TEXT`	Keys on the keyboard, for example, function keys (such as `F2`) or the `ENTER` key.
Example text	Exact user entry. These are words or characters that you enter in the system exactly as they appear in the documentation.
<Example text>	Variable user entry. Pointed brackets indicate that you replace these words and characters with appropriate entries.

C/C++ Precompiler User Manual: SAP DB 7.4

The interface between the SAP DB database system and SAP DB application programs is the database language SQL (Structured Query Language). SQL statements embedded [Page 9] in the application program (Embedded SQL) are used for communication with the database instance. Parameter values are exchanged in special program variables, called host variables [Page 15].

The C/C++ Precompiler [Page 60] prepares C/C++ source code with embedded SQL statements for translation into an executable application program. The compiler checks the syntax and semantics of the embedded statements, converts them to procedure calls in the precompiler runtime environment [Page 65], and generates a C/C++ file, which can then be compiled.

This user manual describes how to embed SQL statements for the SAP DB database system in the programming language C/C++, and how to work with the precompiler.

This manual requires a sound working knowledge of the meaning of elementary SQL statements. For a description of the syntax and semantics of database statements, and the conventions for syntax notation [See SAP DB Library], see the Reference Manual: SAP DB 7.4 [See SAP DB Library].

For general information on the SAP DB database system, see the documentation The SAP DB Database System on the SAP DB Homepage *http://www.sapdb.org* under *Documentation.*

Embedding SQL Statements in C/C++

When you embed database statements in the source code of an application program, observe the following points as well as the general rules [Page 9]:

- Conventions for the Order of Embedded SQL Statements [Page 10]
- Rules for the Declaration Segment [Page 11]
- Conventions for Host Variables [Page 16]
- Using Indicator Variables [Page 18]
- Permitted Data Types [Page 20]
- Connecting to a Database Instance [Page 30]
- Overview of Precompiler Statements [Page 30]
- Database System Messages [Page 55]
- Programming Notes [Page 58]

General Rules

Observe the following general rules when you embed SQL statements [Page 9].

- Precede each embedded SQL statement with the key words **`EXEC SQL`** to delimit it from the programming language statements. These key words must be on the same line, and can only be split with a blank character.
- Finish each embedded SQL statement with a semicolon.
- You can write embedded SQL statements that run over multiple lines.

- You can interrupt embedded SQL statements with comments. Place the comments between the characters **/*** and ***/**.
- You can place comments about SQL statements at the end of the line. Precede comments at the end of a line with two dashes, **--**.
- Precede the names of host variables [Page 15] and indicator variables [Page 18] within embedded SQL statements with a colon **:**
- **No other** names used in an application program are allowed to start with a colon **:**
- Names used in an application program must not start with the characters **sq**. This combination of characters is also reserved by the precompiler.
- The program parts analyzed by the precompiler must not be elements of the preprocessor include statements (**#define**, **#include**).

See also: Overview of Precompiler Statements [Page 30]

```
EXEC SQL BEGIN DECLARE SECTION;
char addr [6];
char fname [8], lname [8];
EXEC SQL END DECLARE SECTION;

/* Generate table „customer“ */

EXEC SQL CREATE TABLE customer
(cno FIXED (4) KEY, address CHAR (5),
firstname CHAR (7), lastname CHAR (7));

/* Get values */

/* Add values to database */

EXEC SQL INSERT INTO customer
(cno, address, first name, last name)
VALUES (100, :addr, :fname, :lname);

/* Message in event of error */

if (sqlca.sqlcode != 0)
printf ("%d       ",sqlca.sqlcode);
```

Conventions for the Order of SQL Statements

When the precompiler runs [Page 61] with the precompiler option [Page 62] `check`, it runs once, sequentially, through the source code of the application program, and sends any static SQL statements [Page 35] that it finds to the database kernel . So that the check can run correctly, the embedded SQL statements must appear in an executable order in the source code.

Observe the following conventions when you embed SQL statements [Page 9].

- `CREATE TABLE` comes before `INSERT`, `UPDATE`, `DELETE`, and `SELECT`.
- `INSERT`, `UPDATE`, `DELETE`, and `SELECT` come before `DROP TABLE`.
- `SELECT` comes before `FETCH`. It must be possible to make a unique assignment between the SELECT statement and the FETCH statement.
- `DECLARE CURSOR` comes before `OPEN`.
- `OPEN` comes before `FETCH`.

- `FETCH` comes before `CLOSE`.

The process flow of the program solely is responsible for the order in which the SQL statements are actually executed at runtime of the program.

The precompiler does not send dynamic SQL statements [Page 37] to the database, since their content is unknown before runtime.

Rules for the Declare Section

In the declare section, you declare all host variables [Page 15] and indicator variables [Page 18] that are exchanged between the database instance and the application program in embedded SQL statements [Page 9]. You can use multiple declaration sections within an application program.

Observe the following rules:

- Start the declare section with **`EXEC SQL BEGIN DECLARE SECTION;`**.
- End the declare section with **`EXEC SQL BEGIN DECLARE SECTION;`**.
- Follow the permitted syntax for the declaration section [Page 12].
- Observe the conventions for host variables [Page 16] and the rules for indicator variables [Page 19].
- Use the data types [Page 20] permitted for the precompiler.
- You can use the memory classes **`typedef`**, **`static`**, **`extern`** or **`auto`**.
- You can specify declarations both outside of and within functions.
- As well as variable declarations, you can specify type definitions and parameter-free constant definitions (**`#define`**) for positive whole numbers in the declare section.

`#define n(a) (a+10)` is not permitted.
`#define n 7` is permitted.
You can, for example, use **`n`** as the length of an array variable:

```
int array [n]
```

- Also specify array lengths for the memory class **`extern`**. These lengths must lie within the range of the corresponding external definitions.

To improve efficiency, only declare those variables in **`EXEC SQL`** declare sections that you are actually using as host variables in embedded SQL statements.

```
EXEC SQL BEGIN DECLARE SECTION;
char no [6];
char fname [8], lname [8];
EXEC SQL END DECLARE SECTION;
```

Syntax of the Declare Section

This syntax list shows you the syntax of the declaration section. The syntax notation [See SAP DB Library] used is BNF.

```
<arrdim> ::=
  <identifier>
| <unsigned_integer>


<chardecspec> ::=
  <storclspec>
| <chartype>
| <chardecspec> [<storclspec>]
| <chardecspec> <typespec>


<chartype> ::=
  char
| VARCHAR
| varchar


<constant> ::=
  <identifier>
| <unsigned_integer>


<decimaldef> ::=
  DECIMAL [<decimaltag>] [(<decimalscale>)]


<decimaldigits> ::=
  <identifier>
| <unsigned_integer>


<decimalfract> ::=
  <identifier>
| <unsigned_integer>


<decimalref> ::=
  DECIMAL <identifier>


<decimalscale> ::=
  decimaldigits
| decimaldigits,decimalfract
```

```
<decimaltag> ::=
  <identifier>


<decimaltype> ::=
  <decimaldef>
| <decimalref>


<declaration> ::=
  <decspec> <declist>;


<declarator> ::=
  <identifier>
| * <identifier>
| <declarator> [<arrdim>]


<declist> ::=
  <possinitdec>
| <declist>,<possinitdec>


<decspec> ::=
  <typespec>
| <chardecspec>
| <intdecspec>
| <filedecspec>
| <floatdecspec>
| decimaltype
| <decspec> [<storclspec>]
| <decspec> <typespec>
| <longdescdecspec>


<expression> ::=
  <basexpr>
| <basexpr> (<expression>) <basexpr>


<fieldlist> ::=
  <typespec> <declist>;
| <fieldlist> <typespec> <declist>;


<filedecspec> ::=
  <storclspec>
| <filetype>
| <chardecspec> [<storclspec>]
| <filedecspec> <typespec>


<filetype> ::=
  SQLFILE
| sqlfile


<floatdecspec> ::=
  <storclspec>
| <floattype>
```

```
| <chardecspec> [<storclspec>]
| <floatdecspec> <typespec>


<floattype> ::=
  float
| long
| double


<initializer> ::=
  <expression>
| {<initlist>}


<initlist> ::=
  <initializer>
| <initlist>,<initializer>


<intdecspec> ::=
  <storclspec>
| <inttype>
| <chardecspec> [<storclspec>]
| <intdecspec> <typespec>


<inttype> ::=
  short
| long
| int
| unsigned


<longdescdecspec> ::=
  <storclspec>
| <longdesctype>
| <longdescdecspec> [<storclspec>]
| <longdescdecspec> <typespec>


<longdesctype> ::=
  SQLLongDesc
| sqllongdesc


<possinitdec> ::=
  <declarator>
| <declarator> = <initializer>


<prepcomline> ::=
  #define <identifier_constant>


<statement> ::=
  <prepcomline>
| <declaration>
```

```
<storclspec> ::=
  external
| static
| auto
| typedef


<structdef> ::=
  struct [<structag>] {<fieldlist>}


<structref> ::=
  struct <identifier>


<structag> ::=
  <identifier>


<structype> ::=
  <structdef>
| <structref>


<typespec> ::=
  <inttype>
| <floatptye>
| <chartype>
| <floattype>
| <decimaltype>
| <longdesctype>
| <structype>
| <identifier>
| const
| volatile


<unichardecspec> ::=
  <storclspec>
| <unichartype>
| <unichardecspec> [<storclspec>]
| <unichardecspec> <typespec>


<unichartype> ::=
  SQLUCS2
| SQLUTF16
| TCHAR (only available for use with Unicode applications)
```

Host Variables

Host variables are variables that are used in embedded SQL statements [Page 9] to exchange values between the application program and the database instance .

Observe the following points when you use host variables:

- Declare the host variables in the declare section [Page 11]. When you do this you can use data types predefined [Page 21] by the precompiler, as well as C/C++ basic data types.

See also: Permitted Data Types [Page 20]

- Observe the conventions for host variables [Page 16]. You can also use structures [Page 16] and arrays [Page 17] in host variables. You can generate structure definitions [Page 26] from database tables and database procedures.

Conventions for Host Variables

Observe the following conventions for host variables [Page 15].

- The name of a host variable can have a maximum of 32 characters.
- The name of a host variable cannot start with the characters **sq**.
- A host variable can be a structure [Page 16] or an array [Page 17]. A structure can contain arrays, and arrays of structures are also possible.
- A host variable can be a pointer variable of permitted data types [Page 20]. *Pointer to Pointer* and *Pointer to Array* are not permitted.
- In each declaration, you can specify a maximum of one pointer and arrays with up to four dimensions. Arrays of pointer variables are permitted, but not *Pointer to Array*.
- The precompiler cannot check the lengths of host variables with the data type `char *`, and writes a warning to the precompiler log [Page 64]. At runtime, the character pointer must point to a NULL-delimited character string.
- So that they can include the closing NULL bytes, host variables with the data type `char [n]` must be declared with one more field than the corresponding column in the database table.
  When `char` variables are send to a database table, the variable content is copied up to the closing NULL byte. Longer table columns are filled up with blank characters.
  When `char` variables are read from a database table, all following blank characters in the table content are truncated, and the closing NULL byte is set in the variable after the last character that is not a blank character. If the host variable is shorter than the table column, then the last character is overwritten.

Structures as Host Variables

Host variables [Page 15] can be structures. When the precompiler runs [Page 61], it splits the structure into its individual components, in the order of the component declarations.

When you use structures as host variables, you can, for example, specify multiple values in the INTO clause of a SELECT statement at the same time.

```
EXEC SQL BEGIN DECLARE SECTION;

typedef char string8 [8];

  struct {
    char addr [6];
    struct {
      string8 lname, fname [3];
    } name;
  } person;

EXEC SQL END DECLARE SECTION;
```

```
EXEC SQL SELECT !person INTO :person
         FROM customer WHERE cno = 100;
```

You can use a simplified notation [Page 17] for structures in SQL statements.

You can also specify the corresponding indicator variable [Page 18] as a structure with the same number of components.

You can generate [Page 26] the structure definition from a database table or a database procedure.

Arrays as Host Variables

Host variables [Page 15] can be arrays. In an array statement [Page 48] they cause an SQL statement to be executed more than once.

In multi-dimensional arrays, the last dimension is run first.

```
EXEC SQL BEGIN DECLARE SECTION;
float p[3][2];
EXEC SQL END DECLARE SECTION;
EXEC SQL CREATE TABLE KOORD (x float, y float);
p[0][0] = 0.0;
p[0][1] = 0.1;
p[1][0] = 1.0;
p[1][1] = 1.1;
p[2][0] = 2.0;
p[2][1] = 2.1;
EXEC SQL INSERT INTO KOORD VALUES (:p);
/* insert generates following content */
/*      x  |  y                                    */
/*     ---------                                   */
/*     0.0 | 0.1                                   */
/*     1.0 | 1.1                                   */
/*     2.0 | 2.2                                   */
```

Also specify the corresponding indicator variable [Page 18] as an array, with the same length and dimension.

You can use a simplified notation [Page 17] for array variables in SQL statements.

Simplified Notation for Structure and Array Variables

Use

In SQL statements, you can use a simplified notation for structure variables [Page 16] and array variables [Page 17], so avoiding the need to list the columns of the database table that are addressed explicitly.

Procedure

- Specify the name of a structure variable or array variable as **!<var>**, for example, in the **<select_list>** of the SELECT statement.
- You can use the tilde character (**~<var>**) instead of an exclamation mark.
- The precompiler derives the column names from the names of the structure components or array elements. The column names can have a maximum of 18 characters.
- You can specify **<[owner.]tablename.>** before **!<var>**. This information is then inserted in front of each column name.
- Instead of the complete variable **!<var>**, you can specify a structured component of **var** of your choice, for example, **!var.x.y**. In this case, the appropriate subset of column names is generated.

```
EXEC SQL BEGIN DECLARE SECTION;

  typedef char string8 [8];
  struct {char addr [6];
  struct {string8 lname, fname [3];} name;} person;

EXEC SQL END DECLARE SECTION;

EXEC SQL CREATE TABLE customer
  (cno FIXED(4) key, addr CHAR(5),
  name_lname CHAR(7), name_fname1 CHAR(7),
  name_fname2 CHAR(7), name_fname3 CHAR(7));

/* Get values */

EXEC SQL INSERT INTO customer
  (cno, !person )
  VALUES (100, :person);

/* Has same effect as: */

EXEC SQL INSERT INTO customer
  (cno, addr, name_lname,
  name_fname1, name_fname2, name_fname3)
  VALUES (100, :person.addr,
  :person.name.lname, :person.name.fname[0],
  :person.name.fname[1],
  :person.name.fname[2]);
```

Indicator Variables

If you want your application program to process NULL values or DEFAULT values, or if values might be truncated when they are exchanged between host variables [Page 15] and the database, then you must specify an indicator variable as well as the host variable in the embedded SQL statement [Page 9].

Observe the rules for indicator variables [Page 19].

The value of an indicator variable is called an indicator value [Page 20], and performs the following tasks:

- It provides information on whether the corresponding host variables have been processed correctly. For this reason, query the indicator value after the SQL statement has been processed.
- It is used to transfer NULL values to the database instance [Page 29].

- It is used to transfer DEFAULT values to the database instance [Page 29].
- It is used in GETVAL statements [Page 50] and PUTVAL statements [Page 49] to ignore individual LONG columns.

If a SELECT statement or a FETCH statement specifies that values are transferred from columns in which NULL values can appear, then you must specify an indicator variable. Otherwise the C/C++ precompiler writes a warning when the precompiler runs [Page 61] with the precompiler option [Page 62] `check`, and you see an error message when you select a NULL value at runtime (`sqlcode = -809`).

In dynamic SQL statements with a descriptor [Page 39], the `hostindicator` component of an sqlvar entry [Page 40] performs the same task as an indicator variable.

Rules for Indicator Variables

Note the following points when you use indicator variables [Page 18]:

- Declare each indicator variable as a data type `long int`, `int` or `short int` in the declaration segment [Page 11].
- Specify the indicator variable behind the corresponding host variable [Page 15] in the embedded SQL statement [Page 9], divided by a blank character.
- Observe the same conventions as for host variables [Page 16].

```
EXEC SQL BEGIN DECLARE SECTION;
char fname [8], lname [8];
int fnameind, lnameind;
EXEC SQL END DECLARE SECTION;
/* Insert NULL value */
fnameind = -1;
strcpy (lname, "COMPANY X");
lnameind = 0;
EXEC SQL INSERT INTO customer (firstname,lastname)
VALUES (:fname :fnameind, :lname :lnameind);
/* Test for truncation */
EXEC SQL SELECT lastname
INTO :lname :lnameind
FROM customer
WHERE lname = "COMPANY X";
if (lnameind > 0)
printf ("%d      ", lnameind );
```

Indicator Values

The table shows the possible indicator values that can be taken by an indicator variable [Page 18] of the `hostindicator` component or an sqlvar entry in the descriptor [Page 40].

Indicator Value	Meaning
`0`	A value has been transferred between the database and the application program without errors. The value sent to the table column or host variable [Page 15] is a defined value.
`-1`	The value of the host variables or of the corresponding table entry is the NULL value.
`-2`	An error occurred (overflow, or division by zero). The value of the host variables or of the corresponding table entry is not defined.
`> 0`	A truncated value has been sent to the host variable or to the corresponding table column. The indicator value specifies the actual length of the value.
`SQL_DEFAULT_PARAM`	Used to transfer DEFAULT values to the database instance [Page 29].
`SQL_IGNORE`	Used to ignore LONG columns in GETVAL statements [Page 50] and PUTVAL statements [Page 49]

See also: Database System Messages (sent through the sqlca structure) [Page 55]

Permitted Data Types

You can use the following data types for host variables [Page 15] in the declaration section [Page 11]:

- All basic data types [Page 20]
- The data type `struct`
- The data types predefined [Page 21] by the precompiler (`VARCHAR, SQLFILE and SQLLongDesc`), and UNICODE data types [Page 24]
- Any data types you have defined yourself previously in a declaration section

Basic Data Types

In the declare section [Page 11], you can use the C/C++ basic data types to declare host variables [Page 15]. A corresponding SAP DB data type exists for each basic data type [See SAP DB Library].

The precompiler performs the following tasks:

- When the precompiler runs [Page 61], it checks whether the memory of the host variables and the corresponding table column is large enough for the maximum possible value to be sent between the database instance and the application program. If this is not the case, the precompiler writes a warning in the precompiler log [Page 64].
- If necessary, the precompiler converts the data types [Page 25] at runtime.

The following table shows the corresponding SAP DB data types for the basic data types:

Description	C/C++ Data Type	SAP DB Data Type

Alphanumeric character	`char`	`CHAR (1)`

Character string with closing NULL byte	`char*`
	`char [n+1], n ≤ 8000`
	`char [n+1], n > 8000`
		`CHAR (n)`
		`VARCHAR (n)`
		`LONG`

Integer	`[unsigned] int,` `short int, long int`	`SMALLINT, INTEGER (2 or 4 bytes),` `FIXED (5), FIXED (10)`
Fixed point number	`float, double`	`FIXED (n,m)`
Floating point number	`float` `double`	`FLOAT (6)` `FLOAT (15)`
Date	`char [9]`	`DATE`
Time	`char [9]`	`TIME`
Timestamp with closing NULL byte	`char [21]`	`TIMESTAMP`
Boolean	All numeric data types (null or not null)	`BOOLEAN`
Byte	`char [n], n ≤ 8000` `char [n], n > 8000`	`[VAR]CHAR (n) BYTE` `LONG BYTE`

When you specify dates, times, and timestamps, stick to the correct date and time format [See SAP DB Library].

Predefined Data Types

The C/C++ precompiler predefines the following permitted data types [Page 20] to make it easier to handle database tables. You can use them to declare host variables [Page 15] in the declaration section [Page 11].

- VARCHAR [Page 21]
- SQLFILE [Page 22]
- UNICODE data types [Page 24]

VARCHAR

Use

You can use the predefined data type [Page 21] VARCHAR to declare host variables [Page 15] to which you want to assign character strings with variable lengths.

The C/C++ precompiler converts the VARCHAR declaration into a structure declaration with a two-byte length and an array or pointer. The current length of a VARCHAR variable is determined by the length field. NULL bytes do not contribute to the calculation of the length. In a VARCHAR declaration with pointer declarer, the application program is responsible for assigning memory at runtime.

```
VARCHAR v [n];
```

is replaced by

```
struct {unsigned short len; unsigned char arr [n];} v;
```

where the current length of the character string is assigned to `len` and the characters themselves are assigned to `arr`.

```
VARCHAR *v;
```

is replaced by

```
struct {unsigned short len; unsigned char arr [1];} *v;
```

See also: Examples for Permitted VARCHAR Declarations [Page 22]

Examples for Permitted VARCHAR Declarations

Examples for permitted VARCHAR [Page 21] declarations:

```
VARCHAR a [21], b [100] [133];
```

```
typedef VARCHAR longstring [ 65534 ];
longstring c, d;
```

```
typedef VARCHAR *PVC;
PVC p;
```

You can assign memory for `p`:

```
n = 100; /* Maximum length of VARCHAR variables*/
p = (PVC) malloc (sizeof (p->len) + n * sizeof (p->arr));
```

You can declare VARCHAR pointers with fixed maximum lengths as follows:

```
typedef VARCHAR VC30 [30];
VC30 *q;
```

`q` is a pointer to a VARCHAR with a maximum length of 30.

Memory is assigned to `q` with the following statement:

```
q = (VC30* ) malloc (sizeof (VC30));
```

SQLFILE

Use

You can enable the direct transfer of file content in and out of LONG columns [Page 28] by using the predefined data type [Page 21] SQLFILE to declare host variables [Page 15].

Procedure

Assign the name of a file to the value of the host variable. Note the following:

- If this file already exists, then its content must be a character string (of any length). The SQL statements INSERT and UPDATE write the content of the file to the LONG column.
- If this file does not already exist, the SELECT statements of a LONG column generate it with the specified file name. The column content becomes the content of the file.
- If this file already exists when a SELECT statement is executed, then the content of the LONG column is appended to its existing content. If you want to avoid this, you must delete the existing file before the SELECT statement is executed, for example, with an EXEC COMMAND statement [Page 54].

- Any errors in the processing of the file terminate the SQL statement and trigger a database error message [Page 55] `sqlcode < 0`.

```
EXEC SQL BEGIN DECLARE SECTION;
SQLFILE f1[30] = "document.doc";
char title[41] = "Manual";
SQLFILE f2[30];
EXEC SQL END DECLARE SECTION;
EXEC SQL CREATE TABLE documents(title CHAR (40) KEY,
document LONG BYTE);
EXEC SQL INSERT INTO documents VALUES (:title, :f1);
strcpy (f2, "tempdocument.doc");
EXEC SQL SELECT document INTO :f2 FROM documents
WHERE title = :title;
```

SQLLongDesc

You can make working with LONG columns [Page 28] easier by using the predefined data type [Page 21] SQLLongDesc to declare host variables [Page 15]. In a FETCH statement, for example, this gives you information on the total length of the LONG column.

The data type SQLLongDesc is declared as a structure in the header file `cpc.h`:

```
typedef struct sqllongdesc {
  char *szBuf;
  int cbColLen;
  int cbBufLen;
  int cbBufMax;
} sqllongdesc;

typedef sqllongdesc SQLLongDesc;
```

The components of this structure declaration have the following meaning:

- `szBuf`: Pointer to a memory area that includes the LONG data
- `cbColLen`: Total length of the corresponding LONG column in the table
- `cbBufLen`: Length of the data currently saved in `szBuf`
- `cbBufMax`: Size (in bytes) of the memory area to which `szBuf` points

When a FETCH or GETVAL [Page 50] statement is executed, the precompiler enters the length of the LONG column in the component `cbColLen`.

You must assign values to the other components before SQL statements. See also: Special Features when Using Data Type SQLLongDesc [Page 24]

Special Features when Using Data Type SQLLongDesc

Note the following points when you use the data type SQLLongDesc:

- If you want to execute INSERT statements or PUTVAL statements [Page 49] with host variables [Page 15] of data type SQLLongDesc [Page 23], you must first assign values to the structure components `szBuf`, `cbBufMax` and `cbBufLen` of this data type.
- If you want to execute FETCH statements or GETVAL statements [Page 50] with host variables [Page 15] of data type SQLLongDesc [Page 23], you must first assign values to the structure components `szBuf` and `cbBufMax` of this data type.

```
SQLLongDesc ldesc;
#define LONGLEN 100000
#define FILLLEN 80000
/* Assign memory for including data */
ldesc.szBuf = (char *) malloc (LONGLEN);
/* Fill memory with values */
memset(ldesc.szBuf, 'X', FILLLEN);
/* Specify size of memory area */
ldesc.cbBufMax = LONGLEN;
/* Specify length of data in buffer */
Ldesc.cbBufLen = FILLLEN;
EXEC SQL INSERT INTO LONGTEST VALUES (:i1, :ldata :il1, :l2
:il2);
```

UNICODE Data Types

The C/C++ precompiler provides predefined data types [Page 21] for working with UNICODE data [Page 27].

The table gives you an overview of these UNICODE data types:

Description	C/C++ Data Type	SAP DB Data Type

Unicode character string with closing NULL byte
 	
The basic C/C++ data type is `unsigned short`.
 	
In UCS 2, the characters are coded in a platform-specific byte configuration.
	`SQLUCS2*`
	`SQLUCS2 [n+2], n < 4000`
	`SQLUSC2 [n+2], n ≥ 4000`
		`CHAR (n) UNICODE,`
		`VARCHAR (n) UNICODE`
		`LONG UNICODE`

Unicode character string with closing NULL byte	`SQLUTF16*` `SQLUTF16 [n+2], n < 4000`	`CHAR (n) UNICODE,` `VARCHAR (n) UNICODE`

The basic C/C++ data type is `unsigned short`. In UTF 16 without surrogate, the characters are coded in a platform-specific byte configuration.	`SQLUTF16 [n+2], n ≥ 4000`	`LONG UNICODE`
Generic data type `CHAR` or `SQLUCS2` data type, depending on the precompiler option [Page 62] – **`G unicode`**.	`TCHAR* TCHAR [n+1], n < 4000` `TCHAR [n+1], n ≥ 4000`	`CHAR (n) UNICODE, VARCHAR (n) UNICODE LONG UNICODE`

See also: Converting Data Types [Page 25]

Converting Data Types

If, in an embedded SQL statement [Page 9], the C/C++ basic data type [Page 20] or the UNICODE data type [Page 24] of a parameter does **not** correspond to the SAP DB data type of the corresponding table column, but **can** be converted, then the precompiler runtime environment [Page 65] converts the data types at runtime of the application program.

The following table shows the possible conversions, and the errors that can occur:

C/C++ Basic Data Types	SAP DB Data Types						
	FIXED	**FLOAT**	**BOOLEAN**	**LONG**	**CHAR**	**VARCHAR**	**DATE/TIME**
short int, int, long int	1,2	1b,2	6	Not permitted	4,5	Not permitted	Not permitted
float, double, long float	1a,2	2	6	Not permitted	4,5	Not permitted	Not permitted
char [n]	4,5	4,5	Not permitted	3	3	3	3
char	4,5	4,5	Not permitted	Not permitted	3	3	3

UNICODE Data Types	SAP DB Data Types						
	FIXED	**FLOAT**	**BOOLEAN**	**LONG**	**CHAR**	**VARCHAR**	**DATE/TIME**
SQLUCS2 [n], SQLUTF16 [n], TCHAR [n]	4,5,7	4,5,7	Not permitted	3,7	3,7	3,7	3,7
SQLUCS2, SQLUTF16, TCHAR	4,5,7	4,5,7	Not permitted	3,7	3,7	3,7	3,7

1 An overflow can occur (`sqlcode < 0`).

1a An overflow can occur when a value is transferred to the database table.

1a An overflow can occur when a value is transferred to the host variable [Page 15].

2 Some decimal places may be cut off (indicator value [Page 20] `> 0`).

3 Some characters may be cut off. The warning message [Page 57] `sqlwarn1` is set. The indicator value specifies the actual length of the character string.

4 An overflow can occur when numerical values are converted into character strings (`sqlcode < 0`).

5 An overflow or invalid number can occur when character strings are converted into numerical values (`sqlcode < 0`).

6 The value 0 is mapped to `false`. All values other than 0 are mapped to `true`.

7 An error message is written when UNICODE values are converted that cannot be translated to 8 bit ASCII (`sqlcode < 0`).

See also: Database System Messages [Page 55]

Generating Structure Definitions

Use

You can use the INCLUDE DECLARE statement [Page 32] to generate a file `<file_name>` from a database table or database procedure. This file derives a corresponding structure definition from the structure of the database table. You can use this structure definition to declare host variables [Page 15] and the corresponding indicator variables [Page 18].

A table has been defined as follows:

```
CREATE TABLE example (
  A FIXED (5),
  B FIXED (8),
  C FIXED (5, 2),
  D FLOAT (5),
  E CHAR (80))
```

The application program can then contain the following statements:

```
EXEC SQL BEGIN DECLARE SECTION;
EXEC SQL INCLUDE "example.h" TABLE example AS STRUCT IND;
struct example s, sa[10], *sp;
struct iexample indi;
EXEC SQL END DECLARE SECTION;
EXEC SQL SELECT * FROM example;
EXEC SQL FETCH INTO :s :indi;
EXEC SQL FETCH INTO :sa[4] :indi;
sp = &sa[9];
EXEC SQL FETCH INTO :sp :indi;
```

The include statement generates the following declarations:

```
struct example {short A;long B;float C,D;char E [81];};
```

and

```
struct iexample {short IA, IB, IC, ID, IE;};
```

Working with UNICODE Data

Use

The C/C++ Precompiler supports the use of UNICODE [See SAP DB Library] data in SAP DB application programs. The only permitted coding formats are UCS2 or UTF-16 (without surrogate), both with a platform-specific byte configuration. All output in the trace file is in UTF 8.

You can use UNICODE character strings in the following locations in host variables [Page 15] only:

- As user name and password in the CONNECT statement [Page 34]
- As a statement name in dynamic SQL statements [Page 37]
- As a result table name in OPEN CURSOR statements [Page 46]
- As an SQL statement
- As parameters for SQL statements

You cannot use UNICODE application programs with ASCII database instances.

Prerequisite

You have installed a UNICODE-compliant database instance [See SAP DB Library].

After a CONNECT statement, the precompiler can recognize automatically whether it is connected to a UNICODE-compliant database instance.

Procedure

- Use suitable host variables with the appropriate predefined UNICODE data types [Page 24] for your UNICODE character strings.
- If the user name or password contains UNICODE characters, then also use UNICODE host variables.
- If host variables contain complete SQL statements, then these statements are sent as ASCII or UNICODE character strings, depending on the type of the host variables. For performance reasons, only use UNICODE host variables for an SQL statement if the statement actually contains UNICODE characters.
- Run the precompiler [Page 61] with the precompiler option [Page 62] **`-G unicode`**.

```
EXEC SQL BEGIN DECLARE SECTION;
```

```
/* "SELECT TABLENAME FROM DOMAIN.TABLES " encoded in UCS2
*/
SQLUCS2 sqlstmt[36] = {0x0053, 0x0045, 0x004C, 0x0045,
0x0043,
0x0054, 0x0020, 0x0054, 0x0041, 0x0042,
0x004C, 0x0045, 0x004E, 0x0041, 0x004D,
0x0045, 0x0020, 0x0046, 0x0052, 0x004F,
0x004D, 0x0020, 0x0044, 0x004F, 0x004D,
0x0041, 0x0049, 0x004E, 0x002E, 0x0054,
0x0041, 0x0042, 0x004C, 0x0045, 0x0053,
0x0000};
SQLUCS2 resultstring[64];
EXEC SQL END DECLARE SECTION;
/* connect ... */
/* parse a unicode sql command and give it a statement name
*/
EXEC SQL PREPARE stmt1 FROM :sqlstmt;
EXEC SQL DECLARE curs1 CURSOR FOR stmt1;
EXEC SQL OPEN curs1;
/* loop over resultset */
while (sqlca.sqlcode != 100)
{
EXEC SQL FETCH curs1 INTO :resultstring;
/* ... */
}
EXEC SQL CLOSE curs1;
```

Working with LONG Columns

To make it easier to work with LONG columns, the C/C++ Precompiler offers the predefined data types SQLFILE [Page 22] and SQLLongDesc [Page 23], as well as the GETVAL statement [Page 50] and the PUTVAL statement [Page 49], for step-by-step processing of the columns.

However, you can also declare host variables [Page 15] for LONG columns as `char` arrays or VARCHAR [Page 21] data types.

```
EXEC SQL CREATE TABLE LONGTEST
(I1 INTEGER, L1 LONG BYTE, L2 LONG BYTE);
EXEC SQL BEGIN DECLARE SECTION;
VARCHAR l1[1000], l2[50];
int i1, il1, il2;
EXEC SQL END DECLARE SECTION;
```

```
EXEC SQL INSERT INTO LONGTEST VALUES
(:i1, :l1 :il1, :l2 :il2);
```

Example for using a descriptor [Page 39]:

```
sqlvartype *actvar;
EXEC SQL BEGIN DECLARE SECTION;
char *insert = "INSERT INTO LONGTEST VALUES (:I1, :L1,
:L2)";
EXEC SQL END DECLARE SECTION;
EXEC SQL PREPARE ins from :insert;
EXEC SQL DECLARE :C1 CURSOR FOR ins;
EXEC SQL DESCRIBE ins;
/* ... */
actvar = &sqlda.sqlvar[1];
(*actvar).hostvartype = sqlvchar;
/* ... */
EXEC SQL OPEN :C1 USING DESCRIPTOR;
```

Transferring NULL Values to the Database Instance

Use

In the application program, you can use indicator values [Page 20] to transfer a NULL value to a database table, for example, in an INSERT or UPDATE statement.

Procedure

Before calling the embedded SQL statement [Page 9], set the indicator value to `-1`. The database instance ignores the value of the corresponding host variable, and enters the NULL value in the table that is being addressed.

You can also use indicator values for transferring DEFAULT values to the database instance [Page 29].

Transferring DEFAULT Values to the Database Instance

Use

In the application program, you can use indicator values [Page 20] to transfer a DEFAULT value to a database table, for example, in an INSERT or UPDATE statement.

Procedure

Before calling the embedded SQL statement [Page 9], set the indicator value to the predefined constant `SQL_DEFAULT_PARAM`. The database instance ignores the value of the corresponding host variable, and enters the DEFAULT value in the table that is being addressed.

This procedure is not permitted for LONG columns.

You can also use indicator values for transferring NULL values to the database instance [Page 29].

Connecting to a Database Instance

Before your application program can execute SQL statements, the program must connect to the database instance and open a database session.

Use the statements for connecting to the database instance [Page 33]. Note the following:

- The statements for connecting to the database instance must be the first SQL statements executed by the application program. However, they do not have to be the first SQL statements in the source code.

  For the correct order of the SQL statements in the source code, see Conventions for the Order of SQL Statements [Page 10].

- One application program can have multiple sessions with one or more database instances. The database instance handles each session as an independent transaction. In the SQL mode [See SAP DB Library] INTERNAL, there can be a maximum of eight parallel sessions with different SAP DB database instances.

Overview of Precompiler Statements

You have the following options when you embed SQL statements [Page 9] in the source code of an application program:

- Static SQL statements [Page 35]
- Dynamic SQL statements [Page 37]
- ARRAY statements [Page 48]

The embedded statements consist of statements to the precompiler, and the actual statement that is sent to the database.

The key words `EXEC SQL` form the simplest precompiler statement. The static SQL statement `EXEC SQL <statement>` specifies that the database kernel executes the database statement `<statement>`.

This table gives you an overview of the precompiler statements and how they are used:

CANCEL statement [Page 54]	Cancels a running SQL statement
CONNECT statement [Page 34]	Statement for connecting to the database instance [Page 33]

DECLARE statements [Page 32]	Indicate declaration sections
DESCRIBE statement [Page 45]	Initializes a descriptor structure for a dynamic SQL statement
EXEC COMMAND statement [Page 54]	Executes an operating system command
EXECUTE statement [Page 46]	Executes a dynamic SQL statement
EXECUTE IMMEDIATE-statement [Page 37]	Calls a dynamic SQL statement without parameters
GETVAL statement [Page 50]	Reads LONG columns piecewise
INCLUDE DECLARE statement [Page 32]	Generates structure definitions
INCLUDE FILE statement [Page 49]	Inserts file content in the source text
OPEN CURSOR statement [Page 46]	Executes a dynamic SQL statement with named result table
PREPARE statement [Page 45]	Prepares a dynamic SQL statement
PUTVAL statement [Page 49]	Inserts values in LONG columns piecewise
SET SERVERDB statement [Page 33]	Statement for connecting to the database instance
TRACE statements [Page 55]	Activates logging in a trace file
USING clause [Page 47]	Assigns parameter values in a dynamic SQL statement
VERSION statement [Page 54]	Displays the software version
WHENEVER statements [Page 51]	Calls actions with conditions

See also: Syntax Directory [Page 75], Working with Multiple Database Sessions [Page 31], Compatibility with Other Database Systems [Page 59]

For the syntax of the database statements in the SQL mode INTERNAL, see the Reference Manual: SAP DB [See SAP DB Library].

Working with Multiple Database Sessions

When you work with multiple database connections [Page 33] in parallel, you must specify the relevant database session in your embedded SQL statements [Page 9]. To do this, specify the following in **<session_spec>**:

- In SQL mode INTERNAL, specify the session number **<session_number>**.
- In SQL mode ORACLE, specify the key word **AT** and the session name **<session_name>**.

See also: CONNECT Statement [Page 34] and Overview of Precompiler Statements [Page 30]

DECLARE Statements

Use

DECLARE statements mark the beginning and end of a declare section [Page 11].

Syntax

```
EXEC SQL BEGIN DECLARE SECTION;
EXEC SQL END DECLARE SECTION;
```

INCLUDE DECLARE Statement

Use

You can use the INCLUDE DECLARE statement to generate structure definitions [Page 26].

Prerequisites

- The precompiler must run [Page 61] with the precompiler option [Page 62] `check`, so that the application program gets the information it requires from the database instance.
- The file `<filename>` must not exist. If it exists, an INCLUDE FILE statement [Page 49] is executed instead of the INCLUDE DECLARE statement.

Syntax

```
EXEC SQL INCLUDE <file_name> <declare_clause>
[<as_clause>] [<ind_clause>];

<declare_clause> ::= <table_clause> | <dbproc_clause>

<table_clause>   ::= TABLE <table_name>

<dbproc_clause>  ::= DBPROC <dbproc_name>

<as_clause>      ::= AS VAR [<variable_declarator>]
|                    AS TYPE [<type_declarator>]
|                    AS STRUCT [<structure_tag>]

<ind_clause>     ::= IND [<ind_variable_declarator>] [<structure_tag>]
```

- **`AS VAR`** generates a variable declaration. **`AS TYPE`** generates a type declaration. **`AS STRUCT`** generates a structure declaration.
- If you do not specify an AS clause, the INCLUDE DECLARE generates a variable declaration by default.
- The default name of the structure definition is the name of the database table or procedure. However, you can also choose another name in the AS clause.
- The names of the structure components are derived from the column names of the table. The data types of the structure components are compatible with the data types of the corresponding table columns.
- You use the IND clause to generate another structure definition that you can specify as a type definition when you declare the indicator variables [Page 18].
- If you do not specify a name in **`<structure_tag>`**, then, by default, the name of the structure definition for the indicator variable is generated from the table name and preceded by an **`I`**.

- The names of the structure components for the indicator variable are, by default, derived from the table name and the column names, preceded by an **i**.
- The data types of the structure components are `short int`.
- If the table contains columns with the data type `long`, then the precompiler generates, as a structure component, a character string with 32767 characters for each of these columns, and displays a warning on the screen. You can then redefine or modify these structure components in the source code.

Statements for Connecting to the Database Instance

Use

Use these embedded SQL statements to connect to a database instance [Page 30] and open a database session.

Procedure

1. In the SQL mode [See SAP DB Library] INTERNAL, use the SET SERVERDB statement [Page 33] to specify the name of the database instance to which you want to connect, and the name of the server on which the database instance is running.

   In the SQL mode ORACLE, enter the name and server of the database instance in the USING clause of the CONNECT statement.

2. Use the CONNECT statement [Page 34] to specify the user name and password and open the database session. At the same time, you can choose options for the database connection.

SET SERVERDB Statement

Use

You use the SET SERVERDB statement to specify information on connecting to the database instance [Page 33] in the SQL mode [See SAP DB Library] INTERNAL. This includes defining which database instance on which server you want the application program to connect to.

Syntax

```
EXEC SQL [<session_number>] SET SERVERDB <database_name>
[ON <database_server>];
```

- You can also specify the name of the database instance **<database_name>** and the server **<database_server>** on which it runs in a host variable [Page 15] with the data type `char`.
- Enter **<session_number>** as the session number if you want your application program to open multiple database sessions in parallel. In the SQL mode INTERNAL, there can be a maximum of eight parallel sessions with different SAP DB database instances.

  You can override the information from the SET SERVERDB statement with the KEY in the CONNECT statement [Page 34].

When you run the precompiler [Page 61], you can also override the information from the SET SERVERDB statement for all database sessions with the precompiler options [Page 62] **-d** and **-n**.

CONNECT Statement

Use

Use the CONNECT statement to connect to the database instance [Page 33], and specify the user name and password for the database session.

Prerequisite

In the SQL mode INTERNAL, you have specified the name of the chosen database instance and server in the SET SERVERDB [Page 33] statement.

You can do without the SET SERVERDB statement if you choose the database instance and server by specifying a KEY in the CONNECT statement.

Syntax

CONNECT Statement in the SQL Mode INTERNAL [Page 34]

CONNECT Statement in the SQL Mode ORACLE [Page 35]

- When you run the precompiler [Page 61], you can use the precompiler option [Page 62] **-u** to override the user name and password specified in the CONNECT statement for the first database session. The values for all **subsequent** database sessions of the application program are, if possible, taken from the source code when the precompiler runs. A warning appears if these values are missing, and the embedded SQL statements of the subsequent database sessions are not checked.
- When you run the precompiler, you can use the precompiler option **-U** to override the KEY specified in the CONNECT statement of the first database session.

CONNECT Statement in the SQL Mode INTERNAL

Use

Use the CONNECT statement [Page 34] to connect to the database instance [Page 33], and specify the user name, password, and options for the database session.

Syntax

```
EXEC SQL [<session_number>] CONNECT
         [<user_name> IDENTIFIED BY <password> | <uidpwd>]
         [<connect_option>...] [KEY <key>];

<uidpwd> ::= <user_name>/<password>

<connect_option> ::= [ISOLATION LEVEL <unsigned_integer>]
                     [TIMEOUT <unsigned_integer>]
```

- Enter **<session_number>** as the session number if you want your application program to open multiple database sessions in parallel. In the SQL mode INTERNAL, there can be a maximum of eight parallel sessions with different SAP DB database instances.

- The user name **<user_name>** and password **<password>** can also be specified as a character string in a host variable [Page 15] (character or UNICODE data types [Page 24]).
- In **<connect_option>** you can specify an isolation level [See SAP DB Library] and timeout value [See SAP DB Library].
- You can specify the **<key>** in a host variable with data type **char**. You can specify an XUSER user key [See SAP DB Library], **DEFAULT** or the environment variable **SQLOPT**. For information on setting the connection options at runtime [Page 66] in this environment variable, see your operating system documentation.
- The values of the **<key>** complete or override the values from the SET SERVERDB statement [Page 33], as well as the explicitly specified user name, password, and CONNECT options.

CONNECT Statement in the SQL Mode ORACLE

Use

Use the CONNECT statement [Page 34] to connect to the database instance [Page 33], and specify the user name, password, and options for the database session.

Syntax

```
EXEC SQL CONNECT
         [<user_name> IDENTIFIED BY <password> | <uidpwd>]
         AT <session_name>
         USING <database_server-database_name>
         [<connect_option>...] [KEY <key>];

<uidpwd> ::= <user_name>/<password>

<connect_option> ::= [ISOLATION LEVEL <unsigned_integer>]
                     [TIMEOUT <unsigned_integer>]
```

- Choose a name for the database name in **<session_name>**. The only limit on the possible number of parallel database sessions in the SQL mode ORACLE is the maximum possible number of database sessions of the database instances.
- The user name **<user_name>** and password **<password>** can also be specified as a character string in a host variable [Page 15] (character or UNICODE data types [Page 24]).
- In **<connect_option>** you can specify an isolation level [See SAP DB Library] and timeout value [See SAP DB Library].
- You can specify the **<key>** in a host variable with data type **char**. You can specify an XUSER user key [See SAP DB Library], **DEFAULT** or the environment variable **SQLOPT**. For information on setting the connection options at runtime [Page 66] in this environment variable, see your operating system documentation.
- The **<key>** values complete or override the explicitly specified user name, password, and CONNECT options.

Static SQL Statements

Use dynamic SQL statements particularly if you already know the structures of the database tables during the programming phase, and the embedded SQL statements [Page 9] do not change at runtime of the application program. The C/C++ precompiler checks the syntax of

static SQL statements by sending them to the database kernel when you run the precompiler [Page 61] with the precompiler option [Page 62] `check`.

If the SQL statements or the structures of the database tables change at runtime of the program, use dynamic SQL statements [Page 37].

You can formulate the following types of static SQL statements:

- Static SQL statement without parameters [Page 36]
- Static SQL statement with parameters [Page 36]

Static SQL Statement Without Parameters

Use

One example of static SQL statements [Page 35] without parameters are DDL statements such as the CREATE TABLE statement. However, you can also formulate INSERTs, UPDATEs, DELETEs, and other statements.

Syntax

```
EXEC SQL [<session_spec>] <statement>;
```

`Specify` **`<session_spec>`**`, when you are working with` multiple database sessions [Page 31].

```
EXEC SQL CREATE TABLE hotel
(hno FIXED (4,0) KEY CONSTRAINT hno BETWEEN 1 AND 9999,
name CHAR(15) NOT NULL, zip FIXED (5,0) NOT NULL, loc
CHAR(20), price FIXED(6,2));

EXEC SQL INSERT INTO hotel VALUES (10,'Excelsior',
79000,'Atlanta', 135.00);

EXEC SQL INSERT INTO hotel VALUES (20,'Flora',
44000,'Philadelphia', 45.00);

EXEC SQL COMMIT WORK;
```

Static SQL Statement with Parameters

Use

The most common use of parameters in a static SQL statement [Page 35] is the use of host variables [Page 15] as parameters in a WHERE condition.

The host variables must be declared in the declaration section [Page 11].

You can use host variables in static SQL statements only in positions where the SQL syntax allows a parameter. For example, you **cannot** specify table names with host variables.

Syntax

```
EXEC SQL [<session_spec>] <statement>;
```

Specify **<session_spec>**, when you are working with multiple database sessions [Page 31].

```
EXEC SQL BEGIN DECLARE SECTION;
char name[16];
EXEC SQL END DECLARE SECTION;
sprintf (name, "Excelsior");
EXEC SQL SELECT name FROM hotel WHERE name = :name;
EXEC SQL FETCH INTO :name;
```

Dynamic SQL Statements

Use dynamic SQL statements if you do not know the structures of the database tables during the programming phase, and the embedded SQL statements [Page 9] can change at runtime of the application program.

If the SQL statements and table structures do not change at runtime, use static SQL statements [Page 35].

You can formulate the following types of dynamic SQL statements:

- Dynamic SQL statements without parameters [Page 37]
- Dynamic SQL statements with parameters [Page 38]

Dynamic SQL Statements Without Parameters

Use

Specify dynamic SQL statements [Page 37] without parameters as a character string **<statement_source>** in a host variable [Page 15] in the application program. The character string itself **cannot** contain any other host variable.

Call the dynamic SQL statement with the EXECUTE IMMEDIATE statement.

Syntax

```
EXEC SQL [<session_spec>] EXECUTE IMMEDIATE <statement_source>;
```

Specify **<session_spec>**, when you are working with multiple database sessions [Page 31].

```
EXEC SQL BEGIN DECLARE SECTION;
char statement [40];
EXEC SQL END DECLARE SECTION;
strcpy (statement, "INSERT HOTEL VALUES (80, 'Royal Plaza',
2000, \'Denver', 90.00)");
EXEC SQL EXECUTE IMMEDIATE :statement;
EXEC SQL EXECUTE IMMEDIATE 'COMMIT WORK';
```

Dynamic SQL Statements with Parameters

Use

You can set parameters for dynamic SQL statements [Page 37], so that any values unknown during the programming phase can be exchanged between the application program and the database instance.

Procedure

1. Replace the values that need to be exchanged with the placeholder ? in the statement.
2. Use the PREPARE statement [Page 45] to prepare the dynamic SQL statement for execution, and assign it a statement name.
3. Execute the dynamic SQL statement with the EXECUTE statement [Page 46] or OPEN CURSOR [Page 46] statement.
   Use the USING clause [Page 47] to assign values to the previously unknown parameters at runtime, for example, from a program variable of your choice. This does not have to be a host variable.
   You can execute a prepared SQL statement as often as you want, specifying new values as parameters each time.

If, during the programming phase, you do not know the table columns addressed by a dynamic SQL statement, and, therefore, the required parameters, then also use a descriptor [Page 39].

```
EXEC SQL BEGIN DECLARE SECTION;
int hno;
char name[16];
float price;
char cmd[100];
char *stmt;
EXEC SQL END DECLARE SECTION;
strcpy (cmd, "INSERT (INTO???) hotel VALUES(10,
'Excelsior', \
79000, 'Atlanta', 135.00)");
EXEC SQL PREPARE STMT1 FROM :cmd;
```

```
EXEC SQL EXECUTE STMT1;

hno = 10;

strcpy (cmd, "SELECT name, price INTO ?, ? \
FROM hotel WHERE hno=?");

stmt = "STMT2";

EXEC SQL PREPARE :stmt FROM :cmd;

EXEC SQL EXECUTE :stmt USING :name, :price, :hno;
```

Using the Descriptor

Use

If, during the programming phase, you do not know the table columns addressed by a dynamic SQL statement with parameters [Page 38], and, therefore, the required parameters, then use a descriptor.

At runtime, the application program determines the number, data types, and lengths of the table columns addressed by an SQL statement, and saves this information in the structure sqlda [Page 39] (SQL Descriptor Area, descriptor for short).

You can then use this information to assign the appropriate program variables to the parameters in the SQL statement.

See also: Example for Using a Descriptor [Page 44]

Prerequisite

You have used the PREPARE statement [Page 45] to prepare the dynamic SQL statement.

Procedure

1. Initialize the descriptor with the DESCRIBE statement [Page 45].
2. Use the information in the descriptor to assign appropriate program variables to the parameters in the dynamic SQL statement. To do this, enter the addresses of these program variables in the descriptor structure as well.

   Make sure that the program variables contain valid values at runtime.
3. Use the EXECUTE statement [Page 46] or the OPEN CURSOR statement [Page 46] with the USING clause [Page 47].

   If you use the SQL mode [See SAP DB Library] INTERNAL, the C/C++ precompiler defines an initial descriptor called `sqlda` automatically. In other modes, and if you want to use other descriptors, declare them as variables with the data type `sqlda`. This data type is defined in the header file `cpc.h`.

Structure of the Descriptor

This table describes the structure of the descriptor (the data structure `sqlda`) that you can use in dynamic SQL statements with parameters [Page 38] in the SQL mode [See SAP DB Library] INTERNAL.

The C/C++ precompiler also supports ORACLE `sqlda` data structures (see Compatibility with Other Database Systems [Page 59]).

sqldaid	Contains the character string `sqlda` for finding the structure in a memory dump
sqlmax	Maximum number of `sqlvar` entries The C/C++ precompiler specifies the constant `sqlmax = 300` for this value, however, you can change it.
sqln	Number of `sqlvar` entries currently allocated
sqld	Number of output parameters used in the SQL statement Any parameters that can be both input and output parameters are also counted as output parameters.
sqlvar	`array [sqlmax]` of sqlvar entries [Page 40] For each parameter, an entry `sqlvar [i]` is generated according to the order in which the parameter appears in the SQL statement. This entry contains information on the data type and the length of the addressed table column.

All constants used in the descriptor are declared in the header file `cpc.h`.

If you do not want to use the standard descriptor, you can declare the descriptor **<descriptor_name>** as a variable of the type `sqldatype` or `struct sqlda` (in SQL mode ORACLE, `SQLDA` only). However, the descriptor must always have the same type within a program.

sqlvar [i] Entries in the Descriptor

Each `sqlvar [i]` entry in the descriptor structure [Page 39] describes the properties of a parameter in the dynamic SQL statement [Page 37], as well as the properties of the required or assigned program variables.

Each `sqlvar [i]` entry consists of the following components:

colname	Contains the name of the table column for a FETCH statement. For all other SQL statements, `colname` contains the entry `columnx`, where `x` is a sequential number.
colio	Specifies whether the parameter is an input or output parameter. **`sqlinppar`: Input parameter** **`sqloutpar`: Output parameter** **`sqlinoutpar`: Input and output parameter**
colmode	Specifies whether NULL values are permitted for the parameter. `colmode` is a byte whose four low-order bits are interpreted as a bit mask.

	Bit 0: Mandatory **Bit 1: Optional (NULL permitted)** **Bit 2: Default** **Bit 3: Escape Character**
coltype	Specifies the SAP DB data type of the parameter. See: SAP DB Data Types in sqlvar Entries [Page 42]
collength	Specifies the length of the parameter or table column. For numerical data types, `collength` specifies the number of digits. For all other data types, `collength` specifies the number of characters. You can modify the value of `collength`, for example, if the closing NULL byte increases the number of characters in a character string by 1.
colfrac	Specifies the number of decimal places for numerical parameters or table columns. For the data type FLOAT/REAL `colfrac = -1`.
hostindicator	Contains the indicator value [Page 20], so performing the same task as an indicator variable [Page 18]. For input parameters, the user can set `hostindicator`. For output parameters, query `hostindicator`. If the table column contains a NULL value, the value of the program variable is not overwritten.
hostvartype	In `hostvartype`, specify the C/C++ data type of the program variable assigned to the parameter (see C/C++ Data Types in sqlvar Entries [Page 43]).  If necessary, modify the values of `collength` and `colfrac`. The DESCRIBE statement initializes `hostvartype` with the value `-1 (undefined)`.
hostcolsize	In `hostcolsize`, specify the size of an array element (in bytes) for array statements [Page 48].
hostvaraddr	In `hostvaraddr`, specify the address of the program variable assigned to the parameter. For array statements, specify the address of the first array element.

	The DESCRIBE statement initializes `hostvaraddr` with the NULL value.
hostindaddr	Contains the address of `hostindicator`. Instead of this, you can also specify the address of an indicator variable. The indicator value is then written to this variable, and the content of `hostindicator` is undefined. For array statements, specify the address of the first array element of the indicator variable. If you do not want to use an indicator value, specify NULL.
colinfo	Used for internal information on converting program variables. Do not change this value.

SAP DB Data Types in sqlvar Entries

This table shows the possible sqlvar entries [Page 40] `coltype` and the corresponding SAP DB data types [See SAP DB Library].

`coltype`	**SAP DB Data Type**
`sqlfixed`	FIXED
`sqlfloat`	FLOAT
`sqlchar`	CHAR
`sqlbyte`	CHAR (n) BYTE
`sqldate`	DATE
`sqltime`	TIME
`sqlexpr`	FLOAT
`sqltimestamp`	TIMESTAMP
`sqllong`	LONG
`sqllongbyte`	LONG BYTE
`sqlboolean`	BOOLEAN
`sqlunicode`	CHAR UNICODE
`sqlsmallint`	SMALLINT
`sqlinteger`	INTEGER
`sqlvarchar`	VARCHAR
`sqlvarbyte`	VARCHAR BYTE
`sqllongunicode`	LONG UNICODE
`sqlvarunicode`	VARCHAR UNICODE

C/C++ Data Types in sqlvar Entries

This table shows the possible sqlvar entries [Page 40] `hostvartype` and the corresponding C/C++ data types.

hostvartype	C/C++ Data Type
`sqlvint1`	`integer` (1 byte)
`sqlvint2`	`integer` (2 bytes)
`sqlvint4`	`integer` (4 bytes)
`sqlvint8`	`integer` (8 bytes)
`sqlvuns1`	`unsigned integer` (1 byte)
`sqlvuns2`	`unsigned integer` (2 bytes)
`sqlvuns4`	`unsigned integer` (4 bytes)
`sqlvuns8`	`unsigned integer` (8 bytes)
`sqlvreal4`	`float` (4 bytes)
`sqlvreal8`	`float` (8 bytes)
`sqlvchar`	`char array` (ends with null byte)
`sqlvcharp`	`char array` (filled up with blank characters)
`sqlvbyte`	`char array` (null bytes permitted and filled up with null bytes)
`sqlvstring1`	String with variable length `struct {len char; char array}`
`sqlvstring2`	String with variable length `struct {len short; char array}`
`sqlvstring4`	String with variable length `struct {len int; char array}`
`sqlvfile`	`char array` (filled up with blank characters) The value of this parameter must be a file name. See also SQLFILE [Page 22]
`sqlvfilec`	`char array` (ends with null byte) The value of this parameter must be a file name. See also SQLFILE [Page 22]
`sqlvlongdesc`	SQLLongDesc [Page 23]
`sqlvucs2`	UCS2 array (filled up with blank characters)
`sqlvutf16`	UTF16 array without surrogate (filled up with blank characters)
`sqlvunicode`	UTF8 array (filled up with blank characters)
`sqlvunicodec`	UTF8 array (ends with null byte)
`sqlvstringunicode`	UTF8 array with variable length `struct {len short; char array}`
`sqlvstringunicode4`	UTF8 array with variable length `struct {len int; char array}`

Example for Using a Descriptor

```
EXEC SQL BEGIN DECLARE SECTION;
char stmnt [255];
EXEC SQL END DECLARE SECTION;
strcpy (stmnt, "select hno, name, price from hotel");
EXEC SQL PREPARE SEL FROM :stmnt;
EXEC SQL EXECUTE SEL;
EXEC SQL PREPARE FET FROM 'FETCH USING DESCRIPTOR';
EXEC SQL DESCRIBE FET;
/* Bind each column to a piece of memory */
for (i=0;i<sqlda.sqln;i++) {
  sqlvartype *sqlvar = &sqlda.sqlvar[i];
  switch(sqlvar->coltype){
    case (sqlfixed) : {
      if (sqlvar->colfrac == 0) {
        sqlvar->hostvartype = sqlvint4;
        sqlvar->hostcolsize = sizeof(int);
        sqlvar->hostvaraddr = malloc(sizeof(int));
      }
      else {
        sqlvar->hostvartype = sqlvreal4;
        sqlvar->hostcolsize = sizeof(float);
        sqlvar->hostvaraddr = malloc(sizeof(float));
      }
      break;
    }
    case (sqlchar) : {
      sqlvar->hostvartype = sqlvchar;
      sqlvar->hostcolsize = sqlda.sqlvar[0].collength;
      sqlvar->hostvaraddr = malloc(sizeof(int));
      break;
    }
    default : {
      /* TODO: support more datatypes */
    }
    }
    }
    /* Process result table */
    EXEC SQL EXECUTE FET USING DESCRIPTOR;
    while (sqlca.sqlcode == 0) {
      EXEC SQL EXECUTE FET USING DESCRIPTOR;
    }
```

PREPARE Statement

Use

Use the PREPARE statement to prepare a dynamic SQL statement with parameters [Page 38] for execution, and assign it a statement name.

Syntax

```
EXEC SQL [<session_spec>] PREPARE <statement_name>
         [INTO <descriptor_name> [USING <using_clause>]]
         FROM <statement_source>;
```

- Specify **<session_spec>**, when you are working with multiple database sessions [Page 31].
- In **<statement_name>**, specify the statement name either in a host variable [Page 15] or as a constant character string.
- In **<statement_name>**, specify the dynamic SQL statement either in a host variable or as a constant character string.

In the PREPARE statement, the **<descriptor_name>** and the **<using_clause>** have no meaning in the SQL mode [See SAP DB Library] INTERNAL. They are used for DB2 compatibility.

DESCRIBE Statement

Use

Use the DESCRIBE statement to initialize the descriptor structure [Page 39], and to determine the parameter and table column information needed for a dynamic SQL statement with descriptor [Page 39].

Prerequisite

You have used the PREPARE statement [Page 45] to prepare the SQL statement for execution.

Syntax

```
EXEC SQL [<session_spec>] DESCRIBE <statement_name>
[INTO <descriptor_name> [<using_clause>]];
```

- Specify **<session_spec>**, when you are working with multiple database sessions [Page 31].
- If the INTO clause is not specified, the precompiler uses the standard descriptor `sqlda`.
- If you do not want to use the standard descriptor, you can declare the descriptor **<descriptor_name>** as a variable of the type `sqldatype` or `struct sqlda` (in SQL mode ORACLE, `SQLDA` only). However, the descriptor must always have the same type within a program.

Note

In the SQL mode [See SAP DB Library] INTERNAL, the **<using_clause>** has no meaning in the DESCRIBE statement. It is used for DB2 compatibility.

EXECUTE Statement

Use

Use the EXECUTE statement to execute a dynamic SQL statement with parameters [Page 38].

Prerequisites

- You have used the PREPARE statement [Page 45] to prepare the SQL statement for execution.
- If you are using a descriptor [Page 39], you have initialized it with the DESCRIBE statement [Page 45].

Syntax

```
EXEC SQL [<session_spec>] [<for_clause>] EXECUTE <statement_name>
[<using_clause>];
```

- Specify **<session_spec>**, when you are working with multiple database sessions [Page 31].
- Use the USING clause [Page 47] to assign values to the parameters.
- Use the FOR clause for array statements [Page 48].

OPEN CURSOR Statement

Use

You can use the OPEN CURSOR statement to execute a dynamic SQL statement with parameters [Page 38], and generate a named result table (cursor).

Prerequisites

- You have used the PREPARE statement [Page 45] to prepare the SQL statement for execution.
- If you are using a descriptor [Page 39], you have initialized it with the DESCRIBE statement [Page 45].
- You have already defined the cursor (the named result table) with a DECLARE CURSOR statement [See SAP DB Library].

Syntax

```
EXEC SQL [<session_spec>] [<for_clause>] OPEN <cursor_name>
         [ USING <parameter_list>
         | USING DESCRIPTOR [<descriptor name>] [KEEP]
         | INTO <parameter_list>
         | INTO DESCRIPTOR [<descriptor name>] [KEEP]]
```

- If, in the programming phase, you do not know how many different result tables are processed by your application program, you can specify the result table name **<cursor_name>** in a host variable [Page 15] in the OPEN CURSOR statement.

- Specify **KEEP** if you want to follow the OPEN CURSOR statement with PUTVAL statements [Page 49].
- Here, **INTO** has the same meaning as specifying **USING <parameter_list>**. It guarantees compatibility with other database systems.

```
EXEC SQL BEGIN DECLARE SECTION;
int hno;
char name[16];
float price;
char cmd[100];
char *stmt;
char *cursor;
EXEC SQL END DECLARE SECTION;
strcpy (cmd, "INSERT (INTO???) hotel VALUES(10,
'Excelsior', \
79000, 'Atlanta', 135.00)");
EXEC SQL PREPARE STMT1 FROM :cmd;
EXEC SQL DECLARE cur1 CURSOR FOR STMT1;
EXEC SQL OPEN cur1;
hno = 10;
strcpy (cmd, "SELECT name, price FROM hotel WHERE hno=?");
stmt = "STMT2";
EXEC SQL PREPARE :stmt FROM :cmd;
cursor = "cur2";
EXEC SQL DECLARE :cursor CURSOR FOR :stmt;
EXEC SQL OPEN :cursor USING :hno;
EXEC SQL FETCH :cursor INTO :name, :price;
```

USING Clause

Use the USING clause for dynamic SQL statements with parameters [Page 38] to assign values to the parameters in EXECUTE [Page 46] or OPEN CURSOR statements [Page 46].

Prerequisites

- You have used the PREPARE statement [Page 45] to prepare the SQL statement for execution.
- If you are using a descriptor [Page 39], you have initialized it with the DESCRIBE statement [Page 45].

Syntax

```
<using_clause> ::=
  USING <parameter_list>
| USING DESCRIPTOR [<descriptor_name>]
| INTO <parameter_list>
| INTO DESCRIPTOR [<descriptor_name>]
```

- If you do not specify **<descriptor_name>**, the standard structure `sqlda` is used.
- Here, **INTO** has the same meaning as specifying **USING**. It guarantees compatibility with other database systems.

Array Statement

Use

Use array statements to read or change more than one data record in a data table with only a single embedded SQL statement [Page 30]. An array statement executes an SQL statement multiple times in a loop, and applies each array element as a parameter in a row of the database table.

You have the following options when you formulate an array statement:

- You specify all parameters in a static SQL statement [Page 35] in array host variables [Page 17].
- You use the FOR clause together with a descriptor and array program variables in a dynamic SQL statement [Page 37].

Syntax

```
EXEC SQL [<session_spec>] [FOR <loop_parameter>] <sql_statement>;
```

- Specify **<session_spec>**, when you are working with multiple database sessions [Page 31].
- The value of **<loop_parameter>** detemines the number of loop repeats. If the array variable has more or fewer components than specified in **<loop_parameter>**, a warning appears in the precompiler log [Page 64] and the statement is executed with the lowest dimension. This also applies if you use multiple array variables that do not have the same number of elements.

If errors occur in the database kernel while the array statement is being executed, the statement is terminated, and the number of successfully processed records is entered under **sqlerrd[2]** in the structure sqlca [Page 55].

The SQL statement SELECT INTO is not permitted for array statements in the SQL mode [See SAP DB Library] INTERNAL.

Example

Example of a static array statement:

```
/* Array Insert */
EXEC SQL BEGIN DECLARE SECTION;
int hno[3];
int zip[3];
float price[3];
char *name[3], *loc[3];
EXEC SQL END DECLARE SECTION;
```

```
hno[0]   = 10;
name[0]  = "Excelsior";
zip[0]   = 89073;
loc[0]   = "Philadelphia";
price[0] = 135.00;
hno[1]   = 30;
name[1]  = "Flora";
zip[1]   = 48159;
loc[1]   = "Atlanta";
price[1] = 45.00;
hno[2]   = 20;
name[2]  = "Continental";
zip[2]   = 86165;
loc[2]   = "Cincinnati";
price[2] = 70.00;
EXEC SQL INSERT INTO HOTEL VALUES (:hno, :name, :zip, :loc,
:price);
```

INCLUDE FILE Statement

Use

You can use the INCLUDE FILE statement to include source code from a file **<file_name>** in your application program. The precompiler places this text at the location of the INCLUDE FILE statement in the source code. The text itself cannot contain an INCLUDE FILE or INCLUDE DECLARE Statement [Page 32].

Syntax

EXEC SQL INCLUDE <file_name>;

If necessary, you can use **EXEC SQL INCLUDE sqlca** to select the position in the source code at which you want the precompiler to declare the structure sqlca [Page 55]. This statement ensures compatibility with other embedded SQL compilers. If you do not select a position, the precompiler places the declaration at the start of the program.

PUTVAL Statement

Use

You can use PUTVAL statements to insert data in a LONG column [Page 28] piecewise.

The PUTVAL statements must follow a INSERT statement [See SAP DB Library]. Each PUTVAL statement that follows this INSERT statement appends the current content of the corresponding host variable [Page 15] to the LONG column of the database table.

Prerequisite

You have started to insert the data in the LONG column with an INSERT statement.

Syntax

```
EXEC SQL [<session_spec>] PUTVAL INTO <table_name>
         [(<column_name>,...)] VALUES (<parameter_list>)
```

- Apart from the key word **PUTVAL**, the PUTVAL statement must be the same as the preceding INSERT statement. For example, the order and number of the parameters in the parameter list must be the same as the parameter list **<parameter_list>** of the INSERT statement.
- Before each PUTVAL statement, assign the new data to the host variable for the LONG column.
- The PUTVAL statement ignores any values in the parameter list that refer to table columns other than the LONG column.
- You cannot execute any other SQL statements between the INSERT statement and the subsequent PUTVAL statement, or between the PUTVAL statements themselves.

  Every other SQL statement closes the LONG column in the database table, so that no more data can be appended.
- Inform yourself about the special features when using the data type SQLLongDesc [Page 24].
- If you want to insert more than one row in a table, use an array statement [Page 48].

```
EXEC SQL CREATE TABLE LONGTEST
(I1 INTEGER, L1 LONG BYTE, L2 LONG BYTE);

EXEC SQL BEGIN DECLARE SECTION;

  int array_size;

  VARCHAR l1[100][500], l2[100][50];

  int i1[100];

EXEC SQL END DECLARE SECTION;

array_size = 100;

EXEC SQL FOR :array_size INSERT INTO LONGTEST VALUES
(:i1, :l1 :il1, :l2 :il2);
```

- If you do **not** want a PUTVAL statement to insert any data in a certain LONG column, then assign the predefined constant `SQL_IGNORE` to the appropriate indicator value [Page 20] in this statement.

  You cannot insert LONG data with the length NULL.

GETVAL Statement

Use

You can use the GETVAL statement to read data piecewise from a LONG column [Page 28].

The GETVAL statements must follow a FETCH statement [See SAP DB Library]. Each GETVAL statement that follows this FETCH statement reads the next part of the content from the LONG column of the database table from the current position.

Prerequisite

You have started to read the data from the LONG column with a FETCH statement.

Syntax

```
EXEC SQL [<session_spec>] GETVAL INTO (<parameter_list>)
```

- Apart from the key word **GETVAL**, the GETVAL statement must be the same as the preceding FETCH statement. For example, the order and number of the parameters in the parameter list must be the same as the parameter list **<parameter_list>** of the FETCH statement.
- You cannot execute any other SQL statements between the FETCH statement and the subsequent GETVAL statement, or between the GETVAL statements themselves.
- If the host variable [Page 15] that corresponds to the LONG column has the data type SQLLongDesc [Page 23], then you can read the total length of the LONG column in the corresponding structure component `cbColLen` after the GETVAL statement has been executed.
- You can determine how often you need to execute the GETVAL statement by querying the warning message [Page 57] `sqlwarn1`. If `sqlwarn1` has the value `W`, then the LONG column has not yet been read completely.

  After the FETCH statement has been executed, the remaining content of the LONG column is read by the following statement:

  ```
  while (sqlca.sqlwarn[1] == 'W')
  EXEC SQL GETVAL INTO :i1, :l1, :l2;
  ```

- If you do not want a GETVAL statement to read the data of a certain LONG column, then assign the predefined constant `SQL_IGNORE` to the appropriate indicator value [Page 20] in this statement.

WHENEVER Statements

Use

You can use WHENEVER statements to program actions that you want to be executed for each subsequent embedded SQL statement [Page 9].

You have the following options:

- You program actions that are executed **before** or **after** each SQL statement.
- You program error handling [Page 52] actions.

The WHENEVER statement must be located in the source code of the application program in front of the SQL statements that it handles. This is so that the C/C++ precompiler [Page 60] can generate the appropriate individual statements when the precompiler runs [Page 61]. The WHENEVER statement is valid for all subsequent SQL statements in the source code until another WHENEVER statement for the same condition takes over, or until the program ends.

Syntax

```
EXEC SQL WHENEVER <condition> <action>;
```

Where **<condition>** is an error (see *Handling Errors with WHENEVER Statements*), or one of the following conditions:

SQLBEGIN	The specified **<action>** is executed before each SQL statement.
SQLEND	The specified **<action>** is executed after each SQL statement.

For **<action>**, specify one of the possible actions for the WHENEVER statement.

Example

Example for Using WHENEVER Statements [Page 53]

Handling Errors with WHENEVER Statements

Use

You can use WHENEVER statements to program standard error handling procedures for errors and exceptions that can occur when embedded SQL statements [Page 9] are executed. You can check four different types of database system messages [Page 55], and choose from four standard reactions to these messages.

Use a WHENEVER statement if, for example, the application program does not query the values of the messages `sqlcode` or `sqlwarn0` after an SQL statement.

Syntax

```
EXEC SQL WHENEVER <condition> <action>;
```

Where **<condition>** is one of the following cases:

SQLWARNING	`sqlwarn0` has the value `W`. At least one warning exists.
SQLERROR	`sqlcode` has a negative value. An error has occurred.
NOT FOUND	`sqlcode` has the value 100 (ROW NOT FOUND). No table row found.
SQLEXCEPTION	`sqlcode` has a positive value greater than 100. An exception has occurred.

For **<action>**, specify one of the possible actions for WHENEVER statements [Page 53].

If **SQLERROR** occurs, the application program must terminate, and the error must be analyzed.

If you call an error handling function **<action>** that also contains SQL statements, and these SQL statements can send error messages, then start the function with a WHENEVER CONTINUE statement to avoid endless loops.

Example

```
EXEC SQL WHENEVER SQLEND CALL CheckSQLCode(&sqlca);

EXEC SQL SELECT DIRECT NAME, LOC
         INTO :name, :loc FROM HOTEL KEY HNO = :hno;
```

```
void CheckSQLCode(sqlcatype *sqlca)
{
  if (sqlca->sqlcode != 0) {
    code = sqlca->sqlcode;
    EXEC SQL WHENEVER SQLERROR CONTINUE;
    EXEC SQL INSERT INTO ERROR VALUES (:code);
  }
}
```

Actions for the WHENEVER Statement

Use WHENEVER statements [Page 51] to call the following actions **<action>**:

STOP	Resets the running transaction correctly, ends the database connection with the RELEASE statement [See SAP DB Library] `COMMIT WORK RELEASE` and terminates the program
CONTINUE	Continues to run the application program. Use **CONTINUE** to deactivate a previous WHENEVER action for this condition.
GOTO <label>	The application program jumps to the label **<label>**. **<label>** can have a maximum of 45 characters.
CALL <function>	The application program calls the function **<function>**. The function call can have a maximum of 50 characters, including the parameters used.

Example for Using WHENEVER Statements

The following examples show the usage of different WHENEVER [Page 51] statements.

```
EXEC SQL WHENEVER SQLWARNING CALL warnproc();
EXEC SQL SELECT * FROM dual;
/* receives a warning and calls warnproc */

EXEC SQL WHENEVER SQLERROR CALL errproc();
EXEC SQL DELETE FROM unknowntable;
/* receives an error and calls errorproc */

EXEC SQL WHENEVER SQLERROR STOP;
/* receives also an error but stops execution */
EXEC SQL DELETE FROM unknowntable;
```

CANCEL Statement

Use

Use the CANCEL statement to cancel the processing of an SQL statement by the database kernel. This means that the SQL statement has no effect. If successful, the CANCEL statement has the return code sqlcode 0. The canceled statement has the sqlcode –102.

Syntax

```
EXEC SQL CANCEL [<cancel_session>];

<cancel_session> ::= <session_number> | <session_name> | CURRENT
```

EXEC COMMAND Statement

Use

You can use the EXEC COMMAND statement to execute operating system commands, for example, to delete files or call the Database Manager CLI [See SAP DB Library].

The same general rules [Page 9] apply to EXEC COMMAND statements as to SQL statements. Precede each system call with the key words **EXEC COMMAND** .

Syntax

```
EXEC COMMAND SYNC <command> RESULT <result_parameter>
| EXEC COMMAND ASYNC <command>;

<command> ::= <string_constant> | <host_variable>

<result_parameter> ::=  <host_variable>
```

- Declare **<result_parameter>** as a host variable [Page 15] with the type `int` (2 bytes).
- In **SYNC** mode, the application program stops until the operating system command has been completed.
- In **ASYNC** mode, the operating system command is executed in the background.
- For information on the correct syntax of the operating system command, see your operating system documentation.

VERSION Statement

Use

You can use the VERSION statement to display the versions of the general SAP DB runtime environment and the precompiler runtime environment.

Syntax

```
EXEC SQL VERSION <runtime_version>, <precompiler_version>;
```

You can specify **<runtime_version>** and **<precompiler_version>** as host variables [Page 15] with the data type `char` and a length of 40 bytes.

TRACE Statements

Use

You can use the TRACE statements to specify that the precompiler runtime environment [Page 65] logs the execution of individual embedded SQL statements in the trace file `<filename>.pct` at runtime of the application program.

Procedure

Include the SQL statements that you want to log in the statements **`SET TRACE ON`** or **`SET TRACE LONG`** and **`SET TRACE OFF`**.

Any SQL statements that appear in external subroutines or modules called by the included SQL statements are also logged.

To activate logging for the **whole** application program, set your chosen trace option [Page 66] when you run the precompiler or call the program. The trace option then overrides the individual TRACE statements.

Syntax

`EXEC SQL SET TRACE ON;`	Activates logging with the trace option TRACE SHORT Every executed SQL statements is logged, including the messages `sqlcode` und `sqlerrd[2]`.
`EXEC SQL SET TRACE LONG;`	Activates logging with the trace option TRACE LONG Every executed SQL statements is logged, including the messages `sqlcode` und `sqlerrd[2]` and all parameter values.
`EXEC SQL SET TRACE OFF;`	Deactivates logging
`EXEC SQL SET TRACE LINE <trace_line>;`	The content of **`<trace line>`** is written to the trace file as a comment. You can specify **`<trace line>`** as a character string or in a host variable [Page 15].

Database System Messages

Database system messages and warnings are sent using the global structure `sqlca` (SQL Communication Area) The precompiler [Page 60] includes this structure automatically in every SAP DB application program. At runtime of the application program, the database saves information on the execution of the last embedded SQL statement [Page 9] in the structure components, for example, information on any errors that occurred.

Query the structure component `sqlcode` after every SQL statement.

The `sqlca` components have the following meaning:

`sqlcaid`	`char [8]` Contains the character string *sqlca* and is used

	to find the `sqlca` when analyzing a memory dump
sqlcabc	`integer` (4 byte) Specifies the length of the `sqlca` in bytes
sqlcode	`integer` (4 byte) Contains the message code of the database system message `sqlcode = 0` indicates that a statement was executed successfully. `sqlcode > 0` indicates problems that must be handled in the program. `sqlcode < 0` indicates errors that terminated the program. For descriptions of all the messages, see Messages: SAP DB [See SAP DB Library]..
sqlerrml	`integer` (2 byte) Specifies the current length of `sqlerrmc`
sqlerrmc	`char [70]` Contains an explanation of the message if `sqlcode` has a value other than 0
sqlerrd	Array of six 4 byte `integer` numbers `sqlerrd [2]` specifies how many table rows were processed by the SQL statement. `-1` indicates that the number of processed rows is unknown. If there are errors in array statements [Page 48], `sqlerrd [2]` contains the number of the last row that was processed without errors. `sqlerrd [5]` specifies the runtime of the statement in the database kernel (in seconds). `sqlerrd [6]` specifies the position in the SQL statement at which a syntax error was found. This value is undefined for array statements. If the SQL statement did not have any errors, then `sqlerrd [6] = 0`.
sqlwarn0	`char [1]` `sqlwarn0 = W` specifies that the SQL statement generated at least one of the warnings `sqlwarn1..sqlwarnc`. If no warning exists, then `sqlwarn0` contains a blank character.
sqlwarn1..sqlwarnc	Warning messages in the structure sqlca [Page 57]
sqlresn	`char [18]` Contains the name of the result table after a SELECT statement is called

	`sqlresn` contains a blank character after all other calls
sqldatetime	`integer` (2 byte) Specifies which date and time format [See SAP DB Library] the data types DATE, TIME und TIMESTAMP have. `1` INTERNAL `2` ISO `3` USA `4` EUR `5` JIS

All other `sqlca` components are used for internal purposes.

Warning Messages in the Structure sqlca

This table shows you the warning messages that can occur as database system messages [Page 55].

Each of the `sqlca` components `sqlwarn1..sqlwarnc` has the value `W` if the execution of an embedded SQL statement [Page 9] generates the corresponding warning.
If the corresponding warning does not exist, the component contains a blank character.

sqlwarn1	`char [1]` Specifies whether a character string of the type `char` was truncated when a host variable [Page 15] and the database exchanged values. If you use an indicator variable [Page 18], and a warning occurs, the variable specifies the length of the original character string.
sqlwarn2	`char [1]` Specifies whether COUNT, MAX, MIN, SUM, AVG, STDDEV or VARIANCE NULL values were found in the database table and ignored by the database kernel when the SET functions [See SAP DB Library] were executed
sqlwarn3	`char [1]` Specifies whether the number of result columns of a SELECT or FETCH statement did not match the number of host variables in the INTO clause
sqlwarn4	`char [1]` Specifies whether an UPDATE or DELETE statement was performed without a WHERE clause and, therefore, in the entire database table
sqlwarn6	`char [1]` Specifies whether the values of a date (DATE or TIMESTAMP) needed to be corrected

	 You see this warning for incorrect dates such as 31.2 or 37.7.
sqlwarn8	`char [1]` Specifies whether all table rows needed to be searched to generate a result table
sqlwarnb	`char [1]` Set if a TIME value is greater than 99 (or greater than 23 in the USA format) The value is corrected to modulo 100 (or 24).
sqlwarnc	`char [1]` For a SELECT statement, specifies whether more rows were found than permitted by the ROWNO predicate [See SAP DB Library] in the WHERE clause

Programming Notes

To optimize your SAP DB application program, remember the following programming notes:

- In SELECT statements [See SAP DB Library], identify the columns that you want to select.
- Use key specifications [See SAP DB Library] to identify the table rows that need to be processed. You can determine key columns from the system tables [See SAP DB Library].
- SELECT statements with multiple result rows with no ORDER clause [See SAP DB Library] do not specify the order in which the table rows are read from the database table. In this case, the logic of your application program cannot assume that there is a specified order.,
- You can use the SINGLE SELECT statement [See SAP DB Library] to process individual rows in the SQL mode INTERNAL.
- Note the different way of handling named result tables [See SAP DB Library] (CURSOR) in the different SQL modes [See SAP DB Library].

  For information on the special features of the SQL mode ORACLE in the SAP DB database system, see the documentation SQL Mode ORACLE: SAP DB [See SAP DB Library].
- Inform yourself about the user and role concept [See SAP DB Library] of the SAP DB database system.
- Execute a RELEASE statement [See SAP DB Library] as the final SQL statement in your application program. This releases the resources needed by the precompiler runtime environment [Page 65].
- In the SAP DB database system, remember that DDL statements such as CREATE TABLE are also subject to the transaction concept [See SAP DB Library], which means that they can be undone or that they must be completed with a COMMIT statement [See SAP DB Library].
- Read the information about the locks [See SAP DB Library] in the database, and define a suitable isolation level [See SAP DB Library] for your statements.

- Program reactions to possible TIMEOUT messages. You can do this by, for example, querying the structure component `sqlcode` after each SQL statement (see Database System Messages [Page 55]).

  For more information on the meaning of the optimum SQL statements and the available options, see Optimizer: SAP DB 7.4 [See SAP DB Library].

Compatibility with Other Database Systems

The SAP DB database system can execute database statements that are written in other SQL modes [See SAP DB Library].

You specify the SQL mode of your application program in the precompiler options [Page 62] when you run the precompiler [Page 61].

Inform yourself about the special features in the SQL mode ORACLE [Page 59] and the SQL mode ANSI [Page 60].

If you only want to create individual embedded SQL statements in another SQL mode, then precede them with the following key words:

- **`EXEC SAPDB`** or **`EXEC INTERNAL`** for individual statements in the INTERNAL mode
- **`EXEC ORACLE`** for individual statements in the ORACLE mode
- **`EXEC ANSI`** for individual statements in the ANSI mode

Special Features in SQL Mode ORACLE

- The isolation levels [See SAP DB Library] 0, 1, 2, 3 are permitted in the SQL mode ORACLE. The default isolation level is 1.
- You need to define the appropriate `sqlda` structures for dynamic SQL statements [Page 37]. They are compatible with the ORACLE `sqlda` structures, but contain additional internal SAP DB information. Use the appropriate INCLUDE files that are included automatically when the precompiler runs [Page 61].

  ```
  /*Declaration*/
  SQLDA orada;
  /*Initialization*/
  orada = sqlald (max_vars, max_name, max_ind_name);
  ```

  where `max_vars` indicates the number of parameters, `max_name` the maximum number of characters in the parameter names, and `max_ind_name` the maximum number of characters in the indicator names.
- With a few exceptions, the return code [Page 55] `sqlcode` of the database system is copied from SAP DB (see Messages: SAP DB [See SAP DB Library]).

This table shows the exceptions and the corresponding SAP DB message codes:

ORACLE Message Code	SAP DB Message Code

-1008	Warnung3
-1007	-804
-1034	-813
0	-4000
Warnung1	-743
100	1403 ROW NOT FOUND

Special Features in SQL Mode ANSI

- The isolation levels [See SAP DB Library] 0, 1, 2, 3 are permitted in the SQL mode ANSI. The default isolation level is 3.
- Dynamic SQL statements [Page 37] in ANSI mode use the SAP DB `sqlda` structure that is included automatically when the precompiler runs.
- With a few exceptions, the return code [Page 55] `sqlcode` of the database system is copied from SAP DB (see Messages: SAP DB [See SAP DB Library]).

This table shows the exceptions and the corresponding SAP DB message codes:

ANSI Message Code	SAP DB Message Code
-250	+250
-300	+300
-320	+320
ANSISTATE 01004	sqlwarn1
ANSISTATE 01003	sqlwarn2

The warning messages `sqlwarn1` and `sqlwarn2` appear in the structure sqlca [Page 57].

Functions of the C/C++ Precompiler

When you run the precompiler [Page 61], the C/C++ precompiler translates the SQL statements embedded [Page 9] in the application program into calls of procedures of the precompiler runtime environment [Page 65], and prepares the source code for the C/C++ compiler

The precompiler performs the following tasks:

- Generates the descriptor structure [Page 39] `sqlda`
- Generates the structure `sqlca` for database system messages [Page 55]
- Includes the standard header file `cpc.h`
- Checks the syntax of the embedded SQL statements
- Generates the precompiler log [Page 64] with any error messages and warnings

Precompiler Option check

If you choose the precompiler option [Page 62] `check` , the precompiler checks the existence of the called database tables and columns, as well as the compatibility of the data types and the access privileges of the users. It does this by sending the embedded SQL statements to the parser or database kernel. A ROLLBACK [See SAP DB Library] resets the effects of all SQL statements executed in this way after the precompiler has completed its run.

Only the syntax can be checked for statements with components unknown to the precompiler. These can be, for example, dynamic SQL statements [Page 37] or CONNECT statements with host variables [Page 15].

The SQL statements are executed in the order in which they appear in the source code, and not in the logical order of the program statements.

Connecting to a Database

Before it can check the embedded SQL statements (with the `check` option), the precompiler must connect to the database instance. The following rules apply:

- If the first static SQL statement of a database session is not a CONNECT statement [Page 34], then all user data of the XUSER user key [See SAP DB Library] `DEFAULT` applies.
- If the first static SQL statement of a database session is a CONNECT statement, then all data is taken from this statement, and any missing data from the XUSER user key `DEFAULT`.
- In the first database session, you can use appropriate precompiler options to override all user data.

Running the Precompiler

To check the source code file **`<filename>.cpc`** of your application program with the C/C++ Precompiler [Page 60], and so make it executable, perform the following tasks:

1. Use the following call to start the precompiler and compiler from the command line:
   `cpc <precompiler_options> <filename> <compiler_options>`
   Specify your preferred precompiler options [Page 62] and compiler options. The precompiler generates a C file `<filename>.c` or a C++ file`<filename>.cpp`, depending on the chosen precompiler options.
2. Translate this file with your C/C++ compiler. The compiler generates an object file. Depending on your chosen precompiler options, the compiler can be started at the same time as the precompiler, so that the object file can be generated from your source code in a single step.
3. Combine this object file with other object files and libraries until you have an executable program. To do this, call the precompiler linker **`cpclnk`** with your chosen options [Page 62], or include the precompiler runtime environment `libpcrl` manually.

See also Example: Compiling a SAP DB Application Program [Page 65]

Call Options for the Precompiler Linker

After running the precompiler [Page 61] and compiling your application program, call the precompiler linker with the following options:

Options for Microsoft Windows Operating Systems

```
cpclnk [-<sdkver>] [-BIT32|-BIT64] [-MT|-MD|-SMT] [<link_options>]
<file_name> [<object_archive>]
```

- If you have installed multiple versions of the precompiler software, choose a version with **<sdkver>**.

  For example, enter **-7402**, if you have installed Version 7.4.02.

- Use the options **-BIT32|-BIT64** and **-MT|-MD|-SMT** to specify the runtime libraries that you want the linker to use.
- Specify your linker options in **<link_options>**. For information on your linker, see your operating system documentation.
- Choose a name for the executable program in **<file_name>**.
- Specify the object files in **<object_archive>**. You can also specify the relevant path.

Options for UNIX Operating Systems

```
cpclnk [-P] [-s] [<link_options>] <file_name> [<object_archive>]
```

- Use option **-P** to call the C++ linker.
- Use option **-s** if you want to link the object files statically.

Precompiler Options

This table shows all the options you can specify when you call the precompiler [Page 61].

-E cansi	Generates ANSI C-compatible code
-E cplus	Generates C++-compatible code
-BIT32	Generates 32 bit-compatible code If you do not specify this option, the precompiler generates 64 bit-compatible code, if possible.
-m <m_begin,m_end>	Defines the valid column range in which the precompiler interprets source code If you do not specify this option, the following defaults apply: **m_begin = 1** **m_end = 132**
-e	Generates a program module that can be combined with other modules Only translate the main module **without** this option, all other modules must be compiled with this option.
-H <check_option>	Sets the chosen check option for the

<check_option> = nocheck|syntax|check	precompiler **nocheck**: the precompiler checks only the syntax of the `exec sql` commands for the embedded SQL statements **syntax**: the precompiler also checks the syntax of the database statements themselves **check**: the precompiler also checks the existence of the called database tables and columns, their compatibility with the data types of the corresponding host variables [Page 15], and the access privileges of the specified database user If you do not specify a check option, the precompiler uses `check` automatically.
-b <maxpacketsize>	Restricts the size of the request buffer for SQL statements when the precompiler runs The default when the precompiler runs is `<maxpacketsize> = 16000` At runtime of the application program, the request buffer is restricted by the support database parameter [See SAP DB Library] _PACKAGE_SIZE of the database instance that is being addressed.
-G unicode	Specifies that the generic precompiler data types, such as TCHAR and LPSTR, are converted to UNICODE data types such as WCHAR and LPWSTR At the same time, the precompiler integrates the macro SAPDB_WITH_UNICODE with the source code. If you do not specify this option, then the generic data types are converted to ANSI data types (CHAR, CHAR*).
-s	Suppresses the screen output of precompiler messages
-w	Suppresses the output of any precompiler warnings
-o	Specifies that the embedded SQL statements are copied to the source code generated by the precompiler as comments
-l	Specifies that the precompiler writes the embedded SQL statements and their error messages to the precompiler log [Page 64]
-c	Prevents the compiler from being called automatically after the precompiler has run successfully However, the source file for the compiler is

	retained.
-T	Sets the default TRACE SHORT for the trace option [Page 66]
-X	Sets the default TRACE LONG for the trace option [Page 66]
-F <trace_file_name>	Sets the default for the name of the trace file (see Trace Options [Page 66])
-n <server_name>	Overrides the database server specified in the SET SERVERDB statement [Page 33] or CONNECT statement [Page 34] for the first database session
-d <database_name>	Overrides the database instance specified in the SET SERVERDB statement or CONNECT statement for the first database session
-u <user_name>,<password>	Overrides the user name and password specified in the CONNECT statement of the first database session
-U <user_key>	Overrides the KEY specified in the CONNECT statement for the first database session
-S <sql_mode> **<sql_mode> = internal|ansi|oracle**	Defines the SQL mode [See SAP DB Library] for your application program If you do not specify an SQL mode, the precompiler uses the `INTERNAL` mode automatically.
-D <datetimeformat> **<datetimeformat> = eur|iso|jis|usa|internal**	Defines the date and time format [See SAP DB Library] used in the application program If you do not specify a format, the precompiler uses the `INTERNAL` format automatically.
-V	Displays the precompiler version
-<sdkver>	Specifies the version of the precompiler software For example, enter **-7402**, if you have installed Version 7.4.02.
-h	Displays all call options for the precompiler and precompiler runtime environment

Precompiler Log

When the precompiler runs [Page 61], the C/C++ precompiler writes a log file `<filename>.pcl`. Depending on the specified precompiler options [Page 62], this file can contain the source code, warnings, and error messages from the precompiler.

If errors occur before the precompiler can generate the log file, then they are logged in the file `sqlerror.pcl`.

Example: Compiling a SAP DB Application Program

You have written a source code file `test.cpc`.

Use the command **`cpc -d DB -u Jones,secret test`**

to generate the file `test.o`. This uses the database instance **`DB`** with the user **`Jones`** and the password **`secret`**.

Use the command **`cpclnk test`**

to link the file with the runtime library and generate an executable program.

If you want a trace file to be written when the `test` program is executed, set the environment variable `SQLOPT` to **`-X`**. The corresponding command in a Bash shell is as follows:

```
export SQLOPT=-X
```

The trace file `test.pct` is written when the `test` program is executed. This file is located in the same directory as the `test` program.

To test the newly installed precompiler, you can use the SAP DB test database at *http://www.sapdb.org*.

After you install the package (5 KB), you have a database called `TST`, and the user `TEST` with the password `TEST`.

Example of the call for compiling the file `test.cpc`:

```
cpc -d TST -u TEST,TEST test
```

Functions of the Precompiler Runtime Environment

The precompiler runtime environment performs the following functions at runtime of the application program:

- Connecting to the databases and opening database session as specified in the connection options [Page 66]
- Assigning parameter values
- Converting data types [Page 25]
- Executing precompiler statements
- Using indicator variables [Page 18] to denote NULL values (undefined values)
- Writing database system messages [Page 55] in the structure `sqlca`
- Writing the trace file as specified in the trace options [Page 66]

See also the example: Executing a SAP DB Application Program [Page 69]

You can use the IRCONF [Page 70] tool to administer the installed versions of the runtime library of the interface programs, including the precompiler.

Connection Options at Runtime

The table shows you all options that you can set for the precompiler runtime environment [Page 65] in the environment variable `SQLOPT` when you connect to the database instance.

The options **`-u`**, **`-U`**, **`-t`** and **`-I`** are then valid for all database connections of the application program for which you have specified the key `<key> DEFAULT` or `SQLOPT` in the CONNECT statement [Page 34].

The options **`-n`** and **`-d`** apply to all database connections and override the settings specified in the CONNECT or SET SERVERDB statement [Page 33].

For information on setting the environment variable, see your operating system documentation.

`-n <server_name>`	Name of the database server
`-d <database_name>`	Name of the database instance
`-u <user_name,password>`	User name and password
`-U <user_key>`	XUSER user key [See SAP DB Library]
`-t <timeout>`	Timeout value [See SAP DB Library] `in seconds` If the default `-1` is specified, then the precompiler runtime environment uses the timeout value specified in the database parameter SESSION_TIMEOUT [See SAP DB Library].
`-I <isolation_level>`	Isolation level [See SAP DB Library] If no isolation level is specified, the precompiler runtime environment uses level 10 automatically.

Trace Options

Use

You can use the trace options to specify that the precompiler runtime environment [Page 65] logs the execution of the entire application program in the trace file `<filename>.pct` at runtime.

If you want to log individual embedded SQL statements [Page 9] only, then use the TRACE statements [Page 55].

You can use the IRTRACE [Page 68] tool to activate and deactivate the trace at runtime.

Procedure

- Specify your chosen default for the trace option in the precompiler options [Page 62] when you run the precompiler [Page 61].
- Set your chosen trace option for runtime in the environment variable `SQLOPT`.

The following table shows the possible trace options that you can specify when running the precompiler **and** calling the application program.

-T	Logs with the option TRACE SHORT Every executed SQL statements is logged, including the messages `sqlcode` und `sqlerrd[2]`.
-X	Logs with the option TRACE LONG Every executed SQL statements is logged, including the messages `sqlcode` und `sqlerrd[2]` and all parameter values.
-F <trace_file_name>	Allows you to choose a name other than `<filename>.pct` for the trace file If you do not specify any other trace options, then the option TRACE SHORT is activated at the same time. If you specify **pid** or **PID** for **<trace_file_name>**, then the name of the trace file becomes `pid<process_id>.pct` automatically. In this way, you can generate multiple trace files in parallel. If you do not specify any other trace options, then logging is deactivated at the same time.

The following table shows the possible trace options that you can specify **only** when you call the application program:

-N	Suppresses the logging of the date and time for the start and end of each executed SQL statement
-L <execution_time>	Specifies that any SQL statements that take longer to execute than **<execution_time>** are logged Specify **<execution_time>** in seconds. The log option TRACE LONG is used automatically. Dates and times cannot be suppressed with **-N**.
-Y <statement_count>	Specifies that logging alternates between the two trace files `<filename>.pct` and `<filename>.prot` in the run directory of the application program. Use **<statement_count>** to specify how many SQL statements are logged in each file. If the number of SQL statements exceeds **<statement_count>** then the trace files are overwritten cyclically.

	If you specify this option, then the option TRACE LONG is used automatically. Dates and times cannot be suppressed with **-N**.

See also Example of a Trace File [Page 69]

Using IRTRACE

IRTRACE is a tool for changing the trace settings [Page 68] and displaying the current trace setting [Page 68] of the precompiler runtime environment [Page 65] for a SAP DB application program translated with the C/C++ precompiler.

To do this, you enter commands at the operating system level. IRTRACE and the application program communicate through a shared memory segment.

If you use IRTRACE to change trace settings, the system makes an entry in the shared memory segment. The application program checks the entries in the shared memory at regular intervals, and changes its trace setting accordingly. The assignment of an application program to the correct entry in the shared memory is made using the process ID of the application program. The entry remains in the shared memory for as long as the corresponding process is active, and can be queried.

When a shared memory segment is being used, the system creates the synchronization file `irtrace.shm`, which the processes of the application program and IRTRACE use to access the shared memory. The system creates this release-independent file in the directory `/opt/sapdb/indep_prog/wrk`. In the installation, the system registers the path `/opt/sapdb/indep_prog` and creates the subdirectory `wrk`. The system also gives read and write rights to the IRTRACE tool and the application program.

Displaying the Current Trace Setting

Use the IRTRACE [Page 68] tool to display the current trace setting.

To do this, enter the following commands:

- Display the current trace setting for the application program:

  irtrace -p <process_id>

- Display overview of trace settings for all application programs where changes have been made with IRTRACE:

  irtrace -p all

Changing the Trace Setting

You can use the IRTRACE [Page 68] tool to change the current trace setting.

- Activating/deactivating/switching the trace for a specified process:
  irtrace -p <process_id> -t <trace_setting>
- Activating or deactivating the trace for **all** application programs on the local server that have been translated by the precompiler.
  irtrace -p all -t <trace_setting>

This table shows the possible values for **<trace_setting>**:

<long>	Activates logging with option TRACE LONG

<short>	Activates logging with option TRACE SHORT
<off>	Deactivates trace

See also: Trace Options [Page 66].

Example of a Trace File

```
PREPARE        :
select name, price into ?, ? from hotel where hno=?
SQL STATEMENT  :  FROM MODULE : docu07            AT LINE : 80
Statement Name : STMT2
PARSEID: OUTPUT:  00000347 00000901 3D001400
START  :  DATE :  2001-11-02    TIME :  0015:40:00
END    :  DATE :  2001-11-02    TIME :  0015:40:00
SQL STATEMENT  :  FROM MODULE : docu07            AT LINE : 80
Statement Name : STMT2
PARSEID: INPUT :  00000347 00000901 3D001400
INPUT :    3: hno                              :                  10
OUTPUT:    1: name                             : Excelsior
OUTPUT:    2: price                            :               135.00
SQLERRD(INDEX_3) : 1
START  :  DATE :  2001-11-02    TIME :  0015:40:00
END    :  DATE :  2001-11-02    TIME :  0015:40:00
```

Example: Executing an Application Program

You want to execute an application program, `test`, that you have translated with the C/C++ Precompiler.

If you want a trace file to be written when the `test` program is executed, set the environment variable `SQLOPT` to **`-X`**. The corresponding command in a Bash shell is as follows:

```
export SQLOPT=-X
```

The trace file `test.pct` is written when the `test` program is executed. This file is located in the same directory as the `test` program.

To test the newly installed precompiler, you can use the SAP DB test database at http://www.sapdb.org.

After you install the package (5 KB), you have a database instance called `TST`, and the user `TEST` with the password `TEST`.

In a Bash shell you can, for example, execute the following command:

```
export SQLOPT="-d TST -u TEST,TEST"
```

The result of this is that your application program uses, at runtime, the database instance `TST` and the user `TEST` with the password `TEST`.

Return Codes

The precompiler [Page 60] returns the following return codes:

0	Precompiler and compiler ran without errors
1	Precompiler ran with errors
2	Precompiler ran without errors; compiler ran with errors
3-120	Number of precompiler errors
126	Error in precompiler data management
127	Precompiler called; compiler not yet called

The SAP DB application program returns the following return codes:

1	No error
2	Error when reading the XUSER file [See SAP DB Library]
3	Database instance is not in the operational state [See SAP DB Library] ONLINE
4	Too many database sessions
5	Unknown combination of user name and password
6	Application program ended with WHENEVER STOP statement [Page 51]
7	SQLOPT contains incorrect options
8	Error when opening or writing to the trace file [Page 66]
9	Internal data structures inconsistent (system error)

Using the IRCONF Tool

For connections and communication between the database server and the client, the runtime library of the interface programs (such as the precompiler and ODBC) must be installed on the client.

You can update this part of the client software. So that you can continue to use and administer any older applications, the various versions of the runtime library of the interface programs are installed in different directories. The registrations in the system perform these installations automatically.

IRCONF is a tool that you can use to register and display the installed versions of the runtime library of the interface programs at a later time. You do this by entering commands at the operating system level, with your choice of options [Page 71].

Options for IRCONF

The following options are available when you use the IRCONF [Page 70] tool:

Specifying the installation path [Page 71]	**-p <path>**
Displaying registered versions of the runtime library [Page 72]	**-s**
Deleting the registration of the runtime library [Page 72]	**-r**
Overriding the driver name [Page 72]	**-d <CPC_driver>**
Registering the version of the runtime library [Page 73]	**-i**
Overriding the version check [Page 73]	**-v <version>**
Specifying an additional key [Page 74]	**-k <key>**
Displaying options and help [Page 74]	**-h**

Specifying the Installation Path: -p

Use

Use the option **-p** to define the installation path. You only need to specify the root directory here.

After the installation, the runtime library of the interface programs is in a subdirectory of the root directory:

- Windows NT: `.\pgm`
- UNIX (32 bit): `./lib`

The installation saves the runtime library of the interface programs for 64 bit UNIX operating systems in the subdirectory `./lib/lib64`.

You can link the option **-p** with the options **-i** (Registering the Version of the Runtime Library [Page 73]) and/or **-v** (Overriding the Version [Page 73]).

Displaying Registered Versions of the Runtime Library: -s

Use

Use the option `-s` to display the registered versions of the runtime library of the interface programs on the client.

Syntax

```
irconf -s
```

Result

You see the paths and versions of the installed runtime library of the interface programs.

```
/opt/sapdb/interfaces/precompiler/runtime/7300 -> 7.3.00.00
/opt/sapdb/interfaces/precompiler/runtime/7401 -> 7.4.01.00
```

Deleting the Registration of the Runtime Library: -r

Use

Use the option **`-r`** to delete the registration of a runtime library of the interface programs on the client. Specify the path in which the client installation is located.

Syntax

```
irconf -r -p <path>
```

```
irconf -r -p /opt/sapdb/interfaces/precompiler/runtime/7.2
```

Result

You see a message about the deletion of the registration, including the path and the version of the runtime library of the interface programs.

```
/opt/sapdb/interfaces/precompiler/runtime/7401 -> 7.4.01.00
registration removed
```

Overriding the Driver Name: -d

Use

The driver name is used for finding out versions information about the runtime libraries of the interface programs. The default is **`libpcr`**.

The option **`-d <CPC_driver>`** is needed for support purposes only.

Syntax

```
irconf -i -d <CPC_driver>
```

```
irconf -i -d <libpcr.old>
```

Registering the Version of the Runtime Library: -i

Use

Use the option **-i** to register a version of the runtime library of the interface programs on the client. Use the option **-p <path>** to specify the path under which this version of the runtime library has been installed.

During registration, the version of the runtime library of the interface programs (Windows NT: `libpcr.dll`, UNIX: `libpcr.so` or `libpcr.sl`) is checked, and the version found by the system is registered. In special cases, you may need to avoid this version check (Overriding the Version: -v [Page 73]).

Syntax

```
irconf -i -p <path>
```

```
irconf -i -p /opt/sapdb/interfaces/precompiler
/runtime/7401
```

Result

You see a message about the successful registration, including the path and the version of the installed runtime library of the interface programs.

```
/opt/sapdb/interfaces/precompiler /runtime/7401 ->
7.4.01.00 registered
```

Overriding the Version Check: -v

Use

The IRCONF [Page 70] tool attempts to determine the version of the installed runtime library of the interface programs. In special cases, you can use the option **-v** to override this feature. You can specify this option when you register the version of a runtime library, and then save the version under a version string of your choice.

You want to register a 64 bit version of the runtime library of the interface programs with a 32 bit version of IRCONF. The 32 bit version of IRCONF cannot determine the version of the 64 bit software, and you need to override the feature that determines the version.

In a special support scenario, you need to register the version of the runtime library under another name, and you need to override the feature that determines the version.

Syntax

```
irconf -i -v <version> -p <path>
```

Result

You see which version is installed under the specified path.

```
/usr/sapdb-if/runtime/7401 -> 7.4.01.00
```

Specifying an Additional Key: -k

Use

The registration procedure does not permit two identical versions of the runtime library of the interface programs to be determined uniquely on one server (in different directories). This can be the case if the test system and the production system are both installed on the same server, but the binaries come from different directories.

In this case, use the option **-k** to name and retrieve the versions. Set the environment variable `SAPDBINSTKEY` to the value specified in **<key>**, so that the applications can use the version registered in this key.

Syntax

```
irconf -i -p <path> -k <key>
```

```
irconf -i -p /opt/sapdb/interfaces/precompiler/runtime/7401
-k db_instance_name
```

Result

You see under which path the version specified in **<key>** was registered.

```
/opt/sapdb/interfaces/precompiler/runtime/7401 -> 7.4.01.00
db_instance_name
```

Displaying Options and Help: -h

Use

You can use this option to display a summary of all options possible with IRCONF [Page 70].

Syntax

```
irconf -h
```

Result

You see the following output:

```
Usage : irconf -i[-v | -d | -p | -k] | -r -p | -s | -h

Options:

-i Registers a version

-r Unregisters a version

-s Displays all installed versions

-v <version> Overrides version number identified by the system

-d <CPC_driver> Default is 'libpcr'

-p <path> Installation path

-k <key> Installation name

-h Displays help
```

IRCONF Error Messages

IRCONF can write the following error messages:

- `Invalid option`
  An invalid option [Page 71] was specified when IRCONF was called.
- `Invalid combination of options`
- `Missing CPC Driver name`
  Option **-d** specified without arguments.
- `Missing path name`
  Option **-p** specified without arguments.
- `Missing installation name`
  Option **-k** specified without arguments.
- `Missing version number`
  Option **-v** specified without arguments.
- `Cannot get version info of CPC Driver`
  The version of the runtime library of the interface programs on the client cannot be found.
- `The specified driver does not exist`
  The specified driver does not exist or cannot be loaded.
- `Cannot find any installed version of CPC Driver`
  No installations of the runtime library of the interface programs are registered.
- `The specified installation does not exist`

Syntax List

This syntax list shows you the syntax of all precompiler statements. The syntax notation [See SAP DB Library] used is BNF.

For the syntax of the database statements in the SQL mode INTERNAL, see the Reference Manual: SAP DB [See SAP DB Library].

as_clause

```
<as_clause> ::=
  AS VAR [<variable_declarator [Page 91]>]
| AS TYPE [<type_declarator [Page 90]>]
| AS STRUCT [<structure_tag [Page 89]>]
```

array_statement

```
<array_statement [Page 48]> ::=
  [<for_clause [Page 84]>] <dml_statement [Page 82]>
```

cancel_session

```
<cancel_session> ::=
  <session_number [Page 88]>
| <session_name [Page 88]>
| CURRENT
```

cancel_statement

```
<cancel_statement [Page 54]> ::=
  EXEC SQL CANCEL [<cancel_session [Page 78]>];
```

c_function

```
<c_function> ::=
  <simple_identifier [See SAP DB Library]>
```

char_host_var

<char_host_var [Page 15]> ::=
  <hostvarprefix> <identifier [See SAP DB Library]>

(for variables of type chardecspec only)

close_statement

<close_statement> ::=
  CLOSE [<cursor_name [Page 80]>]

command

<command> ::=
  <string_constant [Page 89]>
| <char_host_var [Page 79]>
| <unichar_host_var [Page 90]>

connect_option

<connect_option> ::=
  [ISOLATION LEVEL <unsigned_integer [See SAP DB Library]>] [TIMEOUT
<unsigned_integer [See SAP DB Library]>]

connect_statement

<connect_statement [Page 34]> ::=
  <connect_statement_internal [Page 79]>
| <connect_statement_oracle [Page 80]>

connect_statement_internal

<connect_statement_internal [Page 34]> ::=
  EXEC SQL [<session_number [Page 88]>]
  CONNECT [<user_name [See SAP DB Library]> IDENTIFIED BY <password [See SAP
DB Library]> | <uidpwd [Page 90]>]
  [<connect_option [Page 79]>...] [KEY <key [Page 86]>];

connect_statement_oracle

<connect_statement_oracle [Page 35]> ::=
  EXEC SQL
  CONNECT [<user_name [See SAP DB Library]> IDENTIFIED BY <password [See SAP DB Library]> | <uidpwd [Page 90]>]
  AT <session_name [Page 88]> USING <database_server [Page 80]>-<database_name [Page 80]>
  [<connect_option [Page 79]>...] [KEY <key [Page 86]>];

cursor_name

<cursor_name> ::=
  <result_table_name [See SAP DB Library]>
| <char_host_var [Page 79]>
| <unichar_host_var [Page 90]>

database_name

<database_name> ::=
  <string_constant [Page 89]>
| <char_host_var [Page 79]>

database_server

<database_server> ::=
  <string_constant [Page 89]>
| <char_host_var [Page 79]>

dbproc_clause

<dbproc_clause> ::=
  DBPROC <dbproc_name [Page 80]>

dbproc_name

<dbproc_name> ::=
  [<owner [See SAP DB Library]>.]<procedure_name [See SAP DB Library]>
[(<parameter_list [Page 87]>)]

ddl_statement

```
<ddl_statement> ::=
  <create_table_statement [See SAP DB Library]>
| <drop_table_statement [See SAP DB Library]>
| <alter_table_statement [See SAP DB Library]>
| <rename_table_statement [See SAP DB Library]>
| <rename_column_statement [See SAP DB Library]>
| <exists_table_statement [See SAP DB Library]>
| <create_domain_statement [See SAP DB Library]>
| <drop_domain_statement [See SAP DB Library]>
| <create_sequence_statement [See SAP DB Library]>
| <drop_sequence_statement [See SAP DB Library]>
| <create_synonym_statement [See SAP DB Library]>
| <drop_synonym_statement [See SAP DB Library]>
| <rename_synonym_statement [See SAP DB Library]>
| <create_view_statement [See SAP DB Library]>
| <drop_view_statement [See SAP DB Library]>
| <rename_view_statement [See SAP DB Library]>
| <create_index_statement [See SAP DB Library]>
| <drop_index_statement [See SAP DB Library]>
| <alter_index_statement [See SAP DB Library]>
| <rename_index_statement [See SAP DB Library]>
| <comment_on_statement [See SAP DB Library]>
| <create_trigger_statement [See SAP DB Library]>
| <drop_trigger_statement [See SAP DB Library]>
| <create_dbproc_statement [See SAP DB Library]>
| <drop_dbproc_statement [See SAP DB Library]>
```

declare_clause

```
<declare_clause> ::=
  <table_clause [Page 89]>
| <dbproc_clause [Page 80]>
```

declare_cursor_statement

```
<declare_cursor_statement> ::=
  EXEC SQL [<session_spec [Page 88]>]
  DECLARE <cursor_name [Page 80]> CURSOR FOR <statement_source [Page 89]>;
```

declare_statement

```
<declare_statement [Page 32]> ::=
  EXEC SQL BEGIN DECLARE SECTION;
| EXEC SQL END DECLARE SECTION;
```

describe_statement

<describe_statement [Page 45]> ::=
  EXEC SQL [<session_spec [Page 88]>] DESCRIBE <statement_name [Page 89]>
  [INTO <descriptor_name [Page 82]> [<using_clause [Page 90]>]];

descriptor_name

<descriptor_name> ::=
  <sqlda_variable [Page 89]>

dml_statement

<dml_statement> ::=
  <select_statement [See SAP DB Library]>
| <select_into_statement> (nur SQL-Modus ORACLE)
| <insert_statement [See SAP DB Library]>
| <update_statement [See SAP DB Library]>
| <delete_statement [See SAP DB Library]>
| <close_statement [Page 79]>
| <putval_statement [Page 88]>
| <getval_statement [Page 84]>
| <call_statement [See SAP DB Library]>
| <explain_statement [See SAP DB Library]>

dyna_parameter

<dyna_parameter> ::=
  <parameter_marker [Page 87]> [<indicator_marker [Page 85]>]

dyna_parameter_list

<dyna_parameter_list> ::=
  <dyna_parameter [Page 82]>,...>

embedded_sql_statement

<embedded_sql_statement [Page 30]> ::=
  EXEC SQL [<session_spec [Page 88]>]
| <ddl_statement [Page 81]>;
| <dml_statement [Page 82]>;
| <array_statement [Page 78]>;

exec_command_statement

<exec_command_statement [Page 54]> ::=
  EXEC COMMAND <os_command [Page 86]>;

execute_immediate_statement

<execute_immediate_statement [Page 37]> ::=
  EXEC SQL [<session_spec [Page 88]>] EXECUTE IMMEDIATE <statement_source [Page 89]>;

execute_statement

<execute_statement [Page 46]> ::=
  EXEC SQL [<session_spec [Page 88]>] [<for_clause [Page 84]>]
  EXECUTE <statement_name [Page 89]> [<using_clause [Page 90]>];

fetch_spec

<fetch_spec> ::=
  FIRST
| LAST
| NEXT
| PREV
| SAME
| POS (<unsigned_integer [See SAP DB Library]>)
| POS (<int_host_var [Page 86]>)

fetch_statement

<fetch_statement> ::=
  EXEC SQL [<session_spec [Page 88]>] [<for_clause [Page 84]>]
  FETCH [<fetch_spec [Page 83]>] [<cursor_name [Page 80]>] <using_clause [Page 90]>;

file_host_var

<file_host_var> ::=
  <hostvarprefix [Page 84]><identifier [See SAP DB Library]>

(nur für Variablen vom Typ vfldecspec erlaubt)

file_name

```
<file_name> ::=
  <string_constant [Page 89]>
```

float_host_var

```
<float_host_var> ::=
 <hostvarprefix [Page 84]><identifier [See SAP DB Library]>
(for variables of type flpdecspec only)
```

for_clause

```
<for_clause> ::=
  FOR <loop_parameter [Page 86]>
```

getval_statement

```
<getval_statement [Page 50]> ::=
  GETVAL INTO (<parameter_list [Page 87]>)
```

(For LONG columns only)

host_variable

```
<host_variable [Page 15]> ::=
  <char_host_var [Page 79]>
| <unichar_host_var [Page 90]>
| <int_host_var [Page 86]>
| <float_host_var [Page 84]>
```

hostvarprefix

```
<hostvarprefix> ::=
  :
```

include_declare_statement

```
<include_declare_statement [Page 32]> ::=
  EXEC SQL INCLUDE <file_name [Page 84]> <declare_clause [Page 81]>
  [<as_clause [Page 78]>] [<ind_clause [Page 85]>];
```

include_file_statement

<include_file_statement [Page 49]> ::=
  EXEC SQL INCLUDE <file_name [Page 84]>;

include_sqlca_statement

<include_sqlca_statement> ::=
  EXEC SQL INCLUDE SQLCA;

include_statement

<include_statement> ::=
  <include_sqlca_statement [Page 85]>
| <include_file_statement [Page 85]>
| <include_declare_statement [Page 32]>

ind_clause

<ind_clause> ::=
  IND [<ind_variable_declarator [Page 85]>] [<structure_tag [Page 89]>]

indicator_marker

<indicator_marker> ::=
  ?

indicator_variable

<indicator_variable [Page 18]> ::=
  <int_host_var [Page 86]>

ind_variable_declarator

<ind_variable_declarator> ::=
  <simple_identifier [See SAP DB Library]>

int_host_var

```
<int_host_var> ::=
  <hostvarprefix [Page 84]><identifier [See SAP DB Library]>
(for variables of type intdecspec only)
```

key

```
<key> :: =
  <simple_identifier [See SAP DB Library]>
```

label

```
<label> :: =
  <identifier [See SAP DB Library]>
```

loop_parameter

```
<loop_parameter> ::=
  <unsigned_integer [See SAP DB Library]>
| <int_host_var [Page 86]>
```

open_cursor_statement

```
<open_cursor_statement [Page 46]> ::=
  EXEC SQL [<session_spec [Page 88]>] [<for_clause [Page 84]>] OPEN
<cursor_name [Page 80]>
  [  USING <parameter_list [Page 87]>
   | USING DESCRIPTOR [<descriptor_name [Page 82]>] [KEEP]
   | INTO <parameter_list [Page 87]>]
   | INTO DESCRIPTOR [<descriptor_name [Page 82]>] [KEEP]];
```

os_command

```
<os_command> ::=
  <os_command_async [Page 87]>
| <os_command_sync [Page 87]>
```

os_command_async

```
<os_command_async> ::=
  ASYNC <os_command [Page 86]>
```

os_command_sync

```
<os_command_sync> ::=
  SYNC <os_command [Page 86]> RESULT <result_param [Page 88]>
```

parameter

```
<parameter> ::=
  <host_variable [Page 84]> [<indicator_variable [Page 85]>]
```

parameter_list

```
<parameter_list> ::=
  <parameter [Page 87], ...>
```

parameter_marker

```
<parameter_marker> ::=
  ?
```

precom_version

```
<precom_version> ::=
  <char_host_var [Page 79]>
```

prepare_statement

```
<prepare_statement [Page 45]> ::=
  EXEC SQL [<session_spec [Page 88]>] PREPARE <statement_name [Page 89]>
  [INTO <descriptor_name [Page 82]> [USING <using_clause [Page 90]>]]
  FROM <statement_source [Page 89]>;
```

putval_statement

<putval_statement [Page 49]> ::=
  PUTVAL INTO <table_name [See SAP DB Library]> [(<column_name [See SAP DB Library]>,...)]
  VALUES (<parameter_list [Page 87]>)

(For LONG columns only)

result_param

<result_param> ::=
  <int_host_var [Page 86]>

rte_version

<rte_version> ::=
  <char_host_var [Page 79]>

session_name

<session_name> ::=
  <string_constant [Page 89]>
| <char_host_var [Page 79]>

session_number

<session_number> ::=
  1|2|3|4|5|6|7|8|9

session_spec

<session_spec> ::=
  <session_number [Page 88]> (SQL-Modus Internal)
| AT <session_name [Page 88]> (SQL-Modus ORACLE)

set_serverdb_statement

<set_serverdb_statement [Page 33]> ::=
  EXEC SQL [<session number [Page 88]>]
  SET SERVERDB <database_name [Page 80]> [ON <database_server [Page 80]>];

sqlda_variable

```
<sqlda_variable> ::=
  <simple_identifier [See SAP DB Library]>
```

statement_name

```
<statement_name> ::=
  <string_constant [Page 89]>
| <char_host_var [Page 79]>
| <unichar_host_var [Page 90]>
```

statement_source

```
<statement_source> ::=
  'dml_statement [Page 82]'
| 'ddl_statement [Page 81]'
| <statement_name [Page 89]>
```

string_constant

```
<string_constant> ::=
  "<simple_identifier [See SAP DB Library]>"
| '<simple_identifier [See SAP DB Library]>'
```

structure_tag

```
<structure_tag> ::=
  <simple_identifier [See SAP DB Library]>
```

table_clause

```
<table_clause> ::=
  TABLE <table_name [See SAP DB Library]>
```

trace_line

```
<trace_line> ::=
  <string_constant [Page 89]>
| <char_host_var [Page 79]>
```

trace_state

```
<trace_state> ::=
  TRACE ON
| TRACE LONG
| TRACE OFF
| TRACE LINE <trace_line [Page 89]>
```

trace_statement

```
<trace_statement> ::=
  EXEC SQL SET TRACE <trace_state [Page 90]>;
```

type_declarator

```
<type_declarator> ::=
  <simple_identifier [See SAP DB Library]>
```

uidpwd

```
<uidpwd> ::=
  <user_name [See SAP DB Library]>/<password [See SAP DB Library]>
```

unichar_host_var

```
<unichar_host_var> ::=
  <hostvarprefix [Page 84]><identifier [See SAP DB Library]>
(for variables of type unichardecspec only)
```

using_clause

```
<using_clause [Page 47]> ::=
  USING <using_expr [Page 90]>
| INTO <using_expr [Page 90]>
```

using_expr

```
<using_expr> ::=
  <parameter_list [Page 87]>
| DESCRIPTOR [<descriptor_name [Page 82]>]
```

variable_declarator

```
<variable_declarator> ::=
  <simple_identifier [See SAP DB Library]>
```

variable_name

```
<variable_name> ::=
  <simple_identifier [See SAP DB Library]>
```

version_statement

```
<version_statement> ::=
  EXEC SQL VERSION <rte_version [Page 88]>, <precom_version [Page 87]>;
```

whenever_action

```
<whenever_action> ::=
  CALL <c_function [Page 78]>
| CONTINUE
| GOTO <label [Page 86]>
| STOP
```

whenever_condition

```
<whenever_condition> ::=
  SQLWARNING
| SQLERROR
| SQLEXCEPTION
| NOT FOUND
| SQLBEGIN
| SQLEND
```

whenever_statement

```
<whenever_statement [Page 51]> ::=
  EXEC SQL WHENEVER <whenever_condition [Page 91]> <whenever_action [Page 91]>;
```