
COBOL-IT OpenESQL

Guide

4.11.1

Table of contents

About COBOL-IT OpenESQL	3
Documentation	3
Overview	5
Pre-requisite knowledge	5
Getting Started Guide	6
Getting Started	6
Working with the Getting Started application – Basic	7
Tutorials and Best Practice Recommendations	11
Runtime Errors and Diagnostics	15
Migration Guide	20
Migrating to CitOESQL	20
User Guide	28
Modes of Operation	28
Command line syntax	29
Source Code Formats	30
Directive Syntax	31
Control statements in source	32
Programming	33
Reference Manual	46
Developing SQL Applications	46
SQL Statements	80
CitOESQL Directives	0
Legal Notice	0
Third-Party Notices	0

1. About COBOL-IT OpenESQL

Welcome to COBOL-IT OpenESQL (CitOESQL).

1.1 Documentation

Below are the guides available for CitOESQL:



Getting Started

Use the Getting Started application to set up an ODBC data source...



Migration Guide

Learn how to migrate from CitSQL to CitOESQL



User Guide

Walk through the product features.



Reference Manual

This guide describes the programming features available for SQL applications.

1.2 Overview

COBOL-IT OpenESQL is an Embedded SQL (ESQL) preprocessor for COBOL-IT. It reads COBOL source code and writes amended source code where EXEC SQL statements are replaced with calls to a runtime library that accesses ODBC data sources.

COBOL-IT OpenESQL can be used as a stand-alone preprocessor that is executed separately and before the COBOL-IT compiler, or in conjunction with COBOL-IT's `-preprocess=` option, in which case it is invoked by the COBOL compiler.

When used stand-alone, the preprocessor provides conditional compilation and copybook expansion. These tasks may be performed by the COBOL compiler when used with the `-preprocess` option.

The debugger will show the code generated by CitOESQL rather than the original EXEC SQL statements. When used with the compiler's `-preprocess` option, debugging of the original source code is available.

1.3 Pre-requisite knowledge

Some basic knowledge of Embedded SQL is assumed in this document.

It is also assumed that the reader has basic knowledge of the COBOL-IT compiler and debugger and has worked through the getting started process for them. The same applies to COBOL-IT Developer Studio, if this is used.

2. Getting Started Guide

2.1 Getting Started

To work with the Getting Started application you must first set up an ODBC data source for the database of your choice. You will need to know the ODBC Data Source Name (DSN) and a valid UserID and Password for the DSN that is able to create and drop tables.

When working with PostgreSQL you must set the following ODBC options in your environment:

```
UpdatableCursors=0
```

```
UseDeclareFetch=1
```

On Linux you must include the ODBC shared object location in LD_LIBRARY_PATH. For example, if the unixODBC driver manager is installed in its default location, this will be:

```
export LD_LIBRARY_PATH=/usr/local/lib:\$LD_LIBRARY_PATH
```

The Getting Started application, sample.cbl, can be found in the following locations:

```
Windows: %COBOLITDIR%\samples\sql
```

```
Linux: $COBOLITDIR/samples/sql
```

The application executes a series of SQL statements. Open sample.cbl in a text editor of your choice to view these statements. Comments in the source code provide information about alternative methods for opening the database connection.

To work with the application, open a command line window and execute the appropriate cobol-itsetup script for your environment (see the Installing COBOL-IT section of the COBOL-IT Compiler Suite, [Getting Started With Compiler Suite](#) guide for more details). Then change the directory to the SQL sample directory.

2.2 Working with the Getting Started application – Basic

First run the precompiler:

```
citoesql sample.cbl
```

The -g parameter is required if you want to compile for debugging. You can omit it from release builds.

The -conf=citoesql.conf parameter sets compiler options required by CitOESQL including running CitOESQL's preprocessor and linking its runtime library.

You can also run CitOESQL as a standalone preprocessor and then compile the generated output file.

Next compile the file generated by CitOESQL, which will have the same name but a .cbp extension.

Windows: `cobc -g -I%COBOLITDIR%\lib\ citoesqlr_dll.lib sample.cbp`

Linux: `cobc -g -L$COBOLITDIR/lib -lcitoesqlr sample.cbp`

Note

The -I parameter links CitOESQL's runtime library.

The application can now be executed using `cobcrun`. Enter the DSN, User-ID and Password when prompted. The console output should be as follows:

```
cobcrun sample
Create/insert/update/drop test

Enter data source (Eg odbcdemo) MyDSN
Enter username[('.'\|'/')password] (Eg admin/) myUserId.myPassword
Drop table
Error(anticipated) : cannot drop table
-00003701
Cannot drop the table 'mfesqltest', because it does not exist
Create table
Insert row
Commit
Update row
Verify data before rollback
Rollback
Verify data after rollback
Drop table
Disconnect
Create table after commit release
Cannot create table as expected
-00019702
Connection name not found.
Test completed without error
```

You can debug the application using cobcdb, as detailed in the Using the COBOL-IT Debugger section of the COBOL-IT Compiler Suite, [Getting Started With Compiler Suite](#) guide.

To build an executable program rather than the default dll or shared object add -x to the command line as shown in the example below:


```
cobc -x -g -conf=citoesql.conf sample.cbl
```

2.2.1 Working with the Getting Started application in Developer Studio

Prerequisites

Set up a new workspace by following the steps documented in the Developer Studio Getting Started manual. In summary:

1. Validate the COBOL-IT license using the menu sequence **Window > Preferences > COBOL** then click the **Browse** button to select the COBOL-IT license file.
 2. Using the menu sequence **Window > Preferences > General > Editors > Text Editor**, ensure **Show line numbers** has been checked.
 3. Using the menu sequence **Window > Preferences > General > Workspace**:
 - De-select **Build automatically**
 - Select **Refresh using native hooks or polling**
 - Select **Refresh on access**
 - Select **Save automatically before build**
 4. Using the menu sequence **Window > Preferences > Run/Debug > Perspectives**, select **Always** for the **Open the associated perspective when launching**.
-

2.2.2 Project creation and import of source code

Follow the steps listed to create a new project and import source code.

1. Select **File > New > COBOL Project...** to open the COBOL Project Wizard.
2. Enter a project name, for example samples, and click **Finish**.
3. Right click on the newly created project and select **New > Folder**, enter the name object and click **Finish**.
4. Right click on the project again and select **Import** followed by **General** and **FileSystem** then click **Next**.

5. Click **Browse** and navigate to the **COBOL-IT samples > sql directory**, then click **OK**. Select **sample.cbl** and click **Finish**.
 6. Right click on the project again and select **Properties** followed by **COBOL Properties** then check the **Enable source settings** option.
 7. Select **DevOps Tools > Debugging Tools**. Check **Enable source settings** and **Produce debugging metadata in compiled object**.
 8. Select **Dialects > Compiler Configuration Files**. Enter **citoesql.conf** for **Use <file> as configuration file**.
 9. Select **Link > Full Build** and set **Build type** to **Build executable program**.
 10. Select **Standard Options** and check **Save object file in source folder or in <directory>** and enter **object** in the textbox.
 11. Click **Apply** and **Close**.
-

Build and Debugging

To build the project use the menu sequence **Project > Clean**, ensure the project is selected for cleaning and building and then click **Clean**.

You now need to set up a debug configuration as follows:

1. Double click on **sample.cbl** to open the source file.
2. Scroll down to the **EXEC SQL CONNECT** statement, right click the shaded area to the left of the line number of the **EXEC SQL** line and choose **Toggle Breakpoint**. A blue circle should appear next to the line number. You can also use double click to set and unset breakpoints as an alternative to right click.
3. Right click on **sample.cbl** in the project file tree in the left-hand pane, select **Debug As followed by Debug Configurations**.
4. Click on **Cobol Program** followed by the left most icon in the toolbar above (New launch configuration) and enter 'sample' as the configuration name at the top of the dialog box.
5. Check that the **Project Name** is set to the correct project and that **Program** is set to **sample.cbl**.
6. Click **Debug**.
7. Respond to the prompts from the program with the ODBC Data Source Name, User-ID and Password.
8. The application will stop on the breakpoint. You can now use other debugger features to inspect COBOL variables, manage other breakpoints, use single step code execution, etc. Please consult COBOL-IT Developer Studio, [Getting Started: The Debugger Perspective](#) guide for more information.

2.3 Tutorials and Best Practice Recommendations

Note

The tutorial applications are described below for command line use. If you prefer, they can be used in Developer Studio with directives following the pattern described in Working with the Getting Started application in Developer Studio.

You can control the behavior of CitOESQL by setting precompiler directives. This can be done in a variety of ways, on the command line, in source code and in directive files. The order in which sources of directives are used, and hence the reverse order of precedence if a directive is set more than once is:

The command line

Automatic directive files

Source code

Directive files can be specified explicitly with the USE directive, however some directive files, if they exist, are used automatically by CitOESQL. These are recommended as the most convenient way of managing directive use. The automatic directive files, and the order in which they are applied are:

- %COBOLITDIR%\etc\citoesql.dir (Windows) or \$COBOLITDIR/etc/citoesql.dir (Linux and Unix).

This is useful for setting directives that will be applied to all programs.

- `citoesql.dir` in the directory containing the COBOL source code.

This is useful for setting directives that are common to groups of programs.

- A file with the same name and in the same directory as the program being pre-compiled but with a file extension of `'dir'`.

2.3.1 Performance Tuning

Introduction

Reducing the number of interactions required between components to perform a given task will often improve application performance. In a SQL application, the most significant of these is the number of calls made between the client application and the database management system, often referred to as 'round trips'. In some cases, the database client library or ODBC driver will perform such optimizations transparently on behalf of the client application. In other cases, a client application can use optimization techniques explicitly.

CitOESQL provides several ways to optimize the performance of SQL applications and some of these are demonstrated in the tune.cbl sample application.

Note

Benchmark applications such as tune.cbl should be used with great care in predicting the performance of other applications. There are many factors that can affect application performance that benchmarks may fail to consider. Hardware configurations and the relative speeds of CPU, memory, disk and network also play a significant role, as do operating system overheads and other activity on the machine. tune.cbl is intended only to provide an illustrative guide to the relative performance of different coding approaches and the potential usage of the tuning parameters available in CitOESQL.

A prerequisite for running this application is an ODBC data source, see the Getting Started section for more details.

On Windows open a COBOL-IT command line window, on Linux or Unix run the COBOL-IT command line setup script.

- Change directory to the CobolIt (or CobolIT64) samples\sql or samples/sql directory
- Build the application with the following command line:

```
cobc -g -x -conf=citoesql.conf tune.cbl timer.cbl
```

- Run the application with the following command line:

```
tune (Windows) or ./tune (Linux and Unix)
```

- Respond to the prompts for Data Source, User-ID and Password.

You will see elapsed times for;

- Inserting rows via a single row insert statement and an array insert statement.
- Selecting and fetching rows using read only and updatable cursors or SELECT INTO statements using single row or multi-row retrieval.

Transparent Cursor Prefetch

When using single row cursor FETCH statements, CitOESQL can prefetch rows. By default, it will use a prefetch of 8 rows for read only cursors and 4 rows for updatable cursors. If you edit tune.dir you will see the prefetch settings at their default values. You can experiment with them as follows:

- Try setting them to 1 to see how much default prefetching improves performance.

- Try commenting them out with a # at the start of the line to confirm their default values.
- Try using larger values. You may find that very large values perform less well than smaller values in some cases and that for a given application there is often a “sweet spot” that performs best.

Host Variable Arrays

CitOESQL supports the use of host variable arrays to insert and fetch multiple rows at a time. When using array host variables you can use a FOR :count clause if you do not want to use the whole array. Edit tune.cbl and familiarize yourself with the syntax. You can change the 78-level constants demoRows, insertArraySize, readOnlyarraySize and forUpdatearraySize to change the number of rows in the table and the size of the host variable arrays. You can experiment with different values to see how different array sizes impact performance and how the BEHAVIOR directive and its subdirectives perform relative to host variable arrays.

It is often best to use a smaller array size for updatable cursors than for read only cursors. Although a larger array size will generally improve performance, it can also increase contention and lock wait delays. When considering array sizes and prefetch sizes for updatable cursors, you should consider if your databases hold FOR UPDATE locks only when a row is available on the client, for example in Microsoft SQL Server, or if FOR UPDATE locks are held until the current transaction terminates, for example in Oracle and PostgreSQL.

SQL Statement Cache

CitOESQL maintains a cache of prepared statements. The default cache size is 20. You can use the STMTCACHE directive to change this. For a large batch, application values up to the low hundreds may be beneficial.

2.3.2 SQL Syntax Checking Options

CitOESQL syntax checking is designed to be lightweight and tolerant of server-specific SQL extensions. This means that it may not detect all SQL errors. You can enable more rigorous checking with the CHECK directive. This also requires the DB directive and, in most cases, the PASS directive. You can see how this works with the tune.cbl sample:

- Navigate to %COBOLITDIR%\Samples\sql or \$COBOLITDIR/Sample/sql and open tune.dir in an editor of your choice.
- Remove the # character from the start of the line that starts #db and replace the string `<your ODBC Data Source Name>` with the name of your ODBC data source.

- Unless you are using operating system authentication, remove the `#` character from the line that starts `#pass` and replace the string `<UserID>` with your database User-ID, and the string `<Password>` with your database password.
- Remove the `#` character from the line that starts `#check`.
- Save the file, and open `tune.cbl`.
- Search for the first `INSERT INTO` statement and change the keyword `into` to `intoo`.
- Compile `tune.cbl` and notice the syntax-based error message.
- Change the `intoo` back to `into`, the table name `oesqldemo` to `xyz` and re-compile. Note the error message is now for a reference to a non-existent table.

Sometimes you cannot avoid errors because a table does not exist in the database, for example if the table is a temporary table that the database automatically drops at disconnect time. CitOESQL can work around this by executing DDL statements at precompile time.

- Navigate to the start of `tune.cbl` and search for `create table`.
- Note that the `EXEC SQL` statement starts with the prefix `'[also check]'`. This tells the precompiler that this statement should be executed at both precompile and execute time.
- The prefix `'[only check]'` instructs the precompiler that the statement should only be executed at precompile time.
- Go back to the top of the program and search for `'drop table'`. Note the prefix `'[also check ignore error]'`. This instructs the precompiler to execute the statement at both precompile time and execute time and to ignore any precompile time errors.

There may be other cases where a precompile time server-detected error is unavoidable, in which case the prefix `[nocheck]` can be used.

- Return to the start of `tune.cbl` and search for `'set :dbms'`. Note that immediately after opening a database connection, `tune.cbl` uses a `set :<hostVariable = current database CitOESQL` statement to determine the type of database it is connected to.
- Search for `if dbms =` and then scroll down a few lines until you can see two `create table` statements. `tune.cbl` includes two `create table` statements to enable it to use the `TYPE VARCHAR` with most databases. `VARCHAR2` is used with Oracle and exploits the `create or replace` syntax to avoid separate `create` and `drop` statements.
- The `create table` statements have the prefix `[nocheck] [also check ignore error]`. This has the following effect:
 - `[nocheck]` means there will be no syntax check at precompile time.

- `[also check ignore error]` means the statement will be executed at compile time and potentially also at runtime, however in `tune.cbl` COBOL code ensures only one of the two statements will be executed at runtime. Any precompile time execution error will be ignored.
- Note that this is a somewhat contrived example for demonstration purposes.

When migrating an application to a new database, you can use the `IGNORESCHEMAERRORS` directive in conjunction with the `CHECK` directive. This limits server syntax checking to syntax alone and does not treat missing tables and columns as errors. This may be useful in obtaining a quick appraisal of SQL syntax differences that need to be remediated before the schema has been migrated.

2.4 Runtime Errors and Diagnostics

Diagnostic information for SQL errors can be obtained in several ways in CitOESQL.

- The most common method in most embedded SQL applications is via a SQLCA data structure. The SQLCA is supplied by including `exec sql include sqlca endexec`, where `SQLCODE` contains a numeric error code and `SQLSTATE` a 5-character string, both of which identify the error condition. `SQLERRMC` contains a brief error message and `SQLERRML` contains the length of the message.
- An application may simply declare `SQLCODE` and/or `SQLSTATE` without using an SQLCA if it only requires the type of error.
- `SQLERRMC` is limited to 70 bytes. A longer and more complete error can be obtained by declaring a `PIC X(n)` field named `MFSQLMESSAGETEXT` where 'n' can be of any size, but 256 characters is generally sufficient to obtain the complete error message.

Note

ODBC error messages start with one or more component names in square brackets followed by the message text. These identify where the error message was detected, for example, by the ODBC Driver Manager or the ODBC driver, or the database server. CitOESQL removes any text in square brackets from `SQLERRMC`, but not from `MFSQLMESSAGETEXT`.

- Finally, complete diagnostic information, including the possibility of a SQL error returning more than one error, can be obtained via a `GET DIAGNOSTICS` statement.

All these methods are demonstrated by the `errref.cbl` sample application.

- Build the application in the `sql samples` directory via the following

```
command: cobc -x -g -conf=citoesql.conf errref.cbl
```

- Execute the sample via:

```
errref (Windows) or ./errref (Linux and Unix)
```

Note

errref generates the following common warning and error conditions:

- No data found or returned.
- Too many rows returned for SELECT INTO.
- Character data truncation.
- Attempt to insert a row with a duplicate key.

Open the `errref.cbl` file and inspect the code to see how the output was generated using the methods described above.

You will need the output of the application for the next tutorial

2.4.1 Diagnostic mapping for database migration

When migrating an application from one database to another you will need to address differences in SQL syntax between the two systems.

It may also be required to address cases where application logic or operation procedures have dependencies on the error diagnostics returned by the database. CitOESQL can assist this process by allowing diagnostics returned by the new database for a given error condition to be mapped to the diagnostics returned by the old database system, thus avoiding the need to make changes to the source code. This is achieved by use of an error mapping file. Errors can be mapped for an application via the `ERRORMAP` precompiler directive, or for the current connection by executing a `SET ERRORMAP` statement. These methods are demonstrated by the `errmap.cbl` sample.

- Navigate to the `sql samples` directory.
- Open `login.cpy` in a text editor of your choice and edit the `svr` and `usrpass` fields to match your ODBC DSN, User ID and Password. If using Operating System authentication `usrpass` should be `SPACES`. Save the file.

- Open `errmap.dir` and note the directive `errmap=test`. This means that at runtime `errmap` will load mapping information from a file named `test.emap`.
- Open `test.emap`. This is set up to map diagnostics returned by PostgreSQL. You can use the output from `errref` to update this file for other databases.

Note

Error mapping can match diagnostic information to be mapped by any combination of `SQLCODE`, `SQLSTATE` and a substring within the error message text. The mapping process will change `SQLCODE` and `SQLSTATE` to new values; specify the original values to leave them unchanged. The error message text can be:

Left unchanged by omitting it from the mapping

Replaced by new text

suppressed (i.e set to spaces by specifying a single `~` character as the replacement text)

- Replacement error messages start with `[test]`. This will appear in message text returned in `CITSQLMESSAGETEXT` and by `GET DIAGNOSTICS`, but not in `SQLERRMC`. It is there as a simple visual check that an error has been mapped successfully.
- Open `conn.emap`. This is very similar to `test.emap`, but will be used with a `SET ERRORMAP` statement rather than the `ERRORMAP` directive. Replacement error messages start with `[conn]`.
- Set the environment variable `CIT_ERRORMAP_PATH` to the current directory. This can be `'.'`. This will cause `CitOESQL` to look for the mapping files in the current directory and is useful for testing new mapping files. If not set, the default location of `%COBOLITDIR%\etc` (Windows) or `\$COBOLITDIR/etc` (Linux) will be used.
- Compile and run `errormap`.

```
cobc -g -x -conf=citoesql.conf errmap.cbl
errmap
```

- Inspect the output and verify that errors have been mapped.
- Open `errmap.cbl` and inspect the code.

To deploy error mapping files for general use:

- Edit the files and remove any unwanted leading text in square brackets at the start of replacement error messages.
- Copy the files to the default location: `%COBOLITDIR%\etc` (Windows) or `$COBOLITDIR/etc` (Linux)
- Unset environment variable `CIT_ERRORMAP_PATH` (if set).

2.4.2 COBOL-IT CitOESQL files and locations

File name	Windows location relative to %COBOLITDIR%	Linux/Unix location relative to \$COBOLITDIR)	Use
SQLCA.cpy	copy	share/cobol-it/copy	SQLCA definition
SQLDA.cpy	copy	share/cobol-it/copy	SQLDA definition
runcitoesql.bat runcitoesql.sh	bin	bin	Used by cobc - preprocess to execute the citoesql precompiler
citoesql.dir	etc	etc	Global default citoesql directives
citoesql.conf	config	share/cobol-it/config	cobc configuration file for integrated precompilation
citoesqlx.conf	config	share/cobol-it/config	cobc configuration file when citoesql is used without integrated precompilation
*.emap	etc	etc	Default location for error mapping files

2.4.3 Performance and Diagnostic Aids

CitOESQL provides two execution tracing capabilities that may help you in diagnosing bugs and performance issues. Both are controlled by directives.

Setting ODBCTRACE=ALWAYS will enable ODBC tracing in the ODBC driver manager. This can also be done via the ODBC Administrator (Windows) or by editing the ODBC configuration files `odbc.ini` and `odbcinst.ini` (Linux), however when developing or debugging an application you may find the precompiler directive more convenient. Trace information is appended to the trace file if it already exists, so you must remember to delete or clear the trace file between runs. ODBCTRACE traces ODBC entry to and exit from ODBC API calls along with diagnostic error messages. ODBCTRACE writes all trace records to disk immediately and consequently this can have a significant impact on performance.

TRACELEVEL traces the calls an application makes to the `CitOESQL`'s runtime library. The resultant traces can be used to analyse performance issues. TRACELEVEL offers several levels of detail and records directive settings and tuning statistics, such as the number of rows read by a cursor. TRACELEVEL has significantly less performance impact than ODBCTRACE but if an application terminates abnormally not all trace events may be recorded.

3. Migration Guide

3.1 Migrating to CitOESQL

3.1.1 CitSQL and CitOESQL Comparison

Implementation Comparison	CitSQL	CitOESQL
Command Line	<code>>citsql [options @optionFile] (source file(s) @sourcesFile)</code>	<code>>citoesql [options] [@]filesNames(s)</code> Options may include USE

3.1.2 Command line options

CitSQL	CitOESQL
<code>:CloseOnCommit=[True/False]</code> For Oracle compatibility, cursors after a commit.	<code>-CLOSEONCOMMIT=YES NO</code> <code>-CLOSEONROLLBACK=YES NO</code>
<code>:CursorSynteticName=[True/False]</code> Causes the Cursor Name to be generated dynamically at runtime by the RCQ runtime.	Unique cursor names are always generated at runtime.
<code>:DBEncoding=<Codepage or UTF-8]</code> Specifies what encoding is used by the DB storage.	ODBC drivers provide conversion between the client codepage and the database encoding, it is not necessary to inform the precompiler of the database encoding.

CitSQL	CitOESQL
<p><code>:DeallocateCloseCursor=[True/False]</code></p> <p>Causes the CLOSE CURSOR statement to also deallocate the DECLARED cursor.</p>	<p>No equivalent. CitOESQL maintains a cache of prepared statements to optimize performance, cursors, and other prepared statements. These are deallocated when necessary using a least recently used algorithm.</p>
<p><code>:DefaultCCSID=<Valid codepage or UTF-8></code></p> <p>Specifies the default CCSID for string fields that have no explicit CCSID declaration.</p>	<p>No equivalent, however, ODBC drivers will automatically convert data between the client and database encodings.</p>
<p><code>:DebugMode= [TRUE/FALSE]</code></p> <p>Default [FALSE] When set to TRUE, causes log file created when LogMode=TRUE to contain more detail in some situations.</p>	<p>The TRACELEVEL option provides varying degrees of logging for diagnostic and optimization purposes.</p> <p>ODBC provides its own API logging capability that can be turned on and off without recompiling an application.</p>
<p><code>:ForceStringMode=[TRUE/FALSE]</code></p> <p>When set to True, CitSQL noncompound PIC X data fields are sent to the database as a C-String (where the String is terminated by the character X'00'). When set to FALSE, CitSQL sends PIC X data to the database as a byte-array.</p>	<p>- [NO] ALLOWNULLCHAR</p>
<p><code>:FreeFormatOutput=[TRUE/FALSE]</code></p> <p>Default [FALSE] When set to TRUE, causes output of the precompiler to be created in "free" source format.</p>	<p>- SOURCEFORMAT=(FIXED FREE VARIABLE)</p>

CitSQL	CitOESQL
<p><code>:ImmediateCursor=[True/False]</code></p> <p>(CitSQL for PostgreSQL Only) When set to True, causes a PREPARE EXEC to be executed before the OPEN when a CURSOR is declared with OPEN and FETCH statements</p> <p>Attention should be taken when applying the ImmediateCursor preprocessor parameter. Since the full results of the cursor are returned before the OPEN statement, this parameter should only be applied for cursors returning small numbers of lines.</p> <p>(CitSQL for PostgreSQL Only) When set to TRUE, causes a PREPARE EXEC statement to be executed before the OPEN statement when a CURSOR is declared with OPEN and FETCH statements.</p> <p>NOTE: Attention should be taken when applying the ImmediateCursor preprocessor parameter. Since the full results of the cursor are returned before the OPEN statement, this parameter should only be applied for cursors returning small numbers of lines.</p>	<p>Declare cursor is purely declarative which enables it to be placed in the DATA DIVISION, which some legacy applications depend on.</p> <p>CitOESQL optimizes cursor behavior for SELECT INTO statements and OPEN CURSOR statements.</p>
<p><code>:IncludeSearchPath=<Path></code></p> <p>Default [Current Working Directory] A comma, or semicolon separated list of the directories in which CitSQL will look for Include files.</p>	<p>COBCPY environment variable plus current directory.</p>
<p><code>:LibName=<libname></code></p> <p>Defaults are: RCQMYSQL (MySQL) and RCQPGSQL (PostgreSQL).</p>	<p>No equivalent. The runtime library <code>odbcw32.dll</code> (or its Linux equivalent shared object) may not be renamed.</p>
<p><code>:LogMode=[TRUE/FALSE]</code></p> <p>Default is: [FALSE] When set to TRUE the runtime component creates a log file called <code>RCQDLL.log</code> that traces all SQL operations.</p>	<p><code>-TRACELEVEL=<number></code></p>

CitSQL	CitOESQL
<p><code>:MaxMem=<number megabytes></code></p> <p>Default is: 100 Allocates memory for the precompilation of very large source files.</p>	<p>No equivalent.</p>
<p><code>:NoRecCode=<numeric></code></p> <p>Default is 1403. Allows mapping of value returned to indicate the end of a FETCH statement.</p>	<p>Default is 100 as defined by ANSI. - ERRORMAP can be used to override default 100 value.</p>
<p><code>:Prefetch=<numeric></code></p> <p>Allows for the prefetch of records in a network transaction, where there is a whole number that represents the number of records to read in a networked transaction. The Prefetch option is available only with PGSQL.</p>	<p><code>-PF_RO_CURSOR</code> <code>-PF_UPD_CURSOR</code></p> <p>Prefetch sizes may be set separately for read only and updatable cursors. High values may benefit read only cursors but can cause concurrency conflicts for updatable cursors. Available with all databases.</p>
<p><code>:QuoteTranslation=<pattern></code></p> <p>Default is: QDB</p> <p>Allows mapping of single quotes, double quotes, and back quotes. By default, quotes are unchanged, which corresponds to a default value of QuoteTranslation=QDB.</p>	<p>No equivalent.</p> <p>All major databases allow use of single and double quotes consistently in line with ANSI.</p> <p>Mapping of back quotes could be helpful to applications developed for Microsoft Access being ported to other databases.</p>
<p><code>:SelectPrepare=[True/False]</code></p> <p>(CitSQL for PostgreSQL Only) Default is TRUE. Now, the preprocessor causes the PREPARE EXEC statement to be executed prior to the OPEN statement and the results stored. When set to False, the former behavior is applied.</p>	<p>No similar requirement.</p>

CitSQL	CitOESQL
<p><code>:StandardPrefix=<prefix></code></p> <p>Default is: [None] Characters prefixed to generated data items.</p>	<p>Currently all generated data items are prefixed by 'PCS-' (for precompiler services).</p>
<p><code>StepLimit=<numeric></code></p> <p>The CitSQL parser is based on a Backtracking technology. In order to do this, it must set a limit on the number of cases it must be able to consider. You can control this limit with the <code>:StepLimit</code> option. Normally you will not need to use the <code>:StepLimit</code> option.</p> <p>The CitSQL parser is based on a Backtracking technology where it must set a limit on the number of cases it must be able to consider. You can control this limit with the <code>:StepLimit</code> option. In general you will not need to use the <code>:StepLimit</code> option.</p>	<p>No similar requirement</p>
<p><code>;StrictMode=[TRUE/FALSE]</code></p> <p>Default is: [FALSE] When set to TRUE, the CitSQL precompiler aborts in cases where it does not recognize an SQL syntax in an EXEC statement.</p>	<p><code>- [NO]CHECK (plus</code> <code>-DB=<connectionName> and -</code> <code>PASS=<userid>.<password>)</code></p> <p>When set a database connection will be used at compile time to validate SQL statements.</p>
<p><code>:StrictPictureMode=[TRUE/FALSE]</code></p> <p>Default is: [FALSE] When set to TRUE, the CitSQL precompiler aborts in cases where it does not recognize a PICTURE clause.</p>	<p>No equivalent.</p> <p>The preprocessor generates an error if a host variable is used that does not have a data type acceptable for use as a host variable.</p>

CitSQL	CitOESQL
<p><code>: TargetPattern=<pattern></code></p> <p>Describes a group of tokens that can be strung together as components to describe the location and naming convention applied to the precompiled target file.</p>	<p>No equivalent. Output files are always generated in the same location as source files and with an extension of '.cbp' except when COBOL-IT's preprocess directive is used.</p>
<p><code>: TrimMode=[TRUE/FALSE]</code></p> <p>Default is: [FALSE] When set to TRUE, alphanumeric (PIC X) strings that are passed to the database are first trimmed, (right space removed), so that the data in the database does not have trailing spaces.</p>	<p><code>-PICXBINDING={DEFAULT PAD TRIM TRIMALL FIXED VARIABLE}</code></p>
<p><code>: TruncComments=[TRUE/FALSE]</code></p> <p>When set to TRUE, Comments are truncated after column 72. When set to FALSE, comments are not truncated after column 72.</p>	<p>No equivalent.</p>
<p><code>: UTFInput=[TRUE/FALSE]</code></p> <p>When set to TRUE, specifies that the source code contains literals encoded in UTF-8.</p>	<p>No equivalent.</p>

3.1.3 Host Variables

CitSQL	CitOESQL
<p>CitSQL supports the non-COBOL USAGE Clauses:</p> <p>USAGE [LONG]VARCHAR USAGE [LONG] VARRAW USAGE VARYING</p>	<p>CitOESQL supports the non-COBOL USAGE Clauses:</p> <p>USAGE [LONG]VARCHAR USAGE [LONG] VARRAW USAGE VARYING</p> <p>Supported SQL [TYPE] [IS]: DATE, DATE-RECORD TIME, TIME-RECORD TIMESTAMP, TIMESTAMP-RECORD TIMESTAMP-OFFSET, TIMESTAMPOFFSET-RECORD BINARY, VARBINARY, LONG-VARBINARY CHAR, LONG-VARCHAR CHAR-VARYING</p>

3.1.4 ESQL features

CitSQL	CitOESQL
N/A	<p>Passes all the NIST ANSI ESQL compliance tests.</p> <p>Includes support for dynamic SQL.</p> <p>Multiple documented extensions to ANSI ESQL:</p> <ul style="list-style-type: none"> - Array insert and fetch - GET DIAGNOSTICS - SAVEPOINT support <p>Directives and constants in source code based on MF \$SET syntax.</p> <p>Conditional compilation based on MF COBOL conditional compilation built into the precompiler.</p>

3.1.5 Dependencies and Limitations

CitSQL	CitOESQL
N/A	<p>Requires ODBC on Windows and Linux.</p> <p>Database ODBC drivers may depend on database client libraries.</p> <p>Requires 3rd party ODBC Driver Manager for Linux (UnixODBC recommended).</p> <p>32 and 64 bit support on Linux and Windows.</p>

3.1.6 SQL Statement Differences and Limitations

CitSQL	CitOESQL
N/A	<p>Does not support CitSQL EXEC SQL CONNECT statement syntax directly, however, does provide 6 flexible ODBC alternatives.</p> <p>Does not support CitSQL EXEC SQL DECLARE hostvar VARIABLE CCSID xxxxx statement.</p> <p>Does not support CitSQL PIC N USAGE VARCHAR USAGE CLAUSE.</p>

4. User Guide

4.1 Modes of Operation

CitOESQL can be used as a standalone preprocess that executes before and separately from cobc, or in conjunction with cobc's `-preprocess` directive. This latter mode is referred to as integrated precompilation.

4.1.1 Standalone precompilation

When used standalone, you can use `$DISPLAY` and `$SET` statements in source code to manage CitOESQL directives and set constants. CitOESQL handles copybook expansion, constant setting and conditional compilation.

When debugging, you will see the code generated by CitOESQL.

4.1.2 Integrated precompilation

With integrated precompilation the following steps take place: - cobc reads the source code in an initial pass that handles constants and conditional compilation and writes the updated source code to a temporary file. This process also adds metadata comment lines to allow the debugger to locate the original, un-preprocessed source code

- cobc executes the script specified by the `-preprocess` option to execute CitOESQL passing it the name of the temporary file generated in the previous step and another temporary filename to be used for the precompiled output.
- CitOESQL preprocesses EXEC SQL statements and writes the preprocessed output source code to the temporary file requested by cobc.
- cobc executes the remainder of the compilation process.

When using integrated precompilation you will see the original source code in the debugger.

When using integrated precompilation:

- Do not use `$SET` statements in source code to set CitOESQL directives, these will be ignored and have no effect.
- Do not use `$DISPLAY` statements in the source code, in this case cobc will output a warning.

- Do not use conditional compilation that depends on constants set via CitOESQL directives or on the CitOESQL command line.
- Set all constants required for conditional compilation on the cobc command or in source code using \$SET CONSTANT statements that are compatible with cobc.
- Set CitOESQL directives on the CitOESQL command line or in a directives file for CitOESQL (via USE directive(s) on the CitOESQL command line).

To set CitOESQL directives you must edit the script file used to invoke CitOESQL. A sample script file is provided in %COBOLITDIR%\bin\runcitoesql.bat on Windows and \$COBOLITDIR/bin/runcitoesql.sh on Linux. When opening the script for integrated precompilation, cobc will search the current directory and %COBOLITDIR%\bin\ on Windows and \$COBOLITDIR/bin/ on Linux if no path is specified. Alternatively, you may specify an absolute or relative path for your script file. It will often be convenient to copy and rename the default script file to your source code directory and to make this directory current when compiling.

In the script file you should place CitOESQL directives after the directive and before the final two command line parameters (these are the input and output files specified by cobc).

4.2 Command line syntax

The command line for CitOESQL takes the following form:

```
citoesql [directive1 directive2 ...] [@]filename1 [@] filename2 ...
```

Each directive specified on the command line must be preceded by a hyphen.

filename1 filename2 ... is normally a sequence of filenames. Filenames not preceded by an '@' character are assumed to be COBOL source files. Each COBOL file will be preprocessed and if there are no errors detected an output file with the same name but a .cbp extension will be written.

If a filename starts with an '@' character, the file is assumed to be a text file containing a list of input COBOL filenames, one per line.

If the first precompiler directive on the command line is `P`, then exactly two filenames must be supplied. The first is the input filename and the second is the output filename. This directive should only be used in the script used by the COBOL-IT COBOL compiler in association with the `preprocess=<script>` option.

4.2.1 Placing Precompiler Directives in Files

You can place precompiler directives in a file as well as the command line or in source code via the USE directive.

In a directives file:

blank lines or lines with a hash symbol (#) or semicolon (;) in column 1 are ignored

directives optionally be preceded by one or two hyphens(-)

directives may be grouped together on a line and separated by a space or comma(,)

directives may be grouped together and wrapped in a SQL directive thus

```
SQL(directive1 [,] directive2 ...)
```

4.3 Source Code Formats

4.3.1 Source code formats

CitOESQL supports the following source code formats:

- Fixed, which corresponds to traditional COBOL with the source code in columns 8-72 and the indicator in column 7.
- Variable, which extends the right margin from column 72 to column 256 and the indicator column remains in column 7.
- Free, or Terminal, format. In this format the indicator in column 1 is optional, source code can start in column 1 or 2 and extend to column 248.

The source code format is specified by the SOURCEFORMAT directive, for example:

```
citoesql -sourceformat=variable gs.cbl
```

on the command line, or

```
$SET SQL(sourceformat=variable)
```

at the start of a source file, or

```
sourceformat=variable
```

or

```
SQL(sourceformat=variable)
```

in directives file.

4.4 Directive Syntax

4.4.1 Directive syntax

Directives are case insensitive. The format of a directive is one of:

```
[[NO]directiveName[ =directiveSetting]
```

```
[NO]directiveName[ (directiveSetting)]
```

```
[NO]directiveName[ "directiveSetting"]
```

```
[NO]directiveName[ 'directiveSetting']
```

Note

On Windows double quotes on command lines must be escaped by a backslash character.

On a command line a directive must be preceded by one or two hyphens with no whitespace between the hyphen(s) and the directive.

4.5 Control statements in source

You can control precompilation by placing control statements in source files. The \$SET statement is used to set constants and directives. Other control statements provide support for conditional compilation and message output.

4.5.1 Directives

You can set directives in source code using a \$SET control statement thus

```
$SET directiveSetting
```

4.5.2 Constants

You can set a constant in source code using a \$SET CONSTANT control statement.

The following formats are compatible with both standalone and integrated precompilation:

```
$SET CONSTANT constantName [=] constantValue
```

```
$SET CONSTANT constantName [=] "constantValue"
```

CitOESQL also supports the following formats:

```
$SET CONSTANT(constantName [=] constantValue)
```

```
$SET CONSTANT(constantName [=] "constantValue")
```

```
$SET CONSTANT(constantName [=] 'constantValue')
```

To set a constant in a directives file or on the CitOESQL command line use the following formats:

```
[-][-]CONSTANT(constantName [=] constantValue)
```

```
[-][-]CONSTANT(constantName [=] "constantValue")
```

```
[-][-]CONSTANT(constantName [=] 'constantValue')
```


4.5.3 Conditional compilation

CitOESQL supports conditional compilation using `$IF`, `$ELSE` and `$END` statements in standalone mode. These are compatible with the conditional compilation support offered by `cobc`.

CitOESQL also support the following variant, which is not supported by `cobc`:

```
$IF constantName [NOT] DEFINED
```

4.5.4 Messages

In standalone mode you can output a message at compile time using a `$DISPLAY` statement. The message displayed starts with the first non-blank character following `$DISPLAY` and ends with the last non-blank character on the same line.

4.6 Programming

4.6.1 Syntax Checking Options

CitOESQL is an open implementation for the ANSI Embedded SQL standard, and as such can be used with a wide variety of databases, each of which can accept different and sometimes unique SQL syntax. To accommodate these differences, by default, CitOESQL does minimal SQL syntax checking at compile time. You can increase the level of CitOESQL SQL syntax checking at compile time by using the `SQL(CHECK)` directive with other associated directives.

`SQL(CHECK)` connects to the database during compilation, and asks the database to validate SQL syntax with the existing database SQL objects such as tables, columns, etc.

In addition to `SQL(CHECK)`, you must also specify the `SQL(DB)` directive, and optionally the `SQL(PASS)` directive. This combination ensures a successful connection to your database at compile time and returns applicable SQL syntax errors.

Note

When using `SQL(CHECK)`, we suggest that you connect to a local rather than a remote database, as network access could compromise compilation speed.

In addition to the topics below, see the **CHECK** compiler directive topic in the [CitOESQL Reference Manual](#) for more information on the `SQL(CHECK)` compiler directive option.

4.6.2 SQL(CHECK) and Schema Objects

All databases contain Schemas that include SQL objects like tables, columns, views, and temporary tables. If all the SQL objects in a database are available at compile time, using the SQL(CHECK) directive ensures that all SQL syntax is fully checked by the database during compilation.

However, some schema objects might not be available in the database at compile time. In this case, CitOESQL offers two solutions:

SQL(IGNORESCHEMAERRORS) directive

Use this directive with SQL(CHECK) when tables or other SQL objects do not exist in the database. The addition of SQL(IGNORESCHEMAERRORS) enables CitOESQL to ignore invalid object reference errors returned by the database and continue compilation.

[ALSO CHECK] and [ONLY CHECK] statement prefixes

Use these statement prefixes on individual SQL statements in your code if you want CitOESQL to create SQL objects in the database at compile time. With the objects in the database, all invalid object reference errors are returned during compilation, and optionally at run time as well.

For more information on [ALSO CHECK] and [ONLY CHECK], see SQL Statement Prefixes for SQL(CHECK).

4.6.3 SQL(CHECK) Command-line Options

You can improve SQL syntax checking in some scenarios by combining SQL(CHECK) with additional compiler directive options and/or using a local database.

SQL(CHECK) with SQL(DB) and, optionally, SQL(PASS)

When you use the SQL(CHECK) and SQL(DB) directives, together with SQL(PASS) if necessary, CitOESQL opens a connection to a data source at compile time and uses the data source to perform additional checking. This is the recommended way to use CitOESQL and is much more reliable at detecting errors. In addition to detecting syntax errors that are specific to a particular data source, it can also detect misspelled names and invalid use of reserved words.

SQL(CHECK), local database, deployment schema

CitOESQL is at its most effective when you use SQL(CHECK) with a local database that uses the same schema that is used when the application is deployed. This combination compiles programs faster than accessing a networked server for compile-time checking.

SQL(CHECK), local database, no deployment schema

When you do not have access to a data source with the deployment schema installed you can still use SQL(CHECK) to perform additional syntax checking, but you must also use SQL(IGNORESCHEMAERRORS) to avoid errors for invalid name use.

SQL(IGNORESCHEMAERRORS) is also helpful when your program uses temporary tables that exist only at run time.

4.6.4 SQL Statement Prefixes for SQL(CHECK)

To enable complete SQL syntax checking at compile time when tables or temporary tables are not in your database, CitOESQL provides SQL statement prefixes that enable you to execute specific SQL statements at compile time and optionally at run time also.

A statement prefix is coded directly into an EXEC SQL statement and executed only when compiled with SQL(CHECK).

Syntax:

```
EXEC SQL [statementPrefix] [errorFlag[...]] SQLStatement END-EXEC
```

Parameters:

statementPrefix

[ALSO CHECK]

Statement prefix that instructs SQL(CHECK) to execute the following *SQLStatement* both at compile time and a run time.

[ONLY CHECK]

Statement prefix that instructs SQL(CHECK) to execute the following *SQLStatement* at compile time only.

[NOCHECK]

Statement prefix that turns off the effects of the SQL(CHECK) directive for the associated *SQLStatement* only.

errorFlag

[IGNORE ERROR]

Overrides the default behavior of providing a success or error return code upon *SQLStatement* execution, and instead ignores all errors at compile time and proceeds with compilation.

[WITH WARNING]

Overrides the default behavior of providing a success or error return code upon *SQLStatement* execution, and instead returns all errors as warnings at compile time, and proceeds with compilation.

SQLStatement

An SQL statement that conforms to the following:

- A DDL statement such as CREATE TABLE or the specific DML statements INSERT, DELETE or UPDATE
- No host variables used
- Written in a syntax understood by the DBMS vendor

Example 1:

Create a SQL Server temporary table at compile time only so that subsequent SQL that references this table is fully syntax checked by the CitOESQL SQL(CHECK) directive during compilation.

```
EXEC SQL [ONLY CHECK]
create table #temp(col1 int,col2 char(32))
END-EXEC
```

Example 2:

Create a SQL Server temporary table at both compile time and execution time so that, in addition to full compile-time syntax checking, the table is created at execution time.

```
EXEC SQL [ALSO CHECK]
create table #temp(col1 int,col2 char(32))
END-EXEC
```

Example 3:

Create test data at compile time with Oracle whether or not the table exists at compile time.

```
EXEC SQL [ONLY CHECK IGNORE ERROR]
Drop table TT
END-EXEC
```

```
EXEC SQL [ONLY CHECK]
create table TT (col1 int)
END-EXEC
```

```
EXEC SQL [ONLY CHECK]
Insert into table TT values (1)
END-EXEC
```

4.6.5 Tuning Performance

Cursor Types and Performance

CitOESQL handles ambiguously declared embedded SQL cursors by making them forward and readonly, and incapable of retrieving locks. This change improves performance and efficiency, optimizing CitOESQL as most DBMS SQL cursor access plans do.

You can further optimize CitOESQL by using the BEHAVIOR directive, which enables you to change your embedded SQL cursor characteristics (including prefetch processing) without changing any code in your SQL application sources.

Statement Cache

Embedded SQL statements are optimized for repeat execution. When first executed a statement is prepared at the data source. This is analogous to program compilation and means that information is retained about how to execute the statement so that it does not have to be recompiled on subsequent executions.

Prepared statements consume memory on both the client and server; CitOESQL maintains a cache to limit how much memory is consumed by prepared statements. The cache is managed on a Least Recently Used (LRU) basis. When the cache reaches its limit the least recently used statement that can safely be removed from the cache is replaced with the statement currently being executed. Any resources used by the replaced prepared statement are freed.

The default cache size is 20 which is quite small, however, this limit has the ability to avoid problems with some database products that have low limits on server resource consumption. The size of the cache can be changed via the STMTCACHE directive.

In practice, a large batch program may benefit from a statement cache size of 300 or possibly more. Many factors affect the optimum cache size, so it is best to experiment. Initial increments will generally improve performance of programs that use connections with long lifespans, but eventually additional increments benefits may reduce performance. Finding the optimum cache size may take a little effort.

Datetime Data Type Handling

By default, CitOESQL supports ODBC/ISO 8601 formats for all input and output character host variables associated with datetime columns in your DBMS.

For example, when using a SQL Server DBMS, the default data type formats for character host variables are:

SQL Server Data Type	ODBC/ISO 8601 Format
date	yyyy-mm-dd
time	hh:mm:ss
datetime2	yyyy-mm-dd hh:mm:ss.fffffffff

In addition to SQL Server, these formats generally apply to other DBMS vendors that accept ISO 8601 formats. CitOESQL also supports alternative formats for both input and output character host variables. We provide several SQL compiler directive options that enable you to specify alternative formats that override the default.

Input Host Variables - DETECTDATE

The DETECTDATE SQL compiler option directive instructs CitOESQL to examine the contents of PIC X character input host variables, looking for data that matches the default ISO 8601 formats. You can override the default formats by specifying one or more additional directives:

Note

For complete information on each directive, see its corresponding topic under CitOESQL Directives section in the [CitOESQL Reference Manual](#).

DATE

Specify an alternative DATE format.

DATEDELIM

Specify an alternative delimiter for date columns.

- When used with DATE, the alternative delimiter is applied to the alternative date format specified.
- When used without DATE, the alternative delimiter is applied to the ISO 8601 date format.

TIME

Specify an alternative TIME format.

TIMEDELIM

Specify an alternative delimiter for time columns.

- When used with TIME, the alternative delimiter is applied to the alternative time format specified.
- When used without TIME, the alternative delimiter is applied to the ISO 8601 time format.

TSTAMPSEP

Specify a one-character delimiter used between the date and time portions of your input host variables.

The dash character instructs CitOESQL to look for a specific set of delimiters, including a dash, a space, and a T. For example, if you do not specify any alternative date or time formats, and you set TSTAMPSEP to a dash character (-), CitOESQL recognizes the following formats in your input host variables:

- `yyyy-mm-dd-hh.mm.ss.ffffff`
- `yyyy-mm-dd hh.mm.ss.ffffff`
- `yyyy-mm-dd hh:mm:ss.ffffff`
- `yyyy-mm-ddThh.mm.ss.ffffff`
- `yyyy-mm-ddThh:mm:ss.ffffff`

All other characters instruct CitOESQL to search for that specific character between each date and time format, where the date portion is delimited by a dash character (-) and the time portion is delimited by a colon (:).

If you do not specify TSTAMPSEP, CitOESQL defaults to searching for a space character as the delimiter between the date and time formats, where the date portion is delimited by a dash character (-) and the time portion is delimited by a colon (:).

Guidelines for using DETECTDATE

Use of the DETECTDATE directive can create significant processing overhead. To minimize this, we recommend that you follow the guidelines presented in the following usage scenarios:

Scenario	DETECTATE option
My application uses date, time and datetimes values in PIC X input host variables, but I am happy with the supported ODBC/ISO 8601 formats and have no use for alternative formats.	Not required. *
My application uses date, time and datetime values in PIC X input host variables, but I only use those values in date, time, or datetime columns in my database.	CLIENT
My application uses ODBC escape sequences for date, time, and datetimes values in PIC X input host variables, but I only use those values in date, time, or datetime columns in my database.	CLIENT
My application uses date, time, and datetime values in PIC X input host variables, but I only use those values in character columns in my database. Also, my SQL does not use either implicit or explicit characters for date, time, or datetime2 data types.	Not required. Do not use DETECTDATE, DATE, or TIME SQL compiler directive options.
I only use SQLTYPE host variables with date, time and datetime columns, and never use PIC X host variables with date, time or datetime columns.	Not required. **
My application uses date, time and datetime values in PIC X input host variables, and I use those values both in character columns and in date, time and datetime columns in my database, and my character columns might use data in formats that could be confused with the formats for date, time or datetime values.	SERVER

* We recommend that you use DETECTDATE when also using TIME values with Oracle.

** Optionally, you can use alternative datetime SQL compiler directive options.

Note

For complete information on all DETECTDATE options, see the DETECTDATE SQL compiler directive option in the [CitOESQL Reference Manual](#).

Output Host Variables

By default, CitOESQL returns date, time and datetime data types in the ISO 8601 default format. You can override the default format by specifying additional CitOESQL directives as follows:

DBMS Data Type	ODBC/ISO 8601 Format	CitOESQL Directives
date	yyyy-mm-dd	DATE, DATEDELIM
datetime	yyyy-mm-dd hh:mm:ss.ffffff	TSTAMPSEP

Changing the error code logic in an application containing logic originally designed for a specific database can be cumbersome. Consider these scenarios:

- My code expects Oracle SQLCODE 1403 at end of result set processing and not SQLCODE 100 as produced by other databases.
- My code expects z/OS DB2 SQLCODE-811 when a SELECT INTO statement returns more than one row.
- My code does not expect data truncation warnings after a FETCH statement, but my new database sets SQLCODE 1.
- When inserting a row that results in a duplicate key error, my code expects original database error codes.

These are just a few simple examples, however, error code mapping allows maximum flexibility in preserving the current error handling in your application code. When you provide search criteria based on what the new database returns in error situations, (using value 0 when SQLCODE or SQLSTATE values are returned that do not matter), and specify the error values for the original database, you can ensure that your application receives the error codes it expects.

When error mapping is enabled, it is processed after an embedded SQL statement completes execution. If SQLCODE is non-zero or SQLSTATE is not 00000, the error map is used to determine if SQLCODE, SQLSTATE, and optionally the associated error message, should be replaced with values from the error map. This is done by scanning error map records in order until either of the following conditions are met:

- It reaches the end of the map, in which case SQLCODE, SQLSTATE, and the error message are left unchanged.
- A match is made on some combination of SQLCODE, SQLSTATE, and a substring present in the error message.

SQL error mapping files

You control error mapping using an error mapping file. This is a simple text file that specifies which error conditions to map, the replacement values for SQLCODE and SQLSTATE, and optionally replacement values for the error message, including complete suppression of the error message.

You can specify mappings based on the returned values of SQLCODE, SQLSTATE or a substring within the error message, or any combination of these.

LOCATION

The default location for mapping files is %COBOLITDIR%\etc (Windows) or \$COBOLITDIR/etc (Linux)

Note

You can override the default location using the CIT_ERRORMAP_PATH system environment variable.

FILENAME

You can name an SQL error mapping file using any prefix you choose; however, all error mapping files must have an `.emap` extension.

CONTENTS

Each record in a mapping file contains the following values in this order, delimited by commas:

```
{SC-ret-val|0},{SS-ret-val|0},[msg-substr],SC-repl-val,SS-repl-val,[msg-substr-repl-val]
```

Where:

SC-ret-val|0

The returned database value for SQLCODE, or 0 (zero), to indicate that the returned database SQLCODE value does not matter.

SS-ret-val|0

The returned database value for SQLSTATE, or 0 (zero), to indicate that the returned database SQLSTATE value does not matter.

msg-substr

The returned database error message substring, if applicable. Specify a string of characters that appear in the message returned by the database. The error is mapped if the substring is present in the error message, and when the SQLCODE and SQLSTATE conditions are also satisfied. The following syntax rules apply when providing a substring:

- If the substring contains a comma, enclose the entire substring in single (') or double (") quotes
- Message substrings are case sensitive.

Note

The full error message is used for the substring search rather than the 70 bytes subset returned in SQLERRMC.

Other than providing a substring, you also have these two options:

- Omit a value by including a space before the next comma delimiter. The original message is returned, effectively switching off error message replacement for substrings.
- Specify a single tilde (~) character. This populates the message-receiving field, consisting of SQLERRMC, MFSQLMESSAGETEXT (or the host variable for

MESSAGE_TEXT with GET DIAGNOSTICS), with spaces. SQLERRML in the SQLCA is also set to zero rather than the number of characters returned in SQLERRMC.

SC-repl-val

Original database replacement value for SQLCODE.

SS-repl-val

Original database replacement value for SQLSTATE.

msg-substr-repl-val

Replacement value for the error message, if applicable. Syntax rules for msg-substr also apply to msg-substr-repl-val. When omitted, the initial error message is not replaced. Use a single tilde (~) character to completely suppress the message.

SQL error mapping record examples

EXAMPLE 1:

This example is based on a migration from DB/2 for z/OS to PostgreSQL where a SELECT INTO statement returns more than the expected one row. The following mapping file entry changes the returned SQLCODE from 1 to -811 while leaving the PostgreSQL error intact. It does this when PostgreSQL returns SQLCODE 1 and SQLSTATE 21000 (the matching criteria):

```
1, 21000, , -811, 21000
```

EXAMPLE 2:

This example is based on a migration from DB/2 for z/OS to PostgreSQL. The record maps errors that contain the string "duplicate" when a primary key or unique constraint error occurs, and changes the error message to "Unique constraint violation". Notice that the return values PostgreSQL provides for both SQLCODE and SQLSTATE do not matter. The search criteria is based solely on the returned PostgreSQL error message alone:

```
0, 00000, "duplicate", -803, 22002, Unique constraint violation
```

EXAMPLE 3: ODBC generates a warning when a host variable is smaller than the returned value. If an application tests for SQLCODE being non-zero rather than negative, this can break application logic. To completely suppress the warning condition, including the error message, the following record matches warnings where SQLSTATE has the value 01004:

```
0, 01004, , 0, 00000, ~
```

EXAMPLE 4: As an alternative to Example 3, when migrating a legacy application to an environment using UTF-8 and you want to test whether your host variables are large enough, the following record changes this warning to an error with SQLCODE -55 and SQLSTATE "22XYZ":

```
0, 01004, , -55, 22XYZ, Host variable too small
```

SQL error mapping enablement

Use the CitOESQL ERRORMAP compiler directive option to enable error mapping, as documented in the CitOESQL Directive section of the [CitOESQL Reference Manual](#).

If you intend to use multiple error mapping files in one program, use the SQL Statement SET ERRORMAP, as documented in SQL Statements section of the [CitOESQL Reference Manual](#).

5. Reference Manual

5.1 Developing SQL Applications

The following topics describe the programming features available for SQL applications in general.

- **Embedded SQL**

Instructions on how to embed SQL statements into your programs.

- **Host Variables**

The purpose of host variables, how to declare them, and how to use them in your SQL applications.

- **Cursors**

The purpose of cursors, how to declare them, and how to use them in SQL applications.

- **Data Structures**

The purpose and use of the SQLCA and SQLDA data structures available for SQL applications.

- **Dynamic SQL**

An explanation of how dynamic SQL works, and a description of its purpose, advantages, and use.

5.1.1 Embedded SQL

The CitOESQL preprocessor works by taking the SQL statements that you have embedded in your COBOL program and converting them to the appropriate function calls to the database.

Keywords:

In your COBOL program, each embedded SQL statement must be preceded by the introductory keywords:

```
EXEC SQL
```

and followed by the keyword:

```
END-EXEC
```

For example:

```
EXEC SQL
    SELECT au_lname INTO :lastname FROM authors
    WHERE au_id = '124-59-3864'
END-EXEC
```

The embedded SQL statement can be broken over as many lines as necessary following the normal COBOL rules for continuation, but between the EXEC SQL and END-EXEC keywords you can only code an embedded SQL statement; you cannot include any ordinary COBOL code.

The case of embedded SQL keywords in your programs is ignored. You can use all upper-case, all lower-case, or a combination of the two. For example, the following are all equivalent:

```
EXEC SQL CONNECT
exec sql connect
Exec Sql Connect
```

Cursor names, statement names, and connection names:

The case of cursor names, statement names and connection names must match that used when the variable is declared. For example, if you declare a cursor as C1, you must always refer to it as C1 (and not as c1).

The settings for the database determines whether such things as connection names, table and column names, are case-sensitive.

SQL identifiers:

Hyphens are not permitted in SQL identifiers such as table and column names.

SQL identifiers are typically restricted regarding which characters they support. Typically, unquoted identifiers can only contain A-Z, 0-9 and underscore. Some databases might also allow lower-case characters, and/or @ and # symbols. If your SQL identifiers contain any other characters, such as a grave accent, spaces, or DBCS characters, they must be delimited. Refer to your database vendor documentation for more information, including the character to use as the delimiter.

SQL statements:

Most vendors provide SQL Reference documentation with their database software that includes full information about embedded SQL statements. Regardless of the database software, you should, for example, be able to perform the following typical operations using the statements shown:

Operation	SQL Statement(s)
Add data to a table	INSERT
Change data in a table	UPDATE
Retrieve a row of data from a table	SELECT
Create a named cursor	DECLARE CURSOR
Retrieve multiple rows of data using a cursor	OPEN, FETCH, CLOSE

A full syntax description is given for each of the supported embedded SQL statements, together with an example of its use, in the topics under [Embedded SQL](#).

5.1.2 Host Variables

Host variables are data items defined within a COBOL program. They are used to pass values to and receive values from a database. Host variables can be defined in the File Section, Working-Storage Section, Local-Storage Section or Linkage Section of your COBOL program and can be coded using any level number between 1 and 48.

A host variable can be input or output:

Input host variables - to specify data to be transferred from the COBOL program to the database •

Output host variables

to hold data to be returned to the COBOL program from the database

To use host variables, you must declare them in your program and then reference them in your SQL statements.

A host variable can be defined as any of the following types:

- [Simple Host Variables](#)

To store and retrieve a single string of data.

- [Host Arrays](#)

To store and retrieve multiple rows of data.

- [Indicator Variables](#)

A companion variable that stores null value and data truncation information.

- [Indicator Arrays](#)

A companion array used to store null value and data truncation information for multiple rows.

Simple Host Variables

Before you can use a host variable in an embedded SQL statement, you must declare it.

Declaring simple host variables

Generally, host variable declarations are coded as data items bracketed by the embedded SQL statements `BEGIN DECLARE SECTION` and `END DECLARE SECTION`. The following rules also apply:

You can use groups of data items as a single host variable. However, a group item cannot be used in a `WHERE` clause.

CitOESQL trims trailing spaces from character host variables. If the variable consists entirely of spaces, CitOESQL does not trim the first space character because some servers treat a zero-length string as `NULL`.

With CitOESQL, you can use COBOL data items as host variables even if they have not been declared using `BEGIN DECLARE SECTION` and `END DECLARE SECTION`.

Host variable names must conform to the COBOL rules for data items.

Host variables can be declared anywhere that it is legal to declare COBOL data items.

Referencing simple host variables

You reference host variables from embedded SQL statements. When you code a host variable name into an embedded SQL statement, it must be preceded by a colon (`:`) to enable the compiler to distinguish between the host variable and tables or columns with the same name.

EXAMPLE:

```

EXEC SQL
    BEGIN DECLARE SECTION
END-EXEC
01 id          pic x(4).
01 name        pic x(30).
01 book-title  pic x(40).
01 book-id     pic x(5).
EXEC SQL
    END DECLARE SECTION
END-EXEC
. . .
    display "Type your identification number: "
    accept id.
* The following statement retrieves the name of the
* employee whose ID is the same as the contents of
* the host variable "id". The name is returned in
* the host variable "name".
    EXEC SQL
        SELECT emp_name INTO :name FROM employees
        WHERE emp_id=:id
    END-EXEC
    display "Hello " name.
* In the following statement, :book-id is an input
* host variable that contains the ID of the book to
* search for, while :book-title is an output host
* variable that returns the result of the search.
    EXEC SQL
        SELECT title INTO :book-title FROM titles
        WHERE title_id=:book-id
    END-EXEC

```

Host Arrays

An array is a collection of data items associated with a single variable name. You can define an array of host variables (called host arrays) and operate on them with a single SQL statement.

You can use host arrays as input variables in INSERT, UPDATE and DELETE statements and as output variables in the INTO clause of SELECT and FETCH statements. This means that you can use arrays with SELECT, FETCH, DELETE, INSERT and UPDATE statements to manipulate large volumes of data.

Some of the benefits to using host arrays include:

- You can perform multiple CALL, EXECUTE, INSERT or UPDATE operations by executing only one SQL statement, which can significantly improve performance, especially when the application and the database are on different systems.

You can fetch data in batches, which can be useful when creating a scrolling list of information.

As with simple host variables, you must declare host arrays in your program and then reference them in your SQL statements.

Declaring host arrays

Host arrays are declared in much the same way as simple host variables using `BEGIN DECLARE SECTION` and `END DECLARE SECTION`. With host arrays, however, you must use the `OCCURS` clause to dimension the array.

Referencing host arrays

The following rules apply to coding host arrays into embedded SQL statements:

Just as with simple host variables, you must precede a host array name with a colon (:).

If the number of rows available is more than the number of rows defined in an array, a `SELECT` statement returns the number of rows defined in the array, and an `SQLCODE` message is issued to indicate that the additional rows could not be returned.

Use a `SELECT` statement only when you know the maximum number of rows to be selected. When the number of rows to be returned is unknown, use the `FETCH` statement.

If you use multiple host arrays in a single SQL statement, their dimensions must be the same.

CitOESQL does not support the mixing of host arrays and simple host variables within a single SQL statement. They must be all simple or all arrays.

For CitOESQL, you must define all host variables within a host array with the same number of occurrences. If one variable has 25 occurrences, all variables in that host array must have 25 occurrences.

- Optionally, use the `FOR` clause to limit the number of array elements processed to just those that you want. This is especially useful in `UPDATE`, `INSERT` and `DELETE` statements where you may not want to use the entire array. The following rules apply:

If the value of the `FOR` clause variable is less than or equal to zero, no rows are processed.

The number of array elements processed is determined by comparing the dimension of the host array with the `FOR` clause variable. The lesser value is used.

EXAMPLES:

The following example shows typical host array declarations and references.

```

EXEC SQL
    BEGIN DECLARE SECTION
END-EXEC
01 AUTH-REC-TABLES
    05 Auth-id OCCURS 25 TIMES PIC X(12).
    05 Auth-Lname OCCURS 25 TIMES PIC X(40).
EXEC SQL
    END DECLARE SECTION
END-EXEC.
. . .

EXEC SQL
    CONNECT USERID 'user' IDENTIFIED BY 'pwd'
        USING 'db_alias'
END-EXEC
EXEC SQL
    SELECT au-id, au-lname
        INTO :Auth-id, :Auth-Lname FROM authors
END-EXEC
display sqlerrd(3)

```

The following example demonstrates the use of the FOR clause, showing 10 rows (the value of :maxitems) modified by the UPDATE statement:

```

EXEC SQL
  BEGIN DECLARE SECTION
END-EXEC

01 AUTH-REC-TABLES
  05 Auth-id OCCURS 25 TIMES PIC X(12).
  05 Auth-Lname OCCURS 25 TIMES PIC X(40).
01 maxitems PIC S9(4) COMP-5 VALUE 10.
EXEC SQL
  END DECLARE SECTION
END-EXEC.

. . .
EXEC SQL
  CONNECT USERID 'user' IDENTIFIED BY 'pwd'
  USING 'db_alias'
END-EXEC
EXEC SQL
  FOR :maxitems
  UPDATE authors
    SET au_lname = :Auth_Lname
    WHERE au_id = :Auth_id
END-EXEC
display sqlerrd(3)

```

Indicator Variables

Use indicator variables to:

- Assign null values
- Detect null values
- Detect data truncation

Unlike COBOL, SQL supports variables that can contain null values. A null value means that no entry has been made and usually implies that the value is either unknown or undefined. A null value enables you to distinguish between a deliberate entry of zero (for numerical columns) or a blank (for character columns) and an unknown or inapplicable entry. For example, a null value in a price column does not mean that the item is being given away free, it means that the price is not known or has not been set.

Important

When a host variable is null, its indicator variable has the value -1; when a host variable is not null, the indicator variable has a value other than -1.

Indicator variables serve an additional purpose if truncation occurs when data is retrieved from a database into a host variable. If the host variable is not large enough to hold the data returned from the database, the warning flag `sqlwarn1` in the `SQLCA` data structure is set and the indicator variable is set to the size of the data contained in the database.

Declaring indicator variables

Indicator variables are always defined as:

```
pic S9(4) comp-5.
```

Referencing indicator variables

Together, a host variable and its companion indicator variable specify a single SQL value. The following applies to coding a host variable with a companion indicator variable:

Both variables must be preceded by a colon (:).

Place an indicator variable immediately after its corresponding host variable.

- Reference the host variable and indicator variable in a `FETCH INTO` or `SELECT ...INTO` statement with or without an `INDICATOR` clause as follows:

```
:hostvar:indicvar
```

or

```
:hostvar INDICATOR :indicvar
```

You cannot use indicator variables in a search condition. To search for null values, use the `is null` construct instead.

EXAMPLES:

This example demonstrates the declaration of an indicator variable that is used in a `FETCH ...INTO` statement.

```

EXEC SQL
    BEGIN DECLARE SECTION
END-EXEC
01 host-var pic x(4).
01 indicator-var pic S9(4) comp-5.
EXEC SQL
    END DECLARE SECTION
END-EXEC
. . .

EXEC SQL
    FETCH myCursor INTO :host-var:indicator-var
END-EXEC

```

The following shows an embedded UPDATE statement that uses a saleprice host variable with a companion indicator variable, `saleprice-null`:

```

EXEC SQL
    UPDATE closeoutsale
        SET temp_price = :saleprice:saleprice-null,
            listprice = :oldprice
END-EXEC

```

In this example, if `saleprice-null` has a value of -1, when the UPDATE statement executes, the statement is read as:

```

EXEC SQL
    UPDATE closeoutsale
        SET temp_price = null, listprice = :oldprice
END-EXEC

```

This example demonstrates the use of the `is null` construct to do a search:

```
if saleprice-null equal -1
  EXEC SQL
    DELETE FROM closeoutsale
      WHERE temp_price is null
  END-EXEC
else
  EXEC SQL
    DELETE FROM closeoutsale
      WHERE temp_price = :saleprice
  END-EXEC
end-if
```

Indicator Arrays

Just as an indicator variable is used as a companion to a host variable, use an indicator array as a companion to a host array to indicate the null status of each returned row or to store data truncation warning flags.

EXAMPLES:

In this example, an indicator array is set to -1 so that it can be used to insert null values into a column:


```

01 ix PIC 99 COMP-5.
. . .

EXEC SQL
  BEGIN DECLARE SECTION
END-EXEC
01 sales-id      OCCURS 25 TIMES PIC X(12).
01 sales-name    OCCURS 25 TIMES PIC X(40).
01 sales-comm    OCCURS 25 TIMES PIC S9(9) COMP-5.
01 ind-comm      OCCURS 25 TIMES PIC S9(4) COMP-5.
EXEC SQL
  END DECLARE SECTION
END-EXEC.
. . .
PERFORM VARYING ix FROM 1 BY 1 UNTIL ix > 25
  MOVE -1 TO ind-comm (ix)
END-PERFORM.
. . .
EXEC SQL
  INSERT INTO SALES (ID, NAME, COMM)
    VALUES (:sales_id, :sales_name, :sales_comm:ind-comm)
END-EXEC

```

COBOL to SQL Data Type Mapping

SQL has a standard set of data types, but the exact implementation of these varies between databases, and many databases do not implement the full set.

Within a program, COBOL host variable declarations can serve both as COBOL host variables and as SQL database variables. To make this possible, the preprocessor converts COBOL data types to their equivalent SQL data types. We sometimes refer to this conversion process as mapping COBOL data types to SQL data types. The preprocessor looks for specific COBOL picture clause formats that identify those that require mapping to SQL data types. For mapping to be successful, you must declare your COBOL host variables using these specific COBOL picture clauses.

We provide SQL data types for the CitOESQL preprocessor. For complete information on each SQL data type and its required COBOL host variable formats, see the SQL Data Types and ODBC SQL/COBOL Data Type Mappings Reference topics.

SQL TYPES

Manipulating SQL data that involves date, time, or binary data can be complicated using traditional COBOL host variables, and traditional techniques for handling variable-length character data can also be problematic. To simplify working with this data, we provide the SQL TYPE declaration to make it easier to specify host variables that more closely reflect the natural data types of relational data stores. This allows more applications to be built using static rather than dynamic SQL syntax and can also help to optimize code execution.

Note

For a complete listing of available SQL TYPES, see the SQL TYPES reference topic.

EXAMPLE:

Defining date, time, and timestamp fields as SQL TYPES.

This example program shows date, time and timestamp escape sequences being used, and how to redefine them as SQL TYPES. It applies to CitoESQL:

working-storage section.

```
EXEC SQL INCLUDE SQLCA END-EXEC
01 date-field1      pic x(29).
01 date-field2      pic x(29).
01 date-field3      pic x(29).
```

procedure division.

```
EXEC SQL
CONNECT TO 'Net Express 4.0 Sample 1' USER 'admin'
END-EXEC
```

* If the Table is there drop it.

```
EXEC SQL
DROP TABLE DT
END-EXEC
```

* Create a table with columns for DATE, TIME, and DATE/TIME

* NOTE: Access uses DATETIME column for all three.

* Some databases will have dedicated column types.

* If you are creating DATE/TIME columns on another data

* source, refer to your database documentation to see how to * define the columns.

```
EXEC SQL
CREATE TABLE DT ( id INT,
myDate DATE NULL,
myTime TIME NULL,
myTimestamp TIMESTAMP NULL)
END-EXEC
```

* INSERT into the table using the ODBC Escape sequences

```
EXEC SQL
INSERT into DT values (1 ,
{d '1961-10-08'}, *> Set just the date part
{t '12:21:54' }, *> Set just the time part
{ts '1966-01-24 08:21:56' } *> Set both parts
)
END-EXEC
```

* Retrieve the values we just inserted

```
EXEC SQL
SELECT myDate
,myTime
,myTimestamp
```

```

        INTO      :date-field1
                ,:date-field2
                ,:date-field3
    FROM DT
    where id = 1
END-EXEC

```

* Display the results.

```

display 'where the date part has been set :'
        date-field1
display 'where the time part has been set :'
        date-field2
display 'NOTE, most data sources will set a default '
        'for the date part '
display 'where both parts has been set :'
        date-field3

```

* Remove the table.

```

EXEC SQL
    DROP TABLE DT
END-EXEC

```

* Disconnect from the data source

```

EXEC SQL
    DISCONNECT CURRENT
END-EXEC

```

```
stop run.
```

Alternatively, you can use host variables defined with SQL TYPES for date/time variables. Define the following host variables:

```

01 my-id          pic s9(08) COMP-5.

01 my-date        sql type is date.

01 my-time        sql type is time.

01 my-timestamp  sql type is timestamp.

```

and replace the INSERT statement with the following code:

```

*> INSERT into the table using SQL TYPE HOST VARS
      move 1                               to MY-ID
      move "1961-10-08"                     to MY-DATE
      move "12:21:54"                       to MY-TIME
      move "1966-01-24 08:21:56"           to MY-TIMESTAMP

EXEC SQL
      INSERT into DT value (
        :MY-ID
        , :MY-DATE
        , :MY-TIME
        , :MY-TIMESTAMP )
END-EXEC

```

5.1.3 Cursors

When you write code in which the results set returned by a SELECT statement includes more than one row of data, you must declare and use a cursor. A cursor indicates the current position in a results set, in the same way that the cursor on a screen indicates the current position.

A cursor enables you to:

- Fetch rows of data one at a time

- Perform updates and deletions at a specified position within a results set.

The example below demonstrates the following sequence of events:

1. The DECLARE CURSOR statement associates the SELECT statement with the cursor Cursor1.
2. The OPEN statement opens the cursor, thereby executing the SELECT statement.
3. The FETCH statement retrieves the data for the current row from the columns au_fname and au_lname and places the data in the host variables first_name and last_name.
4. The program loops on the FETCH statement until no more data is available.
5. The CLOSE statement closes the cursor.

```

EXEC SQL DECLARE Cursor1 CURSOR FOR
    SELECT au_fname, au_lname FROM authors
END-EXEC
. . .

EXEC SQL
    OPEN Cursor1
END-EXEC
. . .

perform until sqlcode not = zero
    EXEC SQL
        FETCH Cursor1 INTO :first_name, :last_name
    END-EXEC
    display first_name, last_name
end-perform
. . .

EXEC SQL
    CLOSE Cursor1
END-EXEC

```

Declaring a Cursor

Before a cursor can be used, it must be declared. This is done using the DECLARE CURSOR statement in which you specify a name for the cursor and either a SELECT statement or the name of a prepared SQL statement.

Cursor names must conform to the rules for identifiers on the database that you are connecting to, for example, some databases do not allow hyphens in cursor names.

```

EXEC SQL
    DECLARE Cur1 CURSOR FOR
        SELECT first_name FROM employee
        WHERE last_name = :last-name
END-EXEC

```

This example specifies a SELECT statement using an input host variable (`:last-name`). When the cursor OPEN statement is executed, the values of the input host variable are read and the SELECT statement is executed.

```

EXEC SQL
  DECLARE Cur2 CURSOR FOR stmt1
END-EXEC

. . .
move "SELECT first_name FROM emp " &
  "WHERE last_name=?" to prep.
EXEC SQL
  PREPARE stmt1 FROM :prep
END-EXEC

. . .
EXEC SQL
  OPEN Cur2 USING :last-name
END-EXEC

```

In this example, the DECLARE CURSOR statement references a prepared statement (stmt1). A prepared SELECT statement can contain question marks (?) which act as parameter markers to indicate that data is to be supplied when the cursor is opened. The cursor must be declared before the statement is prepared.

Opening a Cursor

Once a cursor has been declared, it must be opened before it can be used. This is done using the OPEN statement, for example:

```

EXEC SQL
  OPEN Cur1
END-EXEC

```

If the DECLARE CURSOR statement references a prepared statement that contains parameter markers, the corresponding OPEN statement must specify the host variables or the name of an SQLDA structure that will supply the values for the parameter markers, for example:

```

EXEC SQL
  OPEN Cur2 USING :last-name
END-EXEC

```

If an SQLDA data structure is used, the data type, length, and address fields must already contain valid data when the OPEN statement is executed.

Using a Cursor to Retrieve Data

Once a cursor has been opened, it can be used to retrieve data from the database. This is done using the FETCH statement. The FETCH statement retrieves the next row from the results set produced by the OPEN statement and writes the data returned to the specified host variables (or to addresses specified in an SQLDA structure). For example:

```
perform until sqlcode not = 0
  EXEC SQL
    FETCH Cur1 INTO :first_name
  END-EXEC
  display 'First name: ' fname
  display 'Last name : ' lname
  display spaces
end-perform
```

When the cursor reaches the end of the results set, a value of 100 is returned in SQLCODE in the SQLCA data structure and SQLSTATE is set to "02000".

As data is fetched from a cursor, locks can be placed on the tables from which the data is being selected.

Closing a Cursor

When your application has finished using the cursor, it should be closed using the CLOSE statement. For example:

```
EXEC SQL
  CLOSE Cur1
END-EXEC
```

Normally, when a cursor is closed, all locks on data and tables are released. If the cursor is closed within a transaction, however, the locks may not be released.

Positioned UPDATE and DELETE Statements

Positioned UPDATE and DELETE statements are used in conjunction with cursors and include WHERE CURRENT OF clauses instead of search condition clauses. The WHERE CURRENT OF clause specifies the corresponding cursor.


```
EXEC SQL
  UPDATE emp SET last_name = :last-name
  WHERE CURRENT OF Cur1
END-EXEC
```

This will update last_name in the row that was last fetched from the database using cursor `Cur1`.

```
EXEC SQL
  DELETE emp WHERE CURRENT OF Cur1
END-EXEC
```

This example will delete the row that was last fetched from the database using cursor `Cur1`.

CitOESQL:

With some ODBC drivers, cursors that will be used for positioned updates and deletes must include a FOR UPDATE clause. Note that positioned UPDATE and DELETE are part of the Extended ODBC Syntax and are not supported by all drivers.

Using Cursors

Cursors are very useful for handling large amounts of data; however, there are a number of issues that you should bear in mind when using cursors, namely: data concurrency, integrity, and consistency.

To ensure the integrity of your data, a database server can implement different locking methods. Some types of data access do not acquire any locks, some acquire a shared lock and some an exclusive lock. A shared lock allows other processes to access the data but not update it. An exclusive lock does not allow any other process to access the data.

When using cursors there are three levels of isolation and these control the data that a cursor can read and lock:

- **Level zero**

Level zero can only be used by read-only cursors. At level zero, the cursor will not lock any rows but may be able to read data that has not yet been committed. Reading uncommitted data is dangerous (as a rollback operation will reset the data to its previous state) and is normally called a "dirty read". Not all databases will allow dirty reads.

- **Level one**

Level one can be used by read-only cursors or updateable cursors. With level one, shared locks are placed on the data unless the FOR UPDATE clause is used. If the FOR UPDATE clause is used, exclusive locks are placed on the data. When the cursor is closed, the locks are released. A

standard cursor, that is a cursor without the FOR UPDATE clause, will normally be at isolation level one and use shared locks.

- **Level three**

Level three cursors are used with transactions. Instead of the locks being released when the cursor is closed, the locks are released when the transaction ends. With level three it is usual to place exclusive locks on the data.

It is worth pointing out that there can be problems with deadlocks or "deadly embraces" where two processes are competing for the same data. The classic example is where one process locks data A and then requests a lock on data B while a second process locks data B and then requests a lock on data A. Both processes have data that the other process requires. The database server should spot this case and send errors to one or both processes.

5.1.4 Data Structures

The CitOESQL preprocessor supplied with this system use two data structures:

Data Structure	Description	Function
SQLCA	SQL Communications Area	Returns status and error information.
SQLDA	SQL Descriptor Area	Describes the variables used in dynamic SQL statements.

SQL Communications Area (SQLCA)

After each embedded SQL statement is executed, error and status information are returned in the SQL Communications Area (SQLCA).

CitOESQL:

The SQLCA provided with COBOL-IT for use with CitOESQL contains two variables (SQLCODE and SQLSTATE), plus a number of warning flags which are used to indicate whether an error has occurred in the most recently executed SQL statement.

Using the SQLCA

The SQLCA structure is supplied in the file sqlca.cpy, which by default is located in the default location specified in the **COBOL-IT CitOESQL files and locations** section in the [CitOESQL Getting Started Guide](#). To include it in your program, use the following statement in the data division:

```
EXEC SQL INCLUDE SQLCA END-EXEC
```

If you do not include this statement, the COBOL Compiler automatically allocates an area, but it is not addressable from within your program. However, if you declare either of the data items `SQLCODE` or `SQLSTATE` separately, the COBOL Compiler generates code to copy the corresponding fields in the `SQLCA` to the user-defined fields after each `EXEC SQL` statement.

If you declare the data item `MFSQLMESSAGETEXT`, it is updated with a description of the exception condition whenever `SQLCODE` is non-zero. `MFSQLMESSAGETEXT` must be declared as a character data item, `PIC X(n)`, where n can be any legal value. This is particularly useful as ODBC error messages often exceed the 70-byte `SQLCA` message field.

Note

You do not need to declare `SQLCA`, `SQLCODE`, `SQLSTATE` or `MFSQLMESSAGETEXT` as host variables.

The `SQLCODE` Variable

Testing the value of `SQLCODE` is the most common way of determining the success or failure of an embedded SQL statement.

For details of `SQLCODE` values, see the relevant database vendor documentation.

The `SQLSTATE` Variable

The `SQLSTATE` variable was introduced in the SQL-92 standard and is the recommended mechanism for future applications. It is divided into two components:

The first two characters are called the class code. Any class code that begins with the letters A through H or the digits 0 through 4 indicates a `SQLSTATE` value that is defined by the SQL standard or another standard.

The last three characters are called the subclass code.

A value of "00000" indicates that the previous embedded SQL statement executed successfully.

For specific details of the values returned in `SQLSTATE`, see the relevant database vendor documentation.

`SQLWARN` Flags

Some statements may cause warnings to be generated. To determine the type of warning, your application should examine the contents of the `SQLWARN` flags.

W - The flag has generated a warning.

blank (space) - The flag has not generated a warning.

The value of a flag is set to **W** if that particular warning occurred, otherwise the value is a blank (space).

Each SQLWARN flag has a specific meaning. For more information on the meaning of the SQLWARN flags, see the relevant database vendor documentation.

The WHENEVER Statement

Explicitly checking the value of SQLCODE or SQLSTATE after each embedded SQL statement can involve writing a lot of code. As an alternative, check the status of the SQL statement by using a WHENEVER statement in your application.

The WHENEVER statement is not an executable statement. It is a directive to the Compiler to automatically generate code that handles errors after each executable embedded SQL statement.

The WHENEVER statement allows one of three default actions (CONTINUE, GOTO or PERFORM) to be registered for each of the following conditions:

Condition	Value of SQLCODE
NOT FOUND	100
SQLWARNING	+1
SQLERROR	\< 0 (negative)

A WHENEVER statement for a particular condition replaces all previous WHENEVER statements for that condition.

The scope of a WHENEVER statement is related to its physical position in the source program, not its logical position in the run sequence. For example, in the following code if the first SELECT statement does not return anything, paragraph A is performed, not paragraph C:

```
EXEC SQL
  WHENEVER NOT FOUND PERFORM A
END-EXEC.
perform B.
EXEC SQL
  SELECT col1 into :host-var1 FROM table1
  WHERE col2 = :host-var2
END-EXEC.
```

- A.
display "First item not found".
- B.
EXEC SQL
 WHENEVER NOT FOUND PERFORM C.
END-EXEC.
- C.
display "Second item not found".

SQLERRM

The SQLERRM data area is used to pass error messages to the application from the database server. The SQLERRM data area is split into two parts:

SQLERRML - holds the length of the error message

SQLERRMC - holds the error text.

Within an error routine, the following code can be used to display the SQL error message:

```
if (SQLERRML \> ZERO) and (SQLERRML \< 80)
  display 'Error Message: ', SQLERRMC(1:SQLERRML)
  else
  display 'Error Message: ', SQLERRMC
end-if.
```

SQLERRD

The SQLERRD data area is an array of six integer status values, set by the database vendor after an SQL error.

```
SQLERRD PIC X9(9) COMP-5 OCCURS 6 VALUE 0.
```

Please consult the relevant database vendor documentation for more detailed information on these values.

The SQL Descriptor Area (SQLDA)

When either the number of parameters to be passed, or their data types, are unknown at compilation time, you can use an SQL Descriptor Area (SQLDA) instead of host variables.

An SQLDA contains descriptive information about each input parameter or output column. It contains the column name, data type, length, and a pointer to the actual data buffer for each input or output parameter. An SQLDA is ordinarily used with parameter markers to specify input values for prepared SQL statements, but you can also use an SQLDA with the DESCRIBE statement (or the INTO option of a PREPARE statement) to receive data from a prepared SELECT statement.

Although you cannot use an SQLDA with static SQL statements, you can use a SQLDA with a cursor FETCH statement.

CitOESQL

The SQLDA structure is supplied in both the `sqllda.cpy` (SQLDA only) and `sqllda78.cpy` (SQLDA plus SQLTYPE definitions) files, which are in the default location specified **COBOL-IT CitOESQL files and locations** section in the [CitOESQL Getting Started Guide](#).

You can include the SQLDA in your COBOL program by adding one or both of the following statements to your data division:

```
EXEC SQL
  INCLUDE SQLDA
END-EXEC

EXEC SQL
  INCLUDE SQLDA78
END-EXEC
```

Using the SQLDA

Before an SQLDA structure is used, your application must initialise the following fields:

SQLN: This must be set to the maximum number of SQLVAR entries that the structure can hold.

The PREPARE and DESCRIBE Statements

You can use the DESCRIBE statement (or the PREPARE statement with the INTO option) to enter the column name, data type, and other data into the appropriate fields of the SQLDA structure.

Before the statement is executed, the SQLN and SQLDABC fields should be initialised as described above.

After the statement has been executed, the SQLD field will contain the number of parameters in the prepared statement. A SQLVAR record is set up for each of the parameters with the SQLTYPE and SQLLEN fields completed.

If you do not know how big the value of SQLN should be, you can issue a DESCRIBE statement with SQLN set to 1 and SQLD set to 0. No column detail information is moved into the SQLDA structure, but the number of columns in the results set is inserted into SQLD.

The FETCH Statement

Before performing a FETCH statement using an SQLDA structure, follow the procedure below:

1. The application must initialize SQLN and SQLDABC as described above.
2. The application must then insert, into the SQLDATA field, the address of each program variable that will receive the data from the corresponding column. (The SQLDATA field is part of SQLVAR).
3. If indicator variables are used, SQLIND must also be set to the corresponding address of the indicator variable.

The data type field (SQLTYPE) and length (SQLLEN) are filled with information from a PREPARE INTO or a DESCRIBE statement. These values can be overwritten by the application prior to a FETCH statement.

The OPEN or EXECUTE Statements

To use an SQLDA structure to specify input data to an OPEN or EXECUTE statement, your application must supply the data for the fields of the entire SQLDA structure, including the SQLN, SQLD, SQLDABC, and SQLTYPE, SQLLEN, and SQLDATA fields for each variable. The following scenarios require additional attention:

- **SQLTYPE field is an odd number**

If the value of the SQLTYPE field is an odd number, you must also supply the address of the indicator variable using SQLIND.

- **Host variable input is COMP**

When using CITOESQL with a host variable input defined as COMP, add **8192** (x2000) to the SQLTYPE field.

- **SQLTYPE field is an odd number and indicator variable is COMP**

If the SQLTYPE field is an odd number, and the indicator variable is defined as a COMP, add **4096** (x1000) to the SQLTYPE field.

- **Host variable input is COMP-5**

When using CITOESQL with a host variable input defined as COMP-5, no change to the SQLTYPE field is required.

The DESCRIBE Statement

After a PREPARE statement, you can execute a DESCRIBE statement to retrieve information about the data type, length and column name of each column returned by the specified prepared statement. This information is returned in the SQL Descriptor Area (SQLDA):

```
EXEC SQL
    DESCRIBE stmt1 INTO :sqllda
END-EXEC
```

If you want to execute a DESCRIBE statement immediately after a PREPARE statement, you can use the INTO option on the PREPARE statement to perform both steps at once:

```
EXEC SQL
    PREPARE stmt1 INTO :sqllda FROM :stmtbuf
END-EXEC
```

The following cases could require that you make manual changes to the SQLTYPE or SQLLEN fields in the SQLDA to accommodate differences in host variable types and lengths after executing DESCRIBE:

- **SQLTYPE: Variable-length character types**

For variable-length character types you can choose to define SQLTYPE as a fixed-size COBOL host variable such as PIC X, N, or G, or a variable-length host variable such as a record with level 49 sub-fields for both length and the actual value. The SQLLEN field could be either 16 or 32 bits depending on the SQLTYPE value.

- **SQLTYPE: Numeric types**

For numeric types you can choose to define SQLTYPE as COMP-3, COMP, COMP-5, or to display numeric COBOL host variables with an included or separate, and leading or trailing sign. The value returned by DESCRIBE depends on the data source. Generally, this is COMP-3 for NUMERIC or DECIMAL columns, and COMP-5 for columns of the tinyint, smallint, integer, or bigint integer types.

- **SQLLEN**

DESCRIBE sets SQLLEN to the size of integer columns in COMP and COMP-5 representations, meaning a value of 1, 2, 4, or 18. You might need to adjust this depending on SQLTYPE. For NUMERIC and DECIMAL columns, it encodes the precision and scale of the result.

5.1.5 Dynamic SQL

If everything is known about an SQL statement when the application is compiled, the statement is known as a static SQL statement.

In some cases, however, the full text of an SQL statement may not be known when an application is written. For example, you may need to allow the end-user of the application to enter an SQL statement. In this case, the statement needs to be constructed at run-time. This is called a dynamic SQL statement.

Dynamic SQL Statement Types

There are four types of dynamic SQL statement:

Dynamic SQL Statement Type	Perform Queries?	Return Data?
Execute a statement once	No	No, can only return success or failure
Execute a statement more than once	No	No, can only return success or failure
Select a given list of data with a given set of selection criteria	Yes	Yes
Select any amount of data with any selection criteria	Yes	Yes

These types of dynamic SQL statement are described more fully in the following sections.

Execute a Statement Once

With this type of dynamic SQL statement, the statement is executed immediately. Each time the statement is executed, it is re-parsed.

Execute a Statement More Than Once

This type of dynamic SQL statement is either a statement that can be executed more than once or a statement that requires host variables. For the second type, the statement must be prepared before it can be executed.

Select a Given List of Data

This type of dynamic SQL statement is a SELECT statement where the number and type of host variables is known. The normal sequence of SQL statements is:

Prepare the statement

Declare a cursor to hold the results

Open the cursor

Fetch the variables

Close the cursor.

Select any Amount of Data

This type of dynamic SQL statement is the most difficult type to code. The type and/or number of variables is only resolved at run time. The normal sequence of SQL statements is:

Prepare the statement

Declare a cursor for the statement

Describe the variables to be used

Open the cursor using the variables just described

Describe the variables to be fetched

Fetch the variables using their descriptions

Close the cursor.

If either the input host variables, or the output host variables are known (at compile time), then the OPEN or FETCH can name the host variables and they do not need to be described.

Preparing Dynamic SQL Statements

The PREPARE statement takes a character string containing a dynamic SQL statement and associates a name with the statement, for example:

```
move "INSERT INTO publishers " &
     "VALUES (?, ?, ?, ?)" to stmtbuf
EXEC SQL
     PREPARE stmt1 FROM :stmtbuf
END-EXEC
```

Dynamic SQL statements can contain parameter markers - question marks (?) that act as a place holder for a value. In the example above, the values to be substituted for the question marks must be supplied when the statement is executed.

Once you have prepared a statement, you can use it in one of two ways:

You can execute a prepared statement.

You can open a cursor that references a prepared statement.

Executing Dynamic SQL Statements

The EXECUTE statement runs a specified prepared SQL statement.

Note

Only statements that do not return results can be executed in this way.

If the prepared statement contains parameter markers, the EXECUTE statement must include either the "using :hvar" option to supply parameter values using host variables or the "using descriptor :sqllda_struct" option identifying an SQLDA data structure already populated by the application. The number of parameter markers in the prepared statement must match the number of SQLDATA entries ("using descriptor :sqllda") or host variables ("using :hvar").

```
move "INSERT INTO publishers " &
      "VALUES (?, ?, ?, ?)" to stmtbuf
EXEC SQL
      PREPARE stmt1 FROM :stmtbuf
END-EXEC
...
EXEC SQL
      EXECUTE stmt1 USING :pubid, :pubname, :city, :state
END-EXEC.
```

In this example, the four parameter markers are replaced by the contents of the host variables supplied via the USING clause in the EXECUTE statement.

EXECUTE IMMEDIATE Statement

If the dynamic SQL statement does not contain any parameter markers, you can use EXECUTE IMMEDIATE instead of PREPARE followed by EXECUTE, for example:

```
move "DELETE FROM emp " &
      "WHERE last_name = 'Smith'" to stmtbuf
EXEC SQL
      EXECUTE IMMEDIATE :stmtbuf
END-EXEC
```

When using EXECUTE IMMEDIATE, the statement is re-parsed each time it is executed. If a statement is likely to be used many times it is better to PREPARE the statement and then EXECUTE it when required.

Dynamic SQL Statements and Cursors

If a dynamic SQL statement returns a result, you cannot use the EXECUTE statement. Instead, you must declare and use a cursor.

First, declare the cursor using the DECLARE CURSOR statement:

```
EXEC SQL
    DECLARE C1 CURSOR FOR dynamic_sql
END-EXEC
```

In the example above, `dynamic_sql` is the name of a dynamic SQL statement. You must use the PREPARE statement to prepare the dynamic SQL statement before the cursor can be opened, for example:

```
move "SELECT char_col FROM mfesqltest " &
    "WHERE int_col = ?" to sql-text
EXEC SQL
    PREPARE dynamic_sql FROM :sql-text
END-EXEC
```

Now, when the OPEN statement is used to open the cursor, the prepared statement is executed:

```
EXEC SQL
    OPEN C1 USING :int-col
END-EXEC
```

If the prepared statement uses parameter markers, then the OPEN statement must supply values for those parameters by specifying either host variables or an SQLDA structure.

Once the cursor has been opened, the FETCH statement can be used to retrieve data, for example:

```
EXEC SQL
    FETCH C1 INTO :char-col
END-EXEC
```

Finally, the cursor is closed using the CLOSE statement:

```
EXEC SQL
    CLOSE C1
END-EXEC
```

CALL Statements

A CALL statement can be prepared and executed as dynamic SQL.

You can use parameter markers (?) in dynamic SQL wherever you use host variables in static SQL

Use of the IN, INPUT, OUT, OUTPUT, INOUT and CURSOR keyword following parameter markers is the same as their use after host variable parameters in static SQL.

- The whole call statement must be enclosed in braces to conform to ODBC canonical stored procedure syntax (the CitOESQL precompiler does this for you in static SQL). For example:

```
move '{call myproc(?, ? out)}' to sql-text
exec sql
    prepare mycall from :sql-text
end-exec
exec sql
    execute mycall using :parm1, :parm2
end-exec
```

- If you use parameter arrays, you can limit the number of elements used with a FOR clause on the EXECUTE, for example:

```
move 5 to param-count
exec sql
    for :param-count
        execute mycall using :parm1, :param2
end-exec
```

EXAMPLE:

The following is an example of a program that creates a stored procedure "mfexecspstest" using data source "SQLServer 2000" and then retrieves data from "publishers" table using a cursor "c1" with dynamic SQL.

```

\$$SET SQL
    WORKING-STORAGE SECTION.

EXEC SQL INCLUDE SQLCA END-EXEC

\*\> after an sql error this has the full message text
01 MFSQLMESSAGETEXT    PIC X(250).
01 IDX                  PIC X(04) COMP-5.

EXEC SQL BEGIN DECLARE SECTION END-EXEC
\*\> Put your host variables here if you need to port
| \*\> to other COBOL compilers

01 stateParam          pic xx.
01 pubid                pic x(4).
01 pubname              pic x(40).
01 pubcity              pic x(20).

01 sql-stat            pic x(256).

EXEC SQL END DECLARE SECTION END-EXEC

PROCEDURE DIVISION.

    EXEC SQL
        WHENEVER SQLERROR perform OpenESQL-Error
    END-EXEC

    EXEC SQL
        CONNECT TO 'SQLServer 2000' USER 'SA'
    END-EXEC

\*\> Put your program logic/SQL statements here

    EXEC SQL
        create procedure mfexecsptest
            (@stateParam char(2) = 'NY' ) as

            select pub_id, pub_name, city from publishers
            where state = @stateParam
    END-EXEC

    exec sql
        declare c1 scroll cursor for dsq12 for read only
    end-exec

```

```

move "{call mfexecspstest(?)}" to sql-stat
exec sql prepare dsq12 from :sql-stat end-exec

move "CA" to stateParam
exec sql
    open c1 using :stateParam
end-exec

display "Testing cursor with stored procedure"
perform until exit
    exec sql
        fetch c1 into :pubid, :pubname, :pubcity
    end-exec

    if sqlcode = 100
        exec sql close c1 end-exec
        exit perform
    else
        display pubid " " pubname " " pubcity
    end-if
end-perform

EXEC SQL close c1 END-EXEC

EXEC SQL DISCONNECT CURRENT END-EXEC
EXIT PROGRAM.
STOP RUN.
*> Default sql error routine / modify to stop program if
*> needed
OpenESQL-Error Section.

display "SQL Error = " sqlstate " " sqlcode
display MFSQLEMSGTEXT
*> stop run
exit.

```

5.2 SQL Statements

With the exception of INSERT, DELETE(SEARCHED) and UPDATE(SEARCHED), which are included for your convenience, the embedded SQL statements described here work somewhat differently, or are in addition to, standard SQL statements.

SQL Statement	Description
BEGIN DECLARE SECTION	Signals the beginning of the DECLARE section.
BEGIN TRAN	Provides compatibility with Embedded SQL implementations that do not conform to the ANSI SQL standard with respect to transaction management and, in particular, the Micro Focus Embedded SQL Toolkit for Microsoft SQL Server.
CALL	Executes a stored procedure.
CLOSE	Discards unprocessed rows and frees any locks held by the cursor.
COMMIT	Makes any changes made by the current transaction on the current connection permanent in the database.
CONNECT	Attaches to a specific database using the supplied username and password.
DECLARE CURSOR	Associates the cursor name with the specified SELECT statement and enables you to retrieve rows of data using the FETCH statement.
DECLARE DATABASE	Declares the name of a database.
DELETE (Positioned)	Deletes the row most recently fetched by using a cursor.
DELETE (Searched)	Removes table rows that meet the search criteria.
DESCRIBE	Provides information on prepared dynamic SQL statements and describes the result set for an open cursor.
DISCONNECT	Closes the connection(s) to a database. In addition, all cursors opened for that connection are automatically closed.
END DECLARE SECTION	Terminates a host variable declaration section begun by a BEGIN DECLARE SECTION statement.
EXECSP	Executes a stored procedure.

SQL Statement	Description
EXECUTE	Processes dynamic SQL statements.
EXECUTE IMMEDIATE	Immediately executes the SQL statement.
FETCH	Retrieves a row from the cursor's results set and writes the values of the columns in that row to the corresponding host variables (or to addresses specified in the SQLDA data structure).
GET HDBC	Enables you to use ODBC calls that require you to supply the ODBC connection handle.
GET HENV	Enables you to use ODBC calls that require you to supply the ODBC environment handle.
GET NEXT RESULT SET	Makes the next result set available to an open cursor.
INCLUDE	Includes the definition of the specified SQL data structure or source file in the COBOL program.
INSERT	Adds new rows to a table.
INTO	Retrieves one row of results and assigns the values of the items returned by an OUTPUT clause in a SQL Server INSERT, UPDATE, or DELETE statement to the host variables specified in the INTO list.
OPEN	Runs the SELECT statement specified in the corresponding DECLARE CURSOR statement to produce the results set that is accessed one row at a time by the FETCH statement.
PREPARE	Processes dynamic SQL statements.
QUERY ODBC	Delivers a results set in the same way as a SELECT statement, and must therefore be associated with a cursor via DECLARE and OPEN, or DECLARE, PREPARE and OPEN.
RESET CONNECTION	Closes all open cursors, even if the application has not appropriately closed them.
ROLLBACK	Backs out any changes made to the database by the current transaction on the current connection, or partially rolls back changes to a previously set save point.

SQL Statement	Description
SAVEPOINT SAVE TRANSACTION RELEASE [TO] SAVEPOINT	Sets a transaction save point to which a current transaction can be rolled back, resulting in a partial roll back.
SELECT DISTINCT (using DECLARE CURSOR)	Associates the cursor name with the SELECT DISTINCT statement and enables you to retrieve rows of data using the FETCH statement.
SELECT INTO	Retrieves one row of results and assigns the values of the items in a specified SELECT list to the host variables specified in the INTO list.
SET AUTOCOMMIT	Enables you to control ODBC AUTOCOMMIT mode at run time.
SET CONNECTION	Sets the named connection as the current connection.
SET ERRORMAP	Changes the SQL error map file for the current connection.
SET host_variable	Provides information about CitOESQL connections and databases.
SET OPTION	Enables you to set CitOESQL options.
SET TRACELEVEL	Enables you to dynamically set or change the reporting level of CitOESQL traces for native applications.
SET TRANSACTION ISOLATION	Sets the transaction isolation level for the current connection to one of the isolation level modes specified by ODBC.
SYNCPOINT	Closes all open cursors that were not opened using the WITH HOLD clause, even if the application has not appropriately closed them.
UPDATE (Positioned)	Updates the rows most recently fetched by using a cursor.
UPDATE (Searched)	Updates a table or view based on specified search conditions.

SQL Statement	Description
WHENEVER	Specifies the default action after running an Embedded SQL statement when a specific condition is met.

5.2.1 BEGIN DECLARE SECTION

Signals the beginning of the DECLARE section.

Syntax:

```
\>\>---EXEC SQL---BEGIN DECLARE SECTION---END-EXEC---\>\<
```

Comments:

The BEGIN DECLARE SECTION statement can be included anywhere where COBOL permits variable declaration. Use END DECLARE SECTION to identify the end of a COBOL declaration section.

Declare sections cannot be nested.

Variables must be declared in COBOL, not in SQL.

To avoid conflict, variables inside a declaration section cannot be the same as any outside the declaration section or in any other declaration section, even in other compilation units.

If data structures are defined within a declaration section, only the bottom-level items (with PIC clauses) can be used as host variables. Two exceptions are arrays specified in FETCH statements and record structures specified in SELECT INTO statements.

Example:

```
WORKING-STORAGE SECTION.
  EXEC SQL BEGIN DECLARE SECTION END-EXEC
  01 staff-id pic x(4).
  01 last-name pic x(30).
  EXEC SQL END DECLARE SECTION END-EXEC
```

5.2.2 BEGIN TRAN

Provides compatibility with Embedded SQL implementations that do not conform to the ANSI SQL standard with respect to transaction management and the Micro Focus Embedded SQL Toolkit for Microsoft SQL Server.

Syntax Format 1:

```
\>\>--EXEC SQL--BEGIN TRAN-.-----.-END-EXEC---\>\<
    \+-transaction_name-+
```

Syntax Format 2:

```
\>\>-EXEC SQL-BEGIN TRANSACTION.-----.-END-EXEC-\>\<
    \+transaction_name+
```

Parameters:

Transaction name

An optional identifier that is ignored.

Comments:

Use the BEGIN TRAN statement in AUTOCOMMIT mode to open a transaction. After you have opened the transaction in AUTOCOMMIT mode, you should execute a COMMIT or ROLLBACK statement to close the transaction and cause a return to AUTOCOMMIT mode.

If you are not opening a transaction in AUTOCOMMIT mode, then this statement has no effect.

Example:

```
EXEC SQL BEGIN TRANSACTION END-EXEC
```

5.2.3 CALL

Executes a stored procedure.

Syntax:

```

>>--EXEC SQL--.-----.->
      +-FOR :host_integer--+ +- :result_hvar -+

>---CALL stored_procedure_name-.-END-EXEC-><
      | +-- , --+ |
      | V | |
      +(parameter)-+

```

Parameters:

Host Integer

A host variable that specifies the maximum number of host array elements processed. Must be declared as PIC S9(4) COMP-5 or PIC S9(9) COMP-5.

result_hvar

A host variable to receive the procedure result.

stored_procedure_name

The name of the stored procedure.

parameter

A literal, a DECLARE CURSOR statement ^{*}, or a host variable parameter of the form:

```
[keyword=]:param_hvar [IN | INPUT |
```

```
INOUT | OUT | OUTPUT]
```

where:

keyword The formal parameter name for a keyword parameter. Keyword parameters can be useful as an aid to readability and where the server supports default parameter values and optional parameters.

param_hvar A host variable.

IN An input parameter.

INPUT An input parameter default).

INOUT An input/output parameter.

OUT An output parameter.

OUTPUT An output parameter.

* Specify DECLARE CURSOR for stored procedures that return a result set. The use of DECLARE CURSOR unbinds the corresponding parameter.

Comments:

Do not use the FOR clause if the CALL is part of a DECLARE CURSOR statement.

For maximum portability, observe the following as general rules:

Avoid literal parameters

Use host variable parameters

Avoid mixing positional parameters and keyword parameters

If your server supports a mixture of positional and keyword parameters, list keyword parameters after positional parameters

Examples:

Call a stored procedure using two positional host variables as input parameters:

```
EXEC SQL
    CALL myProc(param1,param2)
END-EXEC
```

Call a stored procedure using a keyword host variable as an input parameter:

```
EXEC SQL
    CALL myProc (namedParam=:paramValue)
END-EXEC
```

Call a stored procedure using a result host variable and a keyword host variable as an input parameter:

```
EXEC SQL
    :myResult = CALL myFunction(namedParam=:paramValue)
END-EXEC
```

Call a stored procedure using two positional host variables, one as an input parameter and one as an output parameter:

```
EXEC SQL
  CALL getDept(:empName IN, :deptName OUT)
END-EXEC
```

Call a stored procedure using a DECLARE CURSOR statement and a positional host variable as an input parameter (Oracle only):

```
EXEC SQL
  DECLARE cities CURSOR FOR CALL locateStores(:userState)
END-EXEC
```

5.2.4 CLOSE

Discards unprocessed rows and frees any locks held by the cursor.

Syntax:

```
\>\>---EXEC SQL---.-----.----\>
      \+-AT db_name-+

\>--CLOSE---cursor_name---.-----.----END-EXEC---\>\<
```

Parameters:

AT db_name

The name of a database that has been declared using DECLARE DATABASE. This clause is not required, and if omitted, the connection automatically switches to the connection associated with the DECLARE CURSOR statement if different than the current connection, but only for the duration of the statement. Provided for backward compatibility.

cursor_name

A previously declared and opened cursor.

Comments:

The cursor must be declared and opened before it can be closed. All open cursors are closed automatically at the end of the program.

Example:

```

*Declare the cursor...
EXEC SQL
    DECLARE C1 CURSOR FOR
        SELECT staff_id, last_name
        FROM staff
END-EXEC

IF SQLCODE NOT = ZERO
    DISPLAY 'Error: Could not declare cursor.'
    DISPLAY SQLERRMC
    DISPLAY SQLERRML
    EXEC SQL DISCONNECT ALL END-EXEC
    STOP RUN
END-IF

EXEC SQL
    OPEN C1
END-EXEC

IF SQLCODE NOT = ZERO
    DISPLAY 'Error: Could not open cursor.'
    DISPLAY SQLERRMC
    DISPLAY SQLERRML
    EXEC SQL DISCONNECT CURRENT END-EXEC
    STOP RUN
END-IF

PERFORM UNTIL sqlcode NOT = ZERO
*SQLCODE will be zero as long as it has successfully fetched data
EXEC SQL
    FETCH C1 INTO :staff-staff-id, :staff-last-name
END-EXEC
IF SQLCODE = ZERO
    DISPLAY "Staff ID: " staff-staff-id
    DISPLAY "Staff member's last name: " staff-last-name
END-IF
END-PERFORM

EXEC SQL
    CLOSE C1
END-EXEC

IF SQLCODE NOT = ZERO
    DISPLAY 'Error: Could not close cursor.'
    DISPLAY SQLERRMC

```



```

    DISPLAY SQLERRML
END-IF

```

5.2.5 COMMIT

Makes any changes made by the current transaction on the current connection permanent in the database.

Syntax:

```

>>---EXEC SQL-- .----- .--->
                +-AT db_name-+

>---COMMIT----- .----- .--->
                +-WORK-----+
                +-TRAN-----+
                +-TRANSACTION--+

>--- .----- .---END-EXEC--><
                +--RELEASE--+

```

Parameters:

AT *db_name*

The name of a database that has been declared using DECLARE DATABASE. This clause is optional. If omitted, the current connection is committed. If provided, and the connection specified is different than the current connection, the commit is performed on the connection associated with the DECLARE CURSOR statement.

WORK

WORK, TRAN, and TRANSACTION are optional and synonymous.

RELEASE

If RELEASE is specified and the transaction was successfully committed, the current connection is closed.

Example:

```
* Ensure that multiple records are not inserted for a
* member of staff whose staff_id is 99
  EXEC SQL
    DELETE FROM staff WHERE staff_id = 99
  END-EXEC

* Insert dummy values into table
  EXEC SQL
    INSERT INTO staff
      (staff_id
      ,last_name
      ,first_name
      ,age
      ,employment_date)
    VALUES
      (99
      , 'Lee'
      , 'Phil'
      ,19
      , '1992-01-02')
  END-EXEC

  IF SQLCODE NOT = ZERO
    DISPLAY 'Error: Could not insert dummy values.'
    DISPLAY SQLERRMC
    DISPLAY SQLERRML
    EXEC SQL DISCONNECT ALL END-EXEC
    STOP RUN
  END-IF

  EXEC SQL
    COMMIT
  END-EXEC

* Check it was committed OK
  IF SQLCODE = ZERO
    DISPLAY 'Error: Could not commit values.'
    DISPLAY SQLERRMC
    DISPLAY SQLERRML
    EXEC SQL DISCONNECT CURRENT END-EXEC
    STOP RUN
  END-IF

  DISPLAY 'Values committed.'

* Delete previously inserted data
```

```

EXEC SQL
  DELETE FROM staff WHERE staff_id = 99
END-EXEC

IF SQLCODE NOT = ZERO
  DISPLAY 'Error: Could not delete dummy values.'
  DISPLAY SQLERRMC
  DISPLAY SQLERRML
  EXEC SQL DISCONNECT ALL END-EXEC
  STOP RUN
END-IF

```

* Check data deleted OK, commit and release the connection

```

IF SQLCODE NOT = ZERO
  DISPLAY 'Error: Could not delete values.'
  DISPLAY SQLERRMC
  DISPLAY SQLERRML
  EXEC SQL DISCONNECT ALL END-EXEC
  STOP RUN
END-IF

```

```

EXEC SQL
  COMMIT WORK RELEASE
END-EXEC

```

* Check data committed OK and release the connection.

```

IF SQLCODE NOT = ZERO
  DISPLAY 'Error: Could not commit and release.'
  DISPLAY SQLERRMC
  DISPLAY SQLERRML
  EXEC SQL DISCONNECT CURRENT END-EXEC
END-IF

```

```

DISPLAY 'Values committed and connection released.'

```

5.2.6 CONNECT

Attaches to a specific database using the supplied user name and password.

Syntax Format 1:

```

>>---EXEC SQL---CONNECT TO---.----->
                                +-data_source-+

>---.-----USER-.->
+-AS db_name-+      +-user-.-++
                                +-password-+

>---.----->
+-WITH-.-PROMPT-+ +-RETURNING output_connection-+
+-NO-+

>-----END-EXEC-----<<

```

Syntax Format 2:

```

>>---EXEC SQL---CONNECT user--.->
                                +-IDENTIFIED BY password-+
                                +-----'/'password-----+

>---.----->
+-AT db_name--+      +-USING data_source-+

>---.----->
+-WITH-.-PROMPT--+
+-NO-+

>---.-----END-EXEC---<<
+-RETURNING output_connection-+

```

Syntax Format 3:

```

>>-----EXEC SQL---CONNECT WITH PROMPT----->

>---.-----END-EXEC-----<<
+-RETURNING output_connection -+

```

Syntax Format 4:

```

>>-----EXEC SQL---CONNECT RESET-.-END-EXEC-----<<
                                +-name--+

```

Syntax Format 5:

```
>>----EXEC SQL-----CONNECT DSN input_connection----->
>-----END-EXEC-----<
+-RETURNING output_connection -+
```

Syntax Format 6:

```
>>----EXEC SQL---CONNECT USING input_connection----->
>-----END-EXEC-----<
+-AS db_name-+ +-WITH- .-----.-PROMPT-+
+-NO-+
>-----END-EXEC-----<
+-RETURNING output_connection-+
```

Parameters:***data_source***

The name of the ODBC data store. For ODBC data stores, this is the DSN created via the Microsoft ODBC Data Source Administrator. If you omit *data_source*, the default ODBC data source is assumed. The data source can be specified as a literal or as a host variable.

db_name

A name for the connection. Connection names can have as many as 30 characters and can include alphanumeric characters and any symbols legal in filenames. The first character must be a letter. Do not use Embedded SQL keywords or CURRENT or DEFAULT or ALL

for the connection name; they are invalid. If *db_name* is omitted, DEFAULT is assumed. *db_name* can be specified as a literal or a host variable. When connecting to SQL Server, *db_name* is the database to which you are connecting.

user

A valid user-id at the specified data source.

password

A valid password for the specified user-id.

output_connection

A PIC X(n) text string defined by ODBC as the connection string used to connect to a particular data source. Subsequently, you can specify this string as the *input_connection* in a CONNECT USING statement.

input_connection

A PIC X(n) text string containing connection information used by ODBC to connect to the data source. The test string can be either a literal or a host variable.

RESET

Resets (disconnects) the specified connection.

name

You can specify *name* as CURRENT, DEFAULT or ALL.

OS Authentication:

When using Oracle, DB2 or SQL Server with ODBC, you can achieve OS authentication using either of these two methods:

In the CONNECT statement, specify a user ID consisting of a single forward slash and either omit the password or specify all spaces

Completely omit the user ID and password from the CONNECT statement

For complete information on OS authentication requirements for your DBMS product, consult your DBMS documentation.

Comments:

If you use only one connection, you do not need to supply a name for the connection. When you use more than one connection, you must specify a name for each connection. Connection names are global within a process. Named connections are shared by separately compiled programs that are linked into a single executable module.

After a successful CONNECT statement, all database transactions other than CONNECT RESET work through this most recently declared current connection. To use a different connection, use the SET CONNECTION statement.

To cause the ODBC run-time module to prompt at run-time for entry or confirmation of connection details, use CONNECT WITH PROMPT.

Use CONNECT DSN and CONNECT USING to simplify administration.

With CONNECT TO, CONNECT, CONNECT WITH PROMPT, CONNECT DSN and CONNECT USING, you can return connection information to the application.

Note

If the INIT option of the SQL Compiler directive is used, an implicit connection to the database will be made at run time. In this case, it is not necessary to execute an explicit CONNECT statement.

A File DSN cannot contain a password.

Example Format 1:

```
MOVE 'servername' TO svr
MOVE 'username.password' TO usr

EXEC SQL
    CONNECT TO :svr USER :usr
END-EXEC
```

Example Format 2:

```
EXEC SQL
    CONNECT 'username.password' USING 'servername'
END-EXEC
```

Example Format 3:

```
EXEC SQL
  CONNECT WITH PROMPT
END-EXEC
```

Example Format 4:

```
EXEC SQL
  CONNECT RESET
END-EXEC
```

Example Format 5:

```
EXEC SQL
  CONNECT USING 'FileDSN=Oracle8;PWD=tiger'
END-EXEC
```

The example above uses a File DSN.

Example Format 6:

```
01 connectString          PIC X(72) value
                          'DRIVER={Microsoft Excel Driver (*.xls)};'
                          &'DBQ=c:\demo\demo.xls;'
                          &'DRIVERID=22'
                          .
procedure division.

  EXEC SQL
    CONNECT USING :connectString
  END-EXEC
```

The example above connects to an Excel spreadsheet without setting up a data source.

5.2.7 DECLARE CURSOR

Associates the cursor name with the specified SELECT statement and enables you to retrieve rows of data using the FETCH statement. **Syntax Format 1:**

```
>>--EXEC SQL-----,-----DECLARE cursor_name----->
      +-AT db_name-+
>-----,-----,-----,-----,----->
+-SENSITIVE---+    +-FORWARD--+    +-LOCK-----+
+-INSENSITIVE-+    +-KEYSET--+    +-LOCKCC-----+
                  +-DYNAMIC--+    +-OPTIMISTIC---+
                  +-STATIC--+    +-OPTCC-----+
                  +-SCROLL--+    +-OPTCCVAL-----+
                        +-READ ONLY----+
                                  +-READONLY-----+
                                  +-FASTFORWARD--+
                                  +-FAST FORWARD-+

>--CURSOR-----,-----FOR----->
      +----WITH HOLD---+

>----select_stmt-----,----->
      +----stored_procedure_call_statement---+
      +----prepared_stmt_name-----+
      +----OPTIMIZE FOR n ROWS-----+
>-----,-----,----->
      +-FOR READ ONLY-----+
      +-FOR UPDATE-.-----,--+
                +-OF column_list-+
>-----END-EXEC-----<
```

Syntax Format 2:

Note


Format 2 is supported for SQL Server only.

```
>>--EXEC SQL---.-----DECLARE cursor_name----->
      +-AT db_name-+
>--CURSOR FOR---result-set-generating-dml-statement----->

>-----END-EXEC-----<<
```

Parameters:***AT db_name***

The name of a database that has been declared using DECLARE DATABASE.

 Note

If you must use *AT db_name* in a DECLARE CURSOR, the connection for any following statements that reference the cursor automatically switch to the connection associated with the cursor if different than the current connection, but only for the duration of the statement.

cursor_name

Cursor name used to identify the cursor in subsequent statements. Cursor names can contain any legal filename character and be up to 30 characters in length. The first character must be a letter.

select_stmt

Any valid SQL SELECT statement, or a QUERY ODBC statement or a CALL statement for a stored procedure that returns a result set.

prepared_stmt_name

The name of a prepared SQL SELECT statement or QUERY ODBC statement.

stored_procedure_call_stmt

A valid stored procedure call which returns a result set.

n

The number of rows per block fetched when the cursor is opened. The value of *n* must be less than 1000.

column_list

A list of column-names, separated by commas.

result-set-generating-dmlstatement

A SQL Server INSERT, non-positioned UPDATE, or DELETE statement with an OUTPUT clause.

Comments:

Two separately compiled programs cannot share the same cursor. All statements that reference a particular cursor must be compiled together.

The DECLARE CURSOR statement must appear before the first reference to the cursor. The SELECT statement runs when the cursor is opened. The following rules apply to the SELECT statement:

It cannot contain an INTO clause or parameter markers.

It can contain input host variables previously identified in a declaration section.

With some ODBC drivers, the SELECT statement must include a FOR UPDATE clause if positioned updates or deletions are to be performed.

If OPTIMIZE FOR is specified, OpenESQL uses *n* to override the setting of the PREFETCH directive for the cursor. This allows prefetch optimization for individual cursors.

You can specify multiple SELECT statements in a DECLARE CURSOR statement, signifying the return of multiple result sets from either of the following. In either case, the client application must use the GET NEXT RESULT SET statement to retrieve additional result sets.

- A COBOL stored procedure for SQL Server

- A standard OpenESQL application program

The following applies to the behavior of certain DECLARE CURSOR options:

SCROLL selects a scroll option, other than FORWARD, that is supported by the driver.

LOCKCC and LOCK are equivalent.

READ ONLY and READONLY are equivalent.

OPTIMISTIC selects an optimistic concurrency mode (OPTCC or OPTCCVAL) that is supported by the driver.

If a HOLD cursor is requested and the current connection closes cursors at the end of transactions, the OPEN statement will return an error (SQLCODE = -19520).

If the database is Microsoft SQL Server and the NOANSI92 ENTRY directive setting has been used (this is the default setting), then a Microsoft SQL Server specific ODBC call will be made at connect time to request that cursors are not closed at the end of transactions. This is compatible with the Micro Focus Embedded SQL Toolkit for Microsoft SQL Server. The setting for USECURLIB must not be YES.

FAST FORWARD and FASTFORWARD are equivalent. This is a performance optimization parameter that applies only to FORWARD, READ-ONLY cursors. You can obtain even greater performance gains by also compiling the program with the AUTOFETCH directive; this is the most efficient method of getting a results set into an application. The AUTOFETCH directive enables two optimizations that can significantly reduce network traffic. The most dramatic improvement is seen when processing cursors with relatively small result sets that can be cached in the memory of an application.

FASTFORWARD cursors work only with Microsoft SQL Server 2000 or later servers.

Example:

```
EXEC SQL DECLARE C1 CURSOR FOR
    ELECT last_name, first_name FROM staff
END-EXEC
```

```
EXEC SQL DECLARE C2 CURSOR FOR
    QUERY ODBC COLUMNS TABLENAME 'staff'
END-EXEC
```

5.2.8 DECLARE DATABASE

Declares the name of a database.

Syntax:

```
>>---EXEC SQL---DECLARE db_name---DATABASE---END-EXEC---->
```

Parameters:

db_name

A name associated with a database. It must be an identifier and not a host variable. It cannot contain quotation marks.

Comments:

You must DECLARE *db_name* before using a CONNECT ... AT *db_name* statement. You cannot use DECLARE DATABASE with EXECUTE IMMEDIATE or with PREPARE and EXECUTE.

5.2.9 DELETE (Positioned)

Deletes the row most recently fetched by using a cursor.

Syntax:

```
>>---EXEC SQL---.-----.->
      +-AT db_name-+
>--DELETE---FROM---table_name--->
>--WHERE CURRENT OF--cursor_name---END-EXEC---<
```

Parameters:

AT db_name

The name of a database that has been declared using DECLARE DATABASE. This clause is not required, and if omitted, the connection automatically switches to the connection associated with the DECLARE CURSOR statement if different than the current connection, but only for the duration of the statement.

table_name

The same table used in the SELECT FROM option (see DECLARE CURSOR).

cursor_name

A previously declared, opened, and fetched cursor.

Comments:

ODBC supports positioned delete, which deletes the row most recently fetched by using a cursor in the Extended Syntax (it was in the Core Syntax for ODBC 1.0 but was moved to the Extended Syntax for ODBC 2.0). Not all drivers provide support for positioned delete, although OpenESQL sets ODBC cursor names to be the same as COBOL cursor names to facilitate positioned update and delete.

With some ODBC drivers, the select statement used by the cursor must contain a 'FOR UPDATE' clause to enable positioned delete.

You cannot use host arrays with positioned delete.

The other form of DELETE used in standard SQL statements is known as a searched delete.

Most data sources require specific combinations of SCROLLOPTION and CONCURRENCY to be specified either by SET statements or in the DECLARE CURSOR statement.

The ODBC cursor library provides a restricted implementation of positioned delete which can be enabled by compiling with SQL(USECURLIB=YES) and using SCROLLOPTION STATIC and CONCURRENCY OPTCCVAL (or OPTIMISTIC).

Example:

```

* Declare a cursor for update
  EXEC SQL DECLARE C1 CURSOR FOR
    SELECT staff_id, last_name FROM staff FOR UPDATE
  END-EXEC

  IF SQLCODE NOT = ZERO
    DISPLAY 'Error: Could not declare cursor for update.'
    DISPLAY SQLERRMC
    DISPLAY SQLERRML
    EXEC SQL DISCONNECT ALL END-EXEC
    STOP RUN
  END-IF

* Open the cursor
  EXEC SQL
    OPEN C1
  END-EXEC
  IF SQLCODE NOT = ZERO
    DISPLAY 'Error: Could not open cursor for update.'
    DISPLAY SQLERRMC
    DISPLAY SQLERRML
    EXEC SQL DISCONNECT ALL END-EXEC
    STOP RUN
  END-IF

* Display staff member's details and give user the opportunity
* to delete particular members.
  PERFORM UNTIL SQLCODE NOT = ZERO
    EXEC SQL FETCH C1 INTO :staff-id, :last-name END-EXEC
    IF SQLCODE = ZERO
      DISPLAY 'Staff ID: ' staff-id
      DISPLAY 'Last name: ' last-name
      DISPLAY 'Delete <y/n>? ' WITH NO ADVANCING
      ACCEPT usr-input
      IF usr-input = 'y'
        EXEC SQL
          DELETE FROM staff WHERE CURRENT OF C1
        END-EXEC
      IF SQLCODE NOT = ZERO
        DISPLAY 'Error: Could not delete record.'
        DISPLAY SQLERRMC
        DISPLAY SQLERRML
      END-IF
    END-IF
  END-IF
  END-IF
  END-PERFORM

```