Oracle® SQL*Module for Ada

Programmer's Guide

11*g* Release 1 (11.1)

B31228-01

July 2007

ORACLE®

Oracle SQL*Module for Ada Programmer's Guide, 11*g* Release 1 (11.1)

B31228-01

Primary Author: Deepti Kamal

Contributing Author: Shiladitya Guha, Simon Watt

Contributor: Subhranshu Banerjee

Contents

3   Accessing Stored Procedures

4   Developing the Ada Application

5 Running SQL*Module

6   Demonstration Programs

A   New Features

B   Module Language Syntax

C   Reserved Words

D   SQLSTATE Codes

E   System-Specific References

Index

Preface

This book is a comprehensive user's guide and reference for SQL*Module, an Oracle application development tool.

This Guide includes a complete description of *Module Language*, an ANSI/ISO SQL standard for developing applications that access data stored in a relational database. Module Language uses parameterized procedures to encapsulate SQL statements. The procedures can then be called from an Ada application.

This Guide also describes how you can use SQL*Module to call PL/SQL procedures stored in an Oracle database. A number of complete examples using Module Language, Ada code, and stored database procedures are provided.

Intended Audience

Oracle SQL*Module for Ada Programmer's Guide is intended for systems architects, analysts, and developers who are writing large-scale applications that access an Oracle Server. Chapter 1 of this Guide can also be used by managers who need to determine if SQL*Module is an appropriate tool for a planned project.

To use this Guide effectively, you need a working knowledge of the following topics:

- applications programming in Ada
- the SQL database language
- Oracle database concepts and terminology

Familiarity with SQL-standard Module Language is not a prerequisite. This Guide fully documents Module Language.

Documentation Accessibility

Our goal is to make Oracle products, services, and supporting documentation accessible, with good usability, to the disabled community. To that end, our documentation includes features that make information available to users of assistive technology. This documentation is available in HTML format, and contains markup to facilitate access by the disabled community. Accessibility standards will continue to evolve over time, and Oracle is actively engaged with other market-leading technology vendors to address technical obstacles so that our documentation can be accessible to all of our customers. For more information, visit the Oracle Accessibility Program Web site at

http://www.oracle.com/accessibility/

Accessibility of Code Examples in Documentation

Screen readers may not always correctly read the code examples in this document. The conventions for writing code require that closing braces should appear on an otherwise empty line; however, some screen readers may not always read a line of text that consists solely of a bracket or brace.

Accessibility of Links to External Web Sites in Documentation

This documentation may contain links to Web sites of other companies or organizations that Oracle does not own or control. Oracle neither evaluates nor makes any representations regarding the accessibility of these Web sites.

TTY Access to Oracle Support Services

Oracle provides dedicated Text Telephone (TTY) access to Oracle Support Services within the United States of America 24 hours a day, seven days a week. For TTY support, call 800.446.2398.

Standards Conformance

SQL*Module conforms to the American National Standards Institute (ANSI) and International Standards Organization (ISO) standards for Module Language. This includes complete conformance with Chapter 7 of ANSI document X3.135-1989, *Database Language SQL with Integrity Enhancement*.

In addition, SQL*Module conforms to the "Entry SQL" subset of the SQL92 standard, as defined in Chapter 12 of the ANSI Document X3.135-1992.

Note: SQL92 is known officially as International Standard ISO/IEC 9075:1992, Database Language SQL.

SQL*Module supports the Ada83 language standard for Ada.

Oracle has also implemented extensions to the SQL language and to Module Language. This Guide describes both the SQL standard Module Language and the complete set of Oracle extensions. SQL*Module provides an option, called the *FIPS flagger*, which flags all non-standard extensions to SQL and to Module Language, as mandated by the *Federal Information Processing Standard for Database Language SQL*, FIPS publication 127-1. This publication is available at

National Technical Information Service
US Department of Commerce
Springfield VA 22161
U.S.A

Conventions

The following text conventions are used in this document:

Convention	Meaning
boldface	Boldface type indicates graphical user interface elements associated with an action, or terms defined in text or the glossary.
italic	Italic type indicates book titles, emphasis, or placeholder variables for which you supply particular values.
monospace	Monospace type indicates commands within a paragraph, URLs, code in examples, text that appears on the screen, or text that you enter.

1

Introduction to SQL*Module

This chapter introduces you to SQL*Module and answers the following questions:

- What Is SQL*Module?
- SQL*Module as an Interface Builder
- What Is Supported by SQL*Module?
- What SQL Statements are Not Supported by SQL*Module?

What Is SQL*Module?

You use SQL*Module to develop and manage Ada applications that access data in an Oracle database. It allows an Ada application to access relational databases without using embedded SQL, and without using a proprietary application programming interface.

SQL*Module fulfills three roles in application development:

- It compiles SQL standard Module Language files. A Module Language file contains parameterized procedures that encapsulate SQL statements. These procedures are translated by the SQL*Module compiler into calls to a SQL runtime library that interacts with the Oracle server.

- It builds Ada code files that contain interface procedures (often called *stubs*). This allows your application to call stored procedures in an Oracle database directly, without requiring an anonymous PL/SQL block. The interface procedures can be time-stamped, so if the stored procedure is recompiled after the interface procedure was generated, a runtime error occurs.

- SQL*Module can also *generate* stored procedures in a database, by compiling Module Language files, and storing the procedures as part of stored packages, with the appropriate interface mechanism automatically provided in the package's procedure declarations.

Background

The SQL language was designed to operate on entities in a relational database. SQL was not designed to be a general-purpose programming language, and, in fact, it is conceptually very different from 3GLs such as Ada or C. Some of these differences are:

- SQL is a non-procedural database manipulation language, hence it deals with database objects, such as tables, rows, columns, and cursors. A third-generation language deals with data structures such as scalar variables, arrays, records, and lists.

- SQL has a limited ability to express complicated computational operations.

- SQL does not provide the procedural capabilities (such as flow of control statements) that are required to implement efficient programs.

To achieve maximum flexibility when creating large applications, you must combine SQL with host procedural programming language statements. There are several ways to do this, and these are discussed briefly in the next section.

Precompilers

One way to use a procedural language to access data in a relational database is to embed SQL statements directly in a program written in a host 3GL such as C or C++. After the program is coded, you use a precompiler to translate the SQL statements into calls to a runtime library that processes the SQL, and submits the SQL statements to the database.

See Figure 1–1, "Developing with the Precompilers" for details of this process.

While embedded SQL is very useful, it can have drawbacks when very large applications are being developed. There are several reasons for this:

- Use of embedded SQL requires study of the technical details of the precompiler.

- SQL code does not follow the syntactic and semantic constraints of the host language, and can confuse specialized tools, such as syntax-directed editors and "lint" programs.

- When the precompiler processes code that contains embedded SQL, it introduces extra data structures and code in the generated output code, making source-level debugging more difficult.

- Techniques for error handling and recovery in embedded SQL programs can be difficult to understand and apply, and subtle bugs can arise when developers do not appreciate all the problems involved in embedded SQL error handling.

Figure 1–1 shows how you develop applications using the Oracle precompilers.

Figure 1–1   Developing with the Precompilers
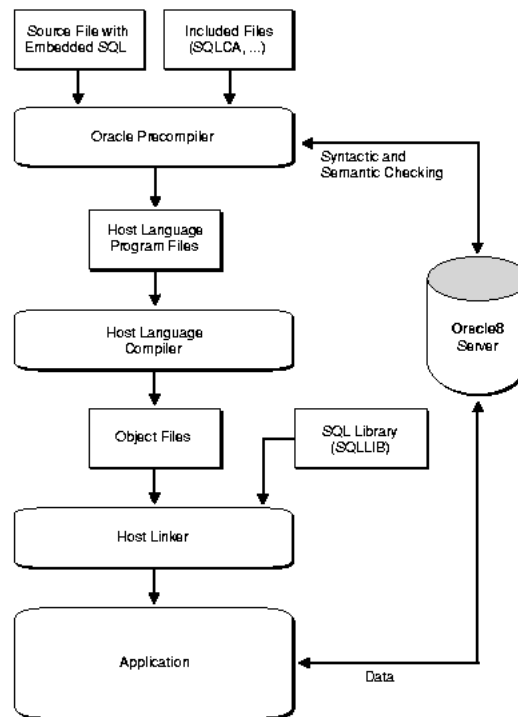
The Module Language Concept

The ANSI SQL standards committee defined the embedded SQL standard in two steps. A formalism called Module Language was defined, then the embedded SQL standard was derived from Module Language.

Using Module Language is very straightforward: place all SQL code in a separate module, and define an interface between the module containing the SQL code and the host program written in Ada. At the most concrete level, the interface simply consists of

- a standard way of calling Module Language procedures from Ada.

- a standard way to return error and warning information

- specification of conversions between SQL datatypes (such as DOUBLE PRECISION or SMALLINT) and host language datatypes or derived types defined in a standard package in Ada

It is also possible to develop more abstract interfaces between the host language and Module Language. One example of this is the SAMeDL (SQL Ada Module Description Language) developed at Carnegie Mellon and the Software Engineering Institute.

Figure 1–2 shows how you would develop an application using SQL standard Module Language.

Figure 1–2   Developing with Module Language
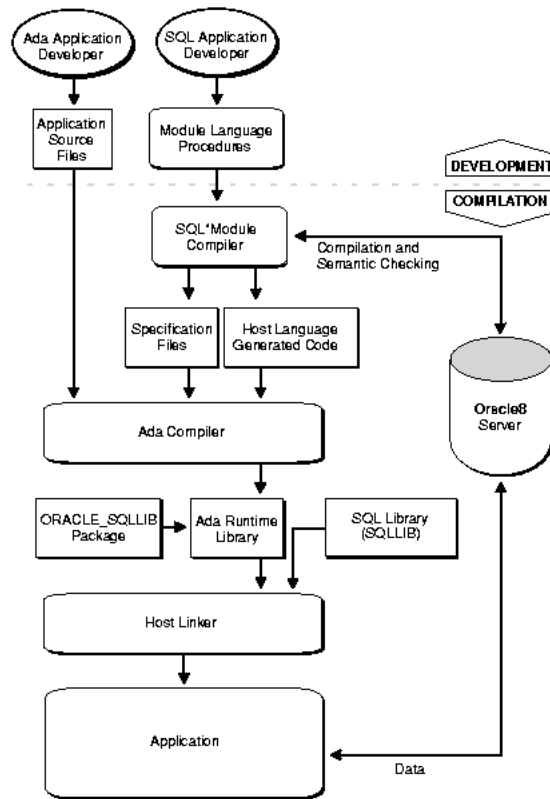
A Module Language compiler such as SQL*Module generates a call-level interface to procedures defined within a module, allowing them to be called like any other host language procedure. Details of the implementation of these procedures are hidden from the application.

The most immediate benefit of this approach is specialization. By separating SQL and the host language, an application developer can focus on using the host language to perform application tasks, and a database developer can focus on using SQL to perform database tasks.

The developer of the application does not need to know SQL. The procedures to be called can be treated as canned routines that return variables of well defined datatypes in the host language. Error handling becomes straightforward, since all module procedures can return a single error indicator.

Figure 1–3 shows the operation of SQL*Module when it is used to compile SQL standard modules. The module file contains a preamble, defining the host 3GL to be Ada, and two simple procedures in Module Language. When SQL*Module compiles this module, it produces two output files: a *source code output file*, that contains calls to the runtime library SQLLIB that do the work of accessing the database, and a *specification file* that declares the procedures in the source code file.

You include the specification files in the host application code that calls the module procedures, using the appropriate language-specific mechanism.

Figure 1–3   Compiling a Module

Specifications File (Output)

```
With Oracle_Sqllib; use Oracle_Sqllib;
with SQL_STANDARD;
Package emp_rec_mod is
. . .
procedure set_comm(emp_number: in sql_standard.int;
   new_comm: in sql_standard.real;
   SQLCODE: out sql_standard.sqlcode_type);
   sql_001 : constant string :=
"UPDATE EMP SET COMM=:NEW_COMM WHERE EMPNO = :EMP_NUMBER ";
procedure get_name(emp_number: in sql_standard.int;
 emp_name: out sql_standard.char;
 sqlstate: out sql_standard.sqlstate_type);
sql_002 : constant string :=
"SELECT ENAME INTO :EMP_NAME FROM EMP" &
   "WHERE EMPNO = :EMP_NUMBER ";
end emp_rec_mod;
```

Module File (Compiled)

```
MODULE emp_rec_mod
LANGUAGE Ada
AUTHORIZATION scott

procedure set_comm (
   :emp_number INTEGER,
   :new_comm REAL,
   SQLCODE);

   UPDATE emp SET comm= :new_comm
      WHERE empno = :emp_number;

procedure get_name (
   :emp_number INTEGER,
   :emp_name CHARACTER(11),
   SQLSTATE);

SELECT ename INTO :emp_name
   FROM emp
   WHERE empno = :emp_number;
```

Source Code Output File (Output)

```
with Oracle_Sqllib; use Oracle_Sqllib;
with emp_rec_mod; use emp_rec_mod;
with SQL_STANDARD; use SQL_STANDARD;
package body emp_rec_mod is
use SQL_STANDARD.CHARACTER_SET;
procedure set_comm(emp_number: in sql_standard.int;
   new_comm: in sql_standard.real;
   SQLCODE: out sql_standard.sqlcode_type) is
   sqlstm : sqlexd;
begin
   . . .
end set_comm;
procedure get_name(emp_number: in sql_standard.int;
   emp_name: out sql_standard.char;
   sqlstate: out sql_standard.sqlstate_type) is
   sqlstm : sqlexd;
   SQLCODE : sql_standard.sqlcode_type := 0;
begin
   . . .
end get_name;
end emp_rec_mod;
```

SQL*Module as an Interface Builder

In addition to its role as a Module Language compiler, SQL*Module can also build host language interfaces to procedures that are stored in an Oracle database.

Figure 1–4 shows schematically how SQL*Module functions as an interface builder. The compiler extracts the interfaces to stored procedures, and creates an Ada output file that contains calls to the procedures. Your Ada application then accesses the stored procedures in the database by calling these interface procedures.

When you create interface procedure files (output files that contain interface procedures for calling stored procedures), you can choose either *early binding* or *late binding*.

The early binding option creates a time stamp in the interface procedure for the time that the stored procedure was last compiled. If the stored procedure has been recompiled after the interface procedure was generated, a runtime error is generated when the interface procedure is called from the host application.

The late binding option calls the stored procedure through an anonymous PL/SQL block, and no time stamp is used. See "Early and Late Binding" on page 3-5 for more information about binding.

Figure 1–4   SQL*Module as an Interface Builder
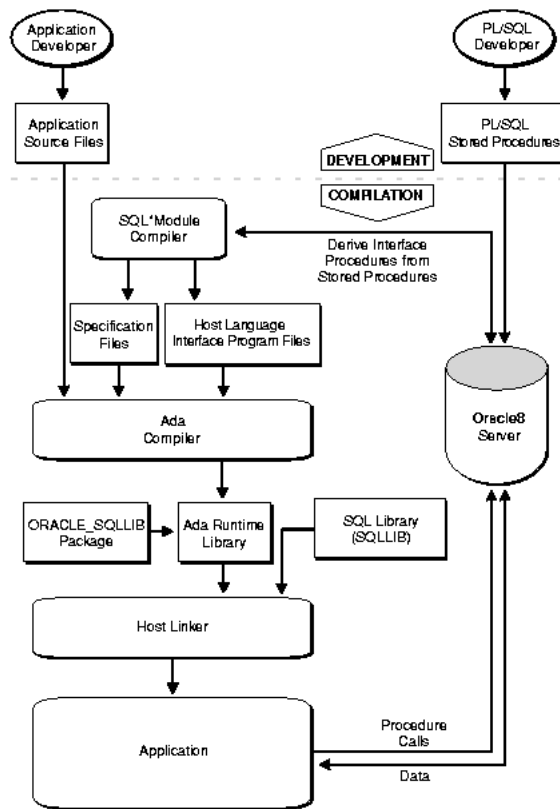
What Is Supported by SQL*Module?

SQL*Module supports international standards for Module Language. Refer to the Preface to this Guide for more information about supported standards. In addition, Oracle has extended the current standard in several ways. For example, datatype conversions between Oracle datatypes and Ada datatypes are defined, comments can be used in a module, and so forth. Chapter 2, "Module Language" describes the Module Language capabilities of SQL*Module in detail. A compile time option, the *FIPS flagger*, is available to flag use of non-standard extensions to Module Language and to SQL.

In addition to the complete Module Language standard, SQL*Module also provides a way for a host application to access PL/SQL stored procedures in an Oracle database. If a package exists in an Oracle database that contains procedures, you can use SQL*Module to build interface procedures corresponding to the PL/SQL procedures in the package. Thus the application can call the stored procedures directly.

SQL*Module generates code and specification files that can be compiled with the Ada compiler.

You can compile your stored procedures or modules to get an interface procedures file and call the modules directly from an Ada host program.

What SQL Statements are Not Supported by SQL*Module?

■   DDL (data definition language) is not supported.

- DML (Data Manipulation Language) statements other than SELECT, UPDATE, DELETE, and INSERT are not supported.

- Transaction control statements other than COMMIT and ROLLBACK, and CONNECT and DISCONNECT are not supported.

2

Module Language

This chapter describes SQL standard Module Language, including Oracle's extensions to standard Module Language. It discusses the following topics:

- The Module
- Structure of a Module
- SQL Datatypes
- SQL Commands
- Text in a Module
- Indicator Parameters
- Status Parameters
- CONNECT Statement
- DISCONNECT Statement
- Multi-tasking
- Multi-tasking Example

This chapter does *not* cover the use of SQL*Module to provide interfaces to stored procedures. See Chapter 3, "Accessing Stored Procedures" for information about calling stored procedures from a host application.

The Module

A *module* is a single file that contains

- introductory material, in a *preamble*
- optional *cursor declarations* for use by queries that can return multiple rows of data
- definitions of *procedures* to be called by the host application

The easiest way to see how a module is constructed is to look at an example. The small module contains a cursor declaration, procedures that open and close the cursor, and a procedure that uses the cursor to fetch data from the familiar EMP table. Two consecutive dashes (- -) begin a comment, which continues to the end of the line. Case is not significant.

An Example Module

```
-- the preamble (contains three clauses)
--
MODULE     EXAMPLE1_MOD    -- Define a module named example1_mod.
```

```
    LANGUAGE   Ada            -- The procedures are compiled into
                              -- Ada, and called from an
                              -- Ada application.

    AUTHORIZATION SCOTT/TIGER  -- Use Scott's tables.
                              -- His password is "tiger"
                              -- (the password does not have to
                              -- be specified in the module file).


-- Declare a cursor for a SELECT statement that
-- fetches four columns from the EMP table.
-- dept_number will be specified
-- in the procedure that opens the cursor.

DECLARE cursor1 CURSOR FOR
    SELECT ename, empno, sal, comm
        FROM emp
        WHERE deptno = :dept_number


-- Define a procedure named "open_cursor1" to open the cursor.
-- After the procedure name is a comma separated parameter list
-- enclosed in ()'s.
PROCEDURE open_cursor1 (
        :dept_number   INTEGER,
        SQLCODE);

    OPEN cursor1;


-- The "fetch_emp_data" procedure gets data from the cursor.
-- SQLCODE will return as 100 when there
-- is no more data.
PROCEDURE fetch_emp_data (
        :empno        INTEGER,
        :empname      VARCHAR2(10),
        :sal          REAL,
        :commission   REAL,
        :comm_ind     SMALLINT,
        SQLCODE);

-- the SQL command is a FETCH on the cursor
    FETCH cursor1
        INTO :empname,
            :empno,
            :sal,
            :commission INDICATOR :comm_ind;

-- Define a procedure to close the cursor.
PROCEDURE close_cursor1 (SQLCODE);
    CLOSE cursor1;


-- Connect to a database
PROCEDURE do_connect (
        :dbname       VARCHAR2(12),
        :username     VARCHAR2(12),
        :passwd       VARCHAR2(12),
        SQLCODE);
```

```
        CONNECT TO :dbname USER :username USING :passwd;


-- Disconnect
PROCEDURE do_disconnect (SQLCODE);

        DISCONNECT CURRENT;
```

> **Note:**   If you are familiar with the syntax of SQL89 Module
> Language, you should note that the newer SQL92 syntax is used in
> this example. This includes parentheses around a comma separated
> parameter list, colons before parameters, and use of the INDICATOR
> keyword. This syntax is supported by the current release of
> SQL*Module, and is used throughout this Guide.

In this example, the LANGUAGE clause (in the second line of the module) specifies
"Ada". This module will be compiled into Ada code. For SQL*Module, this clause is
optional. When present, it is, in effect, a comment.

When SQL*Module compiles this module, it transforms the procedures into Ada
language procedures that open the cursor and call library routines to fetch the data.
SQL*Module also generates a package specification file, which must be compiled into
the Ada library and referenced in the host application using a **with** context clause. See
Chapter 5, "Running SQL*Module" in this Guide for information on running
SQL*Module, and Chapter 6, "Demonstration Programs" for information about Ada
specification files.

A Short Example Program in Ada

To complete the example, a short Ada program that calls the procedures defined in the
module file in "An Example Module" on page 2-1 follows.

```
-- Include TEXT_IO,SQL_STANDARD and EXAMPLE1_MOD package specs.

with
      SQL_STANDARD,
      TEXT_IO,
      EXAMPLE1_MOD;
use
      SQL_STANDARD;


-- Define the main procedure.

procedure EXAMPLE1_DRV is


-- Instantiate new packages for I/O on SQL_STANDARD datatypes.
      package STD_INT_IO is
          new text_io.integer_io(SQL_STANDARD.INT);
      use STD_INT_IO;

      package SQLCODE_IO is
          new text_io.integer_io(SQL_STANDARD.SQLCODE_TYPE);
      use SQLCODE_IO;
```

```
        package STD_FLOAT_IO is
            new text_io.float_io(SQL_STANDARD.REAL);
        use STD_FLOAT_IO;


-- Begin with declarations of all program variables,
-- including parameters for the module procedures.
    SERVICE_NAME         : string(1..12)
                                := "INST1_ALIAS ";
    USERNAME             : string(1..12)
                                := "SCOTT       ";
    PASSWORD             : string(1..12)
                                := "TIGER       ";

    DEPT_NUMBER          : SQL_STANDARD.INT;
    EMPLOYEE_NUMBER      : SQL_STANDARD.INT;
    EMPLOYEE_NAME        : string(1..10);
    SALARY               : SQL_STANDARD.REAL;
    COMMISSION           : SQL_STANDARD.REAL;
    COMM_IND             : SQL_STANDARD.SMALLINT;
    SQLCODE              : SQL_STANDARD.SQLCODE_TYPE;
    LENGTH               : integer;

    CONNECT_ERROR        : exception;
    SQLCODE_ERROR        : exception;

begin

-- Call a module procedure to connect
-- to the Oracle server.
    EXAMPLE1_MOD.DO_CONNECT
        (SERVICE_NAME, USERNAME, PASSWORD, SQLCODE);

-- Test SQLCODE to see if the connect succeeded.
    if SQLCODE /= 0 then
        raise CONNECT_ERROR;
    end if;

    TEXT_IO.NEW_LINE(2);

-- Get an input value from user to use
-- in opening the cursor.
    TEXT_IO.PUT("Enter department number: ");
    STD_INT_IO.GET(DEPT_NUMBER);
    TEXT_IO.NEW_LINE;

-- Call the module procedure to open the cursor.
-- You open the cursor using the input parameter (dept_number).
    EXAMPLE1_MOD.OPEN_CURSOR1(DEPT_NUMBER, SQLCODE);
-- If SQLCODE indicates error, call exception handler.
    if SQLCODE < 0 then
        raise SQLCODE_ERROR;
    end if;

    TEXT_IO.PUT_LINE("Employee  ID Number   Salary  Commission");
    TEXT_IO.PUT("---------------------------------------");

-- Call the FETCH procedure in a loop, to get
-- the employee data.
    loop
```

```
            EXAMPLE1_MOD.FETCH_EMP_DATA
                  (EMPLOYEE_NUMBER,
                   EMPLOYEE_NAME,
                   SALARY,
                   COMMISSION,
                   COMM_IND,
                   SQLCODE);

            TEXT_IO.NEW_LINE;

-- When SQLCODE = 100, no more rows to fetch.
            exit when SQLCODE = 100;

-- When SQLCODE less than 0, an error occurred.
            if SQLCODE < 0 then
                  raise SQLCODE_ERROR;
            end if;

            TEXT_IO.PUT(string(EMPLOYEE_NAME));
            STD_INT_IO.PUT(EMPLOYEE_NUMBER, WIDTH => 9);
            STD_FLOAT_IO.PUT(SALARY, FORE => 6, AFT => 2, EXP => 0);

            if COMM_IND = 0 then
                  STD_FLOAT_IO.PUT(COMMISSION, FORE => 9, AFT => 2, EXP => 0);
            else
                  TEXT_IO.PUT("          Null");
            end if;
        end loop;

        TEXT_IO.NEW_LINE(2);

-- Call the procedure to close the cursor.
        EXAMPLE1_MOD.CLOSE_CURSOR1(SQLCODE);

-- Call the procedure to disconnect.
        EXAMPLE1_MOD.DO_DISCONNECT(SQLCODE);

-- Handle the error exception.

exception
      when SQLCODE_ERROR =>
            TEXT_IO.NEW_LINE(2);
            TEXT_IO.PUT("Error fetching data, SQLCODE returns ");
            PUT(SQLCODE);
            TEXT_IO.NEW_LINE(2);

      when CONNECT_ERROR =>
            TEXT_IO.PUT("Connect failure to " &
                        string(SERVICE_NAME));
            TEXT_IO.NEW_LINE(2);

end EXAMPLE1_DRV;
```

This example demonstrates several important points about SQL*Module:

The types of the parameters in the module procedures are SQL datatypes, for example SMALLINT and REAL, not Ada datatypes.

- Parameters are passed to the generated output code procedures in the normal way. Refer to Chapter 6, "Demonstration Programs" for specific information concerning parameter passing conventions.

- The error status of a call is returned in the SQLCODE or SQLSTATE parameter. There must be a SQLCODE or SQLSTATE status parameter somewhere in the parameter list of each Module Language procedure. See the section "Status Parameters" on page 2-13 for more information.

Structure of a Module

This section is an informal guide to Module Language syntax, including Oracle's extensions to the language. See Appendix B to this Guide for a formal description of Module Language syntax.

A module is contained in a single file and consists of

- a preamble

- zero or more cursor declarations

- one or more procedure definitions

Preamble

The preamble is a set of clauses at the beginning of a module file that describes the module. The clauses are

- MODULE

- LANGUAGE

- AUTHORIZATION

MODULE Clause

The MODULE clause provides a name to the module. The argument is the module name, which must be a legal SQL identifier.

> **Note:** A SQL identifier is a string containing only the 26 letters A through Z, the digits 0 through 9, and the underscore ("_"). The identifier must start with a letter, and cannot be longer than 30 characters (18 characters to comply with the SQL89 standard). You can use lowercase letters (a..z), but a SQL identifier is not case sensitive. So, the identifiers "THIS_IDENTIFIER" and "this_identifier" are equivalent. The characters '$' and '#' are also allowed in a SQL identifier, but you should avoid using them, as they have special meaning in many languages and operating systems.

The module name must be supplied. The module name must be the same as the name of the Ada library unit that you use when you store the Ada-compiled output in your Ada library.

LANGUAGE Clause

The LANGUAGE clause specifies the target language. It takes one argument — the language name, which must be Ada (case is not significant).

The LANGUAGE clause is optional in SQL*Module. See Chapter 5, "Running SQL*Module" for more information about running SQL*Module.

AUTHORIZATION Clause

The AUTHORIZATION clause can determine, or help to determine, the database and schema that SQL*Module uses at compile time.

The argument to the AUTHORIZATION clause can take any one of the following four forms:

- the username: scott

```
- scott
```

- the username plus a password, the two separated by a slash: scott/tiger

```
- scott/tiger
```

- the username plus a database to connect to:

```
scott@{instance_alias | net8_connection_string}
}
```
- a full specification, consisting of the username, a password, and the database to connect to: scott/tiger@{instance_alias | net8_connection_string}

```
- scott/tiger@{instance_alias | net8_connection_string}
```

The instance_alias is an alias containing a database name defined in the tnsnames.ora file. For more information, talk to your database administrator, or see the manual *Oracle Database Net Services Administrator's Guide*.

If you do not include the password in the AUTHORIZATION clause, you can specify it when you run SQL*Module, using the USERID command line option. If you do not specify a USERID value that contains a password, SQL*Module prompts you for a password. If a database name is not supplied, SQL*Module connects using the default database for the specified user.

> **Note:** For security reasons, omit the password in the SQL*Module or in any configuration file. You will be prompted for the password at runtime.

An application that calls module procedures is in effect submitting SQL cursor declarations and SQL statements to Oracle for processing at runtime. The application runs with the privileges of the user *executing* the application, not the schema specified either in the AUTHORIZATION clause or at runtime, when the Module Language code was compiled by SQL*Module.

So, when the application is executed, the user must be able to connect to the database using the same schema as that specified when the modules were compiled, in order to guarantee access to all database objects referenced in the modules.

Cursor Declarations

When a query can return multiple rows of data, you must declare a cursor for the statement. There are two different kinds of cursor that you can use in Module Language. You can declare an ANSI-standard cursor in your module file, and write module language procedures to OPEN the cursor, FETCH from it, and CLOSE it. Or,

you can use a *cursor variable*. Cursor variables are described in "Cursor Variables" on page 3-6.

Using a cursor, you retrieve data one row at a time using the SQL FETCH command. Standard cursors are established in Module Language using the DECLARE CURSOR statement.

> **Note:** The cursor name is a SQL identifier; it is not a procedure name. So, the cursor name does not need to be a valid Ada identifier.

The SQL statement in a DECLARE CURSOR clause must not include an INTO clause. The INTO clause is used in the FETCH command in the procedure that references the cursor.

You cannot use parameters when you *declare* a standard cursor. A placeholder is used instead. Placeholders in a cursor declaration must correspond by name to the parameters in the procedure that opens the cursor.

Note the way the variable *dept_number* is used in the following module fragment, taken from the example used earlier in this chapter:

```
DECLARE cursor1 CURSOR FOR
     SELECT ename, empno, sal, comm
     FROM emp
     WHERE deptno = :dept_number

PROCEDURE open_cursor1 (:dept_number INTEGER, SQLCODE);
     OPEN cursor1;
```

The cursor *declaration* is *NOT* terminated with a semicolon. If it is, SQL*Module generates a syntax error.

> **Caution:** If the application consists of several modules, a cursor cannot be declared in one module, then referenced in another. Nevertheless, cursor names must be unique across all modules in the application.

Procedure Definitions

A procedure definition consists of a

- procedure name
- parameter list
- single SQL statement

Procedure Name

The procedure name is a SQL identifier, and should also be a legal Ada identifier. Procedure names must be unique in the application.

Each procedure in a module must have a distinct name. The procedure name space extends over all modules of an application, so when the application uses more than one module, each procedure must still have a distinct name.

Parameter List

The parameter list contains one or more formal parameters. Each parameter must have a distinct name, which is a SQL identifier. One of the parameters must be a status parameter: SQLSTATE or SQLCODE. It can appear anywhere in the parameter list. You can include both. See "Status Parameters" on page 2-13 for more information about status parameters.

SQL92 Syntax

In SQL89 Module Language, you defined a procedure using the syntax

```
PROCEDURE proc_name
        <param_1>    <datatype_1>
        <param_2>    <datatype_2>
        SQLCODE;
    <sql_statement>;
```

where <param_n> is a formal parameter name and <datatype_n> is a SQL datatype. Following the SQL92 standard, SQL*Module allows the syntax

```
PROCEDURE proc_name (
        :<param_1>    <datatype_1>,
        :<param_2>    <datatype_2>,
        SQLSTATE | SQLCODE );
    <sql_statement>;
```

where the parameter list is surrounded by parentheses, and parameters are separated by commas.

> **Note:** You cannot mix SQL89 and SQL92 syntax. If you separate the elements of the parameter list using commas, you must also place parentheses around the parameter list.

When SQL*Module generates the output procedures, the formal parameters appear with the same names, and in the same order, as they appear in the module procedure. You can use the parameter access conventions appropriate for the Ada language when calling the output procedure from the application. Thus, Ada programmers can use named parameter association in place of, or in addition to, positional parameter association.

SQL Datatypes

The following table lists the SQL and Oracle datatypes that you can use in a module parameter list. For more information about arrays, see "Arrays as Procedure Arguments" on page 4-7.

Table 2–1   Datatypes

SQL Datatype	Meaning
CHARACTER	single character
CHARACTER(L)	character string of length L bytes
DOUBLE PRECISION	approximate numeric
INTEGER	exact numeric, no fractional part
REAL	approximate numeric

Table 2–1   (Cont.) Datatypes

SQL Datatype	Meaning
SMALLINT	exact numeric, no fractional part, equal to or smaller in range than INTEGER
Oracle Datatype	
VARCHAR2(L)	variable-length character string of length L bytes
SQL*Module Datatypes: SQL_CURSOR	cursor variable type
SQL_CONTEXT	task context
ARRAY(N) OF SQL_ CURSOR	Arrays of SQL Datatypes shown. N is the number of elements.
ARRAY(N) OF CHARACTER	
ARRAY(N) OF CHARACTER(L)	
ARRAY(N) OF DOUBLE PRECISION	
ARRAY(N) OF INTEGER	
ARRAY(N) OF REAL	
ARRAY(N) OF SMALLINT	
ARRAY(N) OF VARCHAR2(L)	

Note: CHARACTER can be abbreviated CHAR. INTEGER can be abbreviated INT.

The SQL standard for Module Language allows the use of only a subset of the SQL datatypes for Ada.

Note: All parameters for Module Language procedures must be scalars, arrays, or strings. Records and access types are not supported.

SQL Commands

Module Language supports the following SQL statements:

- ALLOCATE
- CLOSE
- COMMIT
- CONNECT TOs
- CONTEXT ALLOCATE
- CONTEXT FREE
- DELETE
- DISCONNECT

- ENABLE THREADS

- FETCH

- INSERT

- OPEN

- ROLLBACK

- SELECT

- SET CONNECTION

- UPDATE

The DELETE and UPDATE commands may be either searched (the normal mode) or positioned (using the WHERE CURRENT OF <cursor_name> clause). You can use the OPEN command only for ANSI-standard cursors. You must open cursor variables on the Oracle Server, using PL/SQL code.

Text in a Module

In general, Module Language is not case sensitive. You can enter keywords and identifiers in any mixture of uppercase and lowercase. However, case is significant in character string literals.

Text in a module file is free form. White space (spaces, tabs, and new lines) can be placed anywhere in the file to improve readability. The only exception to this is that identifiers, keywords, and string literals cannot be continued to a new line.

The maximum length of a line in a module is 512 characters.

Comments

SQL*Module allows comments in a module file. You can place comments anywhere that white space can appear, except in string literals.

There are two comment styles: SQL-style comments and C-style comments. A SQL-style comment starts with two consecutive dashes, which can appear anywhere on a line, and ends at the end of the line. For example:

```
-- This is a SQL(or Ada)style comment.
-- For multiline comments, you must place the comment
-- dashes on each line.
```

A C-style comment begins with a slash immediately followed by an asterisk (/*), and ends at the next occurrence of an asterisk immediately followed by a slash (*/). C-style comments can span more than one line. C-style comments cannot be nested.

The following example demonstrates the C-style comment:

```
/*
 *  This comment style is often used to
 *  introduce a procedure.
 */
```

Indicator Parameters

You use indicator parameters to set the null/not null status of another (associated) parameter, or to "indicate" if a column value retrieved on a query was null. In the

module procedure parameter list, an indicator parameter always has a SMALLINT type. In the SQL statement, the indicator parameter follows the associated parameter, with no comma separation. The SQL92 standard allows the keyword INDICATOR to be used to separate the indicator parameter and its associated parameter.

In the following example, *grade_indic* is an indicator parameter:

```
PROCEDURE get_grade (
        :grade          REAL,
        :grade_indic    SMALLINT,
        :class_number   INTEGER,
        :student_id     INTEGER,
        SQLCODE);
    SELECT grade
        INTO :grade INDICATOR :grade_indic
        FROM enrollment
        WHERE class_no = :class_number AND student_id = :student_id;
```

Following the SQL89 standard, the SELECT statement would be written without the INDICATOR keyword, as follows:

```
    SELECT grade
        INTO :grade :grade_indic
        FROM enrollment
        WHERE class_no = :class_number AND student_id = :student_id;
```

SQL*Module allows both uses of indicator parameters.

When an indicator parameter is returned from a procedure (an OUT parameter), as in the query example, its returned value has the following meanings:

-1

The Oracle column or expression is null. The value of the associated parameter (*grade* in this example) is indeterminate.

0

Oracle assigned a column or expression value to the associated parameter.

> 0

For character data, Oracle passes a truncated column value in the associated parameter. The value of the indicator parameter shows the original length of the value in the database column.

When indicator parameters are passed as IN parameters, you must set the value in your Ada program. A value of -1 means that Oracle will assign null to the column (regardless of the value of the associated parameter), and a value of zero or greater means that Oracle will use the value of the associated parameter in the UPDATE or INSERT command. Positive values greater than zero have no special meaning; they are equivalent to zero.

> **Caution:** If you try to assign a null to a database column that has the NOT NULL constraint, an Oracle error is returned at runtime.

The following code fragment shows an Ada driver that calls a Module Language procedure with a null indicator parameter value:

```
with SQL_STANDARD;
procedure DRV is
  SQLCODE    : SQL_STANDARD.SQLCODE_TYPE;
```

```
  EMPLOYEE   : string(1..10) := "SCOTT     ";
  COMMISSION : SQL_STANDARD.REAL := 2000.0;
  COMM_IND   : SQL_STANDARD.SMALLINT := -1;
begin
  . . .

  UPDATE_COMMISSION(EMPLOYEE, COMMISSION, COMM_IND, SQLCODE);

  . . .
end;
```

The corresponding Module Language procedure is:

```
PROCEDURE update_commission (
  :employee_name VARCHAR2(10),
  :commission REAL,
  :comm_ind SMALLINT,
  SQLCODE);

  UPDATE emp SET comm = :commission INDICATOR :comm_ind
     WHERE ename = :employee_name;
```

In this example, the parameter *commission* with a value of 2000.0 is passed to the *update_commission* procedure. But, since the indicator parameter is set to -1, employee Scott's commission is set to null in the EMP table.

Status Parameters

There are two special status parameters: SQLSTATE and SQLCODE. The status parameters return values to the calling Ada application that show if

- the procedure completed without error

- an exception occurred, such as "no data found"

- an error occurred

You can place either one or both of the status parameters anywhere in the parameter list. They are always shown last in this Guide, but that is just an informal coding convention. The status parameters are not preceded by a colon, and they do not take a datatype specification. You cannot directly access the status parameters in the SQL statement in the procedure; they are set by SQL*Module.

SQLSTATE is the preferred status parameter; SQLCODE is retained for compatibility with older standards.

For more information about the status parameters and the values they return, see "Error Handling" on page 4-2.

Error Messages

SQL*Module compile time error messages have the MOD prefix. The codes and messages are listed in *Oracle Database Error Messages*.

CONNECT Statement

The connect statement associates a program with a database, and sets the *current connection.* The syntax of the command is shown in the following syntax diagram. Key words, which you must spell exactly as shown, are in upper case; tokens you supply are in lower case. If a clause is optional, it is shown off the main path, which flows to

from left to right. For a more complete discussion of syntax diagrams, see Appendix B, "Module Language Syntax".

A *db-env* (database environment) is a Oracle Net connect string or instance-alias. The *conn-name* (connection name) is optional. For multiple connections, you must specify the connection names. You can omit it if you will only use a single connection. The USING clause is optional. A *passwd is the password.*

Connecting as DEFAULT results in a connection to Oracle by using either TWO_TASK (if it applies to your operating system) or ORACLE_SID and the account specified by the parameter *os_authent_prefix* in your file *init.ora*. The optional token *passwd* is the password.

The ANSI SQL92 standard does not allow *db-env* to be optional. This is an Oracle extension (which will be flagged by the FIPS option) which enables a connection to the default server as a specific user.

You must use Oracle Net in SQL*Module applications. Note that *passwd* can only be a variable and not a character string literal. All other variables can be either a character string literal or a variable previously defined, preceded by ":".

Here is a illustrative code fragment from a module named *demo.mad*, which contains the following procedure to do a connect:

```
...
PROCEDURE ora_connect (:dbid   VARCHAR2(14),
                       :dbname VARCHAR2(14),
                       :uid    VARCHAR2(14),
                       :pwd    VARCHAR2(14),
                       SQLCODE);
CONNECT TO :dbid AS :dbname USER :uid USING :pwd;
...

An Ada application can contain these statements to do the connect:
...
pwd : constant string := "tiger";
...
DEMO.ORA_CONNECT("inst1","RMT1","scott",PWD,SQLCODE);
if SQLCODE /= 0 then
...
```

For more information, see the *Oracle Database Net Services Administrator's Guide*.

SET CONNECTION Statement

The *set connection* statement sets the current connection. Its syntax is:

SET CONNECTION { connection-name | DEFAULT }

DEFAULT is a special case of the connection-name, '/', at the current ORACLE_SID.

DISCONNECT Statement

The disconnect command ends an association between an application and a database environment. It can be summarized as:

DISCONNECT { connection-name | ALL | CURRENT | DEFAULT}

The full ANSI semantics are not followed. Instead of raising an exception condition if the connection has an active transaction, the transaction is (implicitly) rolled back and the connection(s) disconnected.

DISCONNECT ALL only disconnects connections which are established by SQLLIB (that is, by SQL*Module).

DISCONNECT DEFAULT and DISCONNECT *connection-name* terminate only the specified connection.

DISCONNECT CURRENT terminates the connection regarded by SQLLIB as the "current connection" (either the last connection established or the last connection specified in a SET CONNECTION statement).

After the current connection is disconnected, you must execute a set connection or a connect statement to establish a new current connection. Then you can execute any statement that requires a valid database connection.

You must explicitly commit or roll back the current transaction before disconnecting. If you do not commit the transaction before disconnecting, or if the program exits before committing, the current transaction is automatically rolled back.

Here is an example of two procedures from the module demo.mad illustrating the disconnect command:

```
 ...
PROCEDURE ora_disconnect (:dbname VARCHAR2(14),
                          SQLCODE) ;
   DISCONNECT :dbname;


...
PROCEDURE ora_commit(SQLCODE);
   COMMIT WORK;

these procedures are called from your application as follows:
...
DEMO.ORA_COMMIT(SQLCODE);
DEMO.ORA_DISCONNECT("RMT1", SQLCODE);
...
```

A required commit command was executed using the procedure ora_commit (which is also in the file demo.mad) just before the disconnect.

Multi-tasking

Starting with release 8.0, multi-tasking Ada programs are supported by SQL*Module. The new commands that you use in a multi-tasking program are described in the following sections:

ENABLE THREADS

This command initializes the process for later use with Ada tasks. It must be called prior to the creation of any Ada tasks. It is:

```
ENABLE THREADS;
```

SQL_CONTEXT Datatype

The datatype *SQL_CONTEXT* is used to support multi-tasking applications. It points to SQLLIB's runtime context. You pass the context as a parameter to SQL*Module procedures. If it is passed, then it indicates which SQLLIB runtime context will be used for execution. If no SQL_CONTEXT parameter is passed, then SQL*Module uses the global runtime context of SQLLIB.

For example, here is a procedure that uses *SQL_CONTEXT*:

```
PROCEDURE seldept (:ctx SQL_CONTEXT,
                   :dno INTEGER,
                    SQLCODE);
    SELECT deptno INTO :dno FROM emp WHERE dname = 'SALES';
```

In this example, the select statement will use the runtime context pointed to by the variable *ctx*. *ctx* must have been previously allocated in a CONTEXT ALLOCATE statement. Note that you never reference the SQL_CONTEXT variable directly. It appears only in the code that SQL*Module generates.

CONTEXT ALLOCATE

This command allocates storage in memory for a SQLLIB runtime context that is used to support multi-tasking. An example is:

```
CONTEXT ALLOCATE :ctxvar;
```

ctxvar is of type SQL_CONTEXT. If sufficient storage is available, *ctxvar* will contain the address of the runtime context. Otherwise, *ctxvar* will be zero and an error will be returned in the provided status variables.

CONTEXT FREE

CONTEXT FREE frees all storage associated with the SQLLIB runtime context. It does not disconnect any active connection. Prior to deallocating the runtime context, you must execute the DISCONNECT command for each active connection.

The CONTEXT FREE statement always succeeds and *ctxvar* is zero after execution. If the context is *ctxvar*, then, an example is:

```
CONTEXT FREE :ctxvar;
```

Multi-tasking Restrictions

■   All database connections must be established prior to task activation.

■   Multi-tasking applications cannot be used to access database stored procedures.

Multi-tasking Example

Here is part of a module, adademo.mad:

```
PROCEDURE enable_threads (SQLCODE);
    ENABLE THREADS;
PROCEDURE allocate_context (:ctx SQL_CONTEXT, SQLCODE);
    CONTEXT ALLOCATE :ctx;
PROCEDURE free_context (:ctx SQL_CONTEXT, SQLCODE);
    CONTEXT FREE :ctx;
PROCEDURE disconn_db (:ctx SQL_CONTEXT,
                      :dbname VARCHAR2(14),
```

```
                              SQLCODE);
    DISCONNECT :dbname;

these procedures are called as follows:
with ADADEMO;
-- Declare contexts CTX1, CTX2
...
ADADEMO.ENABLE_THREADS(SQLCODE):

ADADEMO.ALLOCATE_CONTEXT(CTX1, SQLCODE);
ADADEMO.ALLOCATE_CONTEXT(CTX2, SQLCODE);
-- Spawn tasks that process data:
...

An example of explicitly disconnecting and freeing contexts is:
-- After processing data:
...
ADADEMO.DISCONN_DB(CTX1, DBNAM1);
ADADEMO.DISCONN_DB(CTX2, DBNAM2);
ADADEMO.FREE_CONTEXT(CTX1);
ADADEMO.FREE_CONTEXT(CTX2);
```

3

Accessing Stored Procedures

This chapter describes how to use SQL*Module to generate interface procedures to call stored procedures. It covers the following topics:

- PL/SQL
- Stored Procedures
- Stored Packages
- Accessing Stored Procedures
- Case of Package and Procedure Names
- Early and Late Binding
- Cursor Variables
- Dynamic SQL
- The WITH INTERFACE Clause
- Storing Module Language Procedures
- Connecting to a Database

> **Note:** The examples in this chapter use the tables defined in Chapter 6, "Demonstration Programs".

PL/SQL

This section contains a brief overview of PL/SQL, Oracle's procedural language extension to SQL. PL/SQL is a modern block-structured language that enables you to

- declare constants and variables
- control execution flow, using IF ... THEN ... ELSE, EXIT, GOTO, and other procedural constructs
- create loops, using WHILE ... LOOP and FOR ... LOOP
- assign constant or variable expressions to a variable
- issue SQL Data Manipulation Language and Transaction Control statements
- define exceptions, handle them using WHEN EXCEPTION_NAME THEN ..., and raise them using RAISE EXCEPTION_NAME

See the *Oracle Database PL/SQL Language Reference* for complete information about the PL/SQL language.

Procedures

A PL/SQL procedure is a named PL/SQL block. Unlike an anonymous block, a procedure can

- take parameters

- be invoked from a separate application

- be compiled once, but invoked many times

- be stored in compiled form in a database, independent of the shared SQL cache

A procedure contains one or more PL/SQL blocks. The following example computes the grade point average. The student ID number is passed as a parameter to the procedure, and the computed grade point average is returned by the procedure.

```
PROCEDURE get_gpa(
          student_id IN NUMBER,
          gpa        OUT NUMBER) IS
   n               NUMBER;
   grade_temp      NUMBER;
   gpa_temp        NUMBER; -- needed because PL/SQL cannot read
                           -- an OUT parameter like GPA
   CURSOR c1(sid) IS
       SELECT grade FROM enrollment
           WHERE student_id = sid;

   BEGIN
       n := 0;
       gpa := 0;
       OPEN c1(student_id);
       LOOP
           FETCH c1 INTO grade_temp;
           EXIT WHEN c1%NOTFOUND;      -- c1%NOTFOUND is TRUE
                                       -- when no more data found
           gpa_temp := gpa_temp + grade_temp;
           n := n + 1;
       END LOOP;
       IF n > 0 THEN
           gpa := gpa_temp / n;
       END IF;
       CLOSE c1;
   END;
END PROCEDURE get_gpa;
```

The procedure declaration adds a parameter list to the PL/SQL block. In this example, student_id is a parameter whose mode is IN. The mode of a parameter indicates whether the parameter passes data to a procedure (IN), returns data from a procedure (OUT), or can do both (IN OUT). The parameter gpa is an OUT parameter. It returns a value, but you cannot use it to pass a value to the procedure. Nor can you read its value inside the procedure, even after a value has been assigned to it.

Stored Procedures

You can store PL/SQL procedures in the database, and call these stored procedures from Oracle applications. Storing a procedure in the database offers many advantages. Only one copy of the procedure needs to be maintained; it is in the database, and it can be accessed by many different applications. This considerably reduces maintenance requirements for large applications. A stored procedure is not recompiled each time it is called.

You can store procedures in the database by using Oracle tools such as SQL*Plus. You create the source for the procedure using your text editor, and execute the source using SQL*Plus (for example, with the @ operator). When you input the source, use the CREATE PROCEDURE command. (You can also use CREATE OR REPLACE PROCEDURE, to replace an already stored procedure of the same name.)

See the *Oracle Database Reference* for complete information about the CREATE PROCEDURE command.

Stored Packages

The examples of stored procedures shown so far in this chapter involve standalone procedures (sometimes called top-level procedures). These are useful in small applications. However, to gain the full power of stored procedures, you should use packages.

A package encapsulates procedures, as well as other PL/SQL objects. Stored packages that are used with Ada applications have two parts: a package specification and a package body. The specification is the (exposed) interface to the host application; it declares the procedures that are called by the application. A complete PL/SQL package specification can also declare functions, as well as other PL/SQL objects such as constants, variables, and exceptions. However, an Ada application using SQL*Module cannot access or reference PL/SQL objects other than subprograms. The package body contains the PL/SQL code that defines the procedures and other objects that are declared in the package specification.

Although an Ada application can only access public subprograms, a called subprogram can in turn call private subprograms, and can access public and private variables and constants in the package.

For complete information about stored packages, see the *Oracle Database PL/SQL Language Reference*.

Accessing Stored Procedures

You can use SQL*Module to provide a bridge that enables your host application to access procedures stored in the database. A host application written in Ada cannot call a stored database subprogram directly. But you can use SQL*Module to construct an interface procedure ("stub") that calls the stored database subprogram. Table 3–1 shows, in schematic form, how this process works.

Figure 3–1   Accessing a Stored Procedure
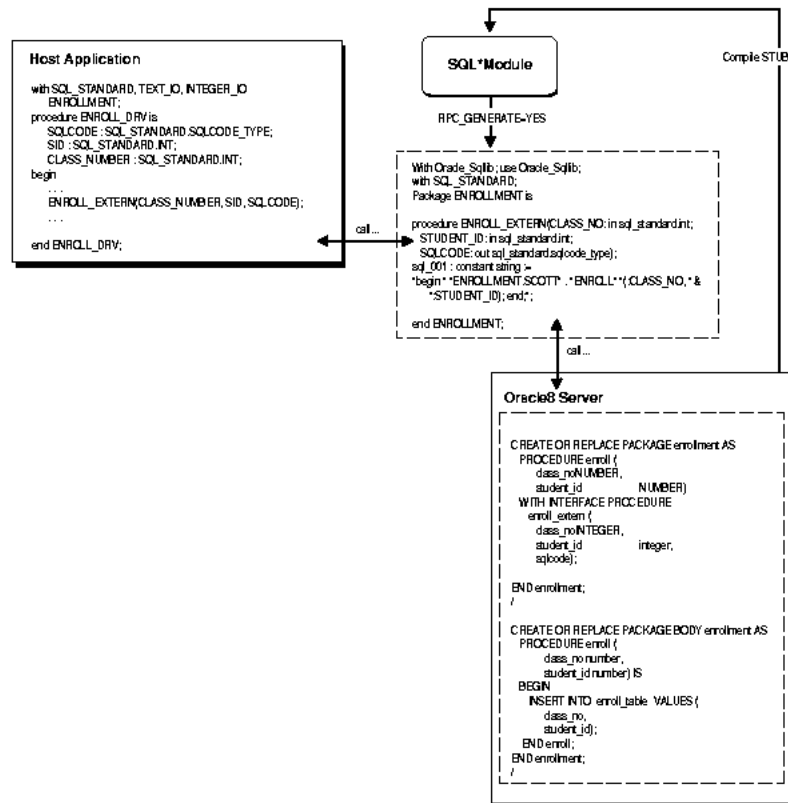
In this example, there is a procedure stored in the database called enroll. The PL/SQL source code that created the procedure is shown in the right-hand box. The WITH INTERFACE clause in the procedure is described in the section "The WITH INTERFACE Clause" on page 3-9. The procedure has two database parameters: class_no and student_id. The SQLCODE error return parameter is added in the interfacing clause.

Case of Package and Procedure Names

The Oracle Server always translates the names of database objects to uppercase when they are inserted into the database. This includes the names of packages and procedures. For example, if you are loading a package into the database in the SCOTT schema, and have a PL/SQL source file that contains the line

```
CREATE PACKAGE school_records AS ...
```

then Oracle inserts the name into the schema as SCHOOL_RECORDS, not the lowercase "school_records". The following SQL*Module command (in UNIX)

```
modada rpc_generate=yes pname=school_records userid=scott
```

generates an error, since there is no package named "school_records" in the schema.

If you prefer to have your package and procedure names stored in lowercase in the database, you must quote all references to the name in the PL/SQL source file, or as you insert them into the database using SQL*Plus. So, you would code

```
CREATE PACKAGE "school_records" AS ...
```

Note also that SQL*Module preserves the case of subprogram names when creating interface procedure files.

However, if you really do want uppercase names, some operating systems (OPEN VMS is an example) require that you quote the name when you specify it on the command line. So, you would enter the command as

```
modada rpc_generate=yes pname="SCHOOL_RECORDS" user=scott
```

See your system-specific Oracle documentation, and your operating system documentation, for additional information on case conventions for command lines that are in effect for your operating system.

Early and Late Binding

When you generate RPCs (remote procedure calls) using SQL*Module, you have a choice of early binding or late binding. Your choice of early or late binding is controlled by the BINDING option.

When you choose early binding, SQL*Module generates a call to the procedure stored in the database, and also uses a time stamp that is associated with the call. The time stamp records the date and time (to the nearest second) that the stored procedure was last compiled. The time stamp is created by the Oracle database. If a host application calls the stored procedure through the interface procedure, and the time stamp recorded with the interface procedure is earlier than the time stamp on the stored procedure recorded in the database, an error is returned to the host application in the SQLCODE or SQLSTATE status parameter. The SQLCODE error is 4062 "time stamp of name has been changed".

The late binding option, on the other hand, does not use a time stamp. If your application calls a stored procedure that has been recompiled since SQL*Module generated the interface procedure, no error is returned to the application.

With late binding, SQL*Module generates the call to the stored procedure using an anonymous PL/SQL block. The following example shows a specification for a stored procedure that is part of a package in the SCOTT schema:

```
PACKAGE emppkg IS

    PROCEDURE get_sal_comm (emp_num      IN   NUMBER,
                            salary       OUT  NUMBER,
                            commission   OUT  NUMBER)
    WITH INTERFACE
    PROCEDURE get_sal_emp  (
                            emp_num      INTEGER,
                            salary       REAL,
                            commission   REAL INDICATOR comm_ind,
                            comm_ind     SMALLINT,
                            SQLCODE);
END emppkg;
```

If you generate an RPC interface procedures output file for the package using the command

```
modada pname=EMPPKG rpc_generate=yes binding=late userid=scott/tiger
```

SQL*Module generates a call in the output file, as follows:

```
With Oracle_Sqllib; use Oracle_Sqllib;
with SQL_STANDARD;
```

```
Package EMPPKG is

procedure GET_SAL_EMP(EMPNUM: in sql_standard.int;
 SALARY: out sql_standard.real;
 COMMISION: out sql_standard.real;
 COMM_IND: out sql_standard.smallint;
 SQLCODE: out sql_standard.sqlcode_type);
sql_001 : constant string :=
"begin ""EMPPKG.SCOTT""." &
    """GET_SAL_COMM""(:EMPNUM, :SALARY, :COMMISION:COMM_IND); end;";

end EMPPKG;
...
```

In other words, the call to the stored procedure get_sal_comm is performed using an anonymous PL/SQL block. This is the way stored procedures are called from an Oracle precompiler or Oracle Call Interface application.

The advantages of late binding are

- greater flexibility

- changes in the stored procedure(s) are transparent to the user

- gives behavior similar to interactive SQL (for example, SQL*PLus)

The disadvantages of late binding are

- There might be additional performance overhead at runtime, due to the necessity of compiling the PL/SQL anonymous block.

- It is difficult to detect runtime PL/SQL compilation errors in the host application. For example, if the anonymous block that calls the late-bound procedure fails at runtime, there is no convenient way for the host application to determine the cause of the error.

- The lack of time-stamp capability means that changes, perhaps radical changes, in the stored procedure could be made after the host application was built, and the application would have no way of detecting this.

Use the BINDING={EARLY | LATE} command line option to select early or late binding when generating RPC interface procedures. See Chapter 5, "Running SQL*Module" for a description of this and other command line options.

Cursor Variables

You can use cursor variables in your application. A cursor variable is a reference to a cursor that is defined and opened on the Oracle Database version 8 server. See the *Oracle Database PL/SQL Language Reference* for complete information about cursor types.

The advantages of cursor variables are

- Encapsulation: queries are centralized, placed in the stored procedure that opens the cursor variable. The logic is hidden from the user.

- Ease of maintenance: if you need to change the cursor, you only need to make the change in one place: the stored procedure. There is no need to change each application.

- Convenient security: the user of the application is the username used when the application connects to the server. The user must have execute permission on the stored procedure that opens the cursor. But the user does not need to have read

permission on the tables used in the query. This capability can be used to limit access to the columns and rows in the table.

Cursor Variable Parameters

You define a cursor variable parameter in your module by using the type SQL_CURSOR. For example:

```
PROCEDURE alloc_cursor (
        SQLCODE,
        :curs SQL_CURSOR);
```

In this example, the parameter curs has the type SQL_CURSOR.

Allocating a Cursor Variable

You must allocate the cursor variable by using the Module Language command ALLOCATE. For example, to allocate the SQL_CURSOR curs that is the formal parameter in the example, you write the statement:

```
ALLOCATE :curs;
```

> **Note:** You use the ALLOCATE command only for cursor variables. You do not need to use it for standard cursors.

Opening a Cursor Variable

You must open a cursor variable on the Oracle Server. You cannot use the OPEN command that you use to open a standard cursor to open a cursor variable. You open a cursor variable by calling a PL/SQL stored procedure that opens the cursor (and defines it in the same statement).

For example, consider the following PL/SQL package, stored in the database:

```
CONNECT scott/tiger

CREATE OR REPLACE PACKAGE cursor_var_pkg AS

    TYPE emp_record_type IS RECORD (ename EMP.ename%TYPE;);
    TYPE curtype IS REF CURSOR RETURN emp_record_type;

    PROCEDURE OPEN1(cur1 IN OUT curtype)
    WITH INTERFACE
    PROCEDURE
        OPEN1 (SQLCODE integer, cur1 SQL_CURSOR);

end cursor_var_pkg;

CREATE OR REPLACE PACKAGE BODY cursor_var_pkg AS

    PROCEDURE OPEN1(cur1 IN OUT curtype) IS
    BEGIN
        OPEN cur1 FOR SELECT ename FROM emp_view;
    END;
END cursor_var_pkg;

COMMIT;
```

After you have stored this package, and you have generated the interface procedures, you can open the cursor curs by calling the OPEN1 stored procedure from your Ada driver program. You can then call module procedures that FETCH the next row from the opened cursor. For example:

```
PROCEDURE fetch_from_cursor (
    SQLCODE,
    :curs SQL_CURSOR,
    :emp_name VARCHAR2(11));

  FETCH :curs INTO :emp_name;
```

In your driver program, you call this procedure to fetch each row from the result defined by the cursor. When there is no more data, the value +100 is returned in SQLCODE.

> **Note:** When you use SQL*Module to create the interface procedure to call the stored procedure that opens the cursor variable, you must specify BINDING=LATE. Early binding is not supported for cursor variables in this release.

Opening in a Stand-alone Stored Procedure

In the example, a cursor type was defined inside a package, and the cursor was opened in a procedure in that package. But it is not always necessary to define a cursor type inside the package that contains the procedures that open the cursor.

If you need to open a cursor inside a standalone stored procedure, you can define the cursor in a separate package, then reference that package in the standalone stored procedure that opens the cursor. Here is an example:

```
PACKAGE dummy IS
    TYPE EmpName IS RECORD (name VARCHAR2(10));
    TYPE emp_cursor_type IS REF CURSOR RETURN EmpName;
END;
-- and then define a standalone procedure:
PROCEDURE open_emp_curs (
      emp_cursor IN OUT dummy.emp_cursor_type;
    dept_num   IN    NUMBER) IS
    BEGIN
       OPEN emp_cursor FOR
           SELECT ename FROM emp WHERE deptno = dept_num;
    END;
END;
```

Return Types

When you define a reference cursor in a PL/SQL stored procedure, you must declare the type that the cursor returns. See the Oracle Database PL/SQL Language Reference for complete information on the reference cursor type and its return types.

Closing a Cursor Variable

Use the Module Language CLOSE command to close a cursor variable. For example, to close the emp_cursor cursor variable that was OPENed in the examples, use the statement

```
CLOSE :emp_cursor;
```

Note that the cursor variable is a parameter, and so you must precede it with a colon.

You can reuse ALLOCATEd cursor variables. You can OPEN, FETCH, and CLOSE as many times as needed for your application. However, if you disconnect from the server, then reconnect, you must reallocate cursor variables.

Restrictions on Cursor Variables

The following restrictions apply to the use of cursor variables:

1. You can only use cursor variables with the commands:

- ALLOCATE

- FETCH

- CLOSE

2. The DECLARE CURSOR command does not apply to cursor variables.

- You cannot FETCH from a CLOSEd cursor variable.

- You cannot FETCH from a non-ALLOCATEd cursor variable.

- Cursor variables cannot be stored in columns in the database.

- A cursor variable itself cannot be declared in a package specification. Only the type of the cursor variable can be declared in the package specification.

- A cursor variable cannot be a component of a PL/SQL record.

Dynamic SQL

Dynamic SQL is the capability of executing SQL commands that are stored in character string variables. The package DBMS_SQL parses data definition language (DDL) and Data Manipulation (DML) statements at runtime. DBMS_SQL has functions such as OPEN_CURSOR, PARSE, DEFINE_COLUMN, EXECUTE, FETCH_ROWS, COLUMN_VALUE, and so on. Use these functions in your program to open a cursor, parse the statement, and so on.

For more details on this package, see Oracle Database Advanced Application Developer's Guide

The WITH INTERFACE Clause

The stored procedure format in the previous section can be used for stored procedures that are to be called from applications written using Oracle tools. For example, a SQL*Plus script can call the GET_GPA procedure in "Procedures" on page 3-2 just as it is written.

You can code a WITH INTERFACE clause, or you can let SQL*Module generate a default WITH INTERFACE clause for stored procedures that have been stored without this clause.

This clause, when added to a procedure declaration in the package specification, lets you add parameters that are essential to perform an RPC to a PL/SQL procedure, through a calling interface procedure in the output file. In addition, the WITH INTERFACE clause uses SQL datatypes, not the PL/SQL datatypes that are used in the stored procedure definition. The additional features of the WITH INTERFACE clause are

- use of SQL datatypes

- optional indicator parameters

- use of the SQLSTATE and SQLCODE status parameters

Note: The procedures names that you code in WITH INTERFACE clauses must be unique within the entire application. If you let SQL*Module generate default WITH INTERFACE, then overloaded procedure names are resolved using an algorithm described in "MAPPING" on page 5-13.

Arrays are not allowed in WITH INTERFACE clauses.

Examples

The following package declaration shows how you use the WITH INTERFACE clause to map PL/SQL datatypes to SQL datatypes, and add the SQLCODE or SQLSTATE status parameters. Status parameters are filled in automatically as the procedure executes. They are not directly accessible within the procedure body.

```
CREATE or REPLACE PACKAGE gpa_pkg AS
  PROCEDURE get_gpa (student_id  IN  NUMBER,
                     gpa         OUT NUMBER)
  WITH INTERFACE
  PROCEDURE get_gpa_if
                    (student_id  INTEGER,
                     gpa         REAL,
                     SQLCODE     INTEGER
                     SQLSTATE    CHARACTER(6));
...
```

The interface procedure name specified in the WITH INTERFACE clause can be the same as the name of the procedure itself, or, as in this example, it can be different. However, the name specified in the WITH INTERFACE clause is the name that must be used when you invoke the stored procedure from your host application.

In the example, the datatypes in the WITH INTERFACE clause are SQL datatypes (INTEGER and REAL). These types are compatible with the PL/SQL datatype NUMBER.

You must include either a SQLCODE or a SQLSTATE parameter in the parameter list of the WITH INTERFACE clause. You can include both. SQLSTATE is the recommended parameter; SQLCODE is provided for compatibility with the SQL89 standard.

> **Note:** Parameters in the PL/SQL procedure specification cannot be constrained. Parameters in the WITH INTERFACE clause must be constrained where required.

The following package definition shows an example of the WITH INTERFACE clause:

```
CREATE OR REPLACE PACKAGE gpa_pkg AS

  PROCEDURE get_gpa(student_id        IN      NUMBER,
                    student_last_name IN OUT CHARACTER,
                    gpa               OUT     NUMBER)
  WITH INTERFACE
  PROCEDURE get_gpa_if
                    (student_id        INTEGER,
                     student_last_name CHARACTER(15)
```

```
                        INDICATOR sname_ind,
        sname_ind       SMALLINT,
        gpa             REAL,
        SQLSTATE        CHARACTER(6),
        SQLCODE         INTEGER);
END;
```

In the example, the student_last_name parameter is a CHARACTER, which is both a PL/SQL and a SQL datatype. In the PL/SQL part of the procedure definition, the parameter must be unconstrained, following the syntax of PL/SQL. But in the WITH INTERFACE clause, you must specify the length of the parameter.

The student_last_name parameter also takes an indicator parameter, using the syntax shown. See Appendix B for the formal syntax of the WITH INTERFACE clause.

SQL Datatypes

The SQL datatypes that you can use in the WITH INTERFACE clause are listed in Table 3–1, along with their compatible PL/SQL datatypes.

Table 3–1   SQL Datatypes

SQL Datatypes	Range or Size	SQL Meaning	Compatible PL/SQL Datatypes
CHARACTER (N) OR CHAR (N)	1 < N < 32500 bytes	String of length N (if N is omitted, N is effectively 1)	VARCHAR2(N), CHAR(N), DATE
DOUBLE PRECISION	Implicit precision 38	Approximate numeric type	NUMBER
INTEGER or INT	System specific	Integer type	NUMBER, BINARY_INTEGER
SMALLINT	System specific	Small (or short) integer type	NUMBER, BINARY_INTEGER
REAL	System-specific	Approximate numeric type	NUMBER
VARCHAR2(N)	1 < N < 32500 bytes	Character array of length N	VARCHAR2(N), CHAR(N),DATE
SQL_CURSOR		Cursor variable type	REF cursor

Note: SQL datatypes compatible with NUMBER are also compatible with types derived from NUMBER, such as REAL.

The size of integer and small integer types is system specific. For many systems, integers are 32 bits wide and small integers are 16 bits, but check your system documentation for the size on your system.

DATE Datatype

SQL*Module does not directly support the Oracle DATE datatype. You can, however, use character strings when you fetch, select, update, or insert DATE values. Oracle does the conversion between internal DATEs and character strings. See the Oracle Database Reference for more information about the DATE datatype, and conversion between DATEs and character strings.

The Default WITH INTERFACE Clause

If a package has already been defined in the database with no WITH INTERFACE clauses for the subprograms, you can still generate interface procedures to call the subprograms. The default WITH INTERFACE clause that is generated by SQL*Module when there is no WITH INTERFACE clause in the package or procedure gives you all the features of the standard WITH INTERFACE clause:

- the SQLCODE error handling parameter
- the SQLSTATE error handling parameter
- indicator parameters
- datatype mapping between PL/SQL base and derived datatypes and SQL types

Procedures

When SQL*Module generates an interface procedure with a default WITH INTERFACE clause, it generates a SQLCODE parameter in the first parameter position, and a SQLSTATE parameter in the second position. Then, for each actual parameter in the stored procedure or stored function, a parameter is generated with the appropriate mapped host language datatype. Each parameter is followed by an indicator parameter, mapped to the correct host language type from the SQL datatype SMALLINT.

Functions

If SQL*Module is generating a default WITH INTERFACE clause for functions in a package, then the WITH INTERFACE clause is generated as if the function were a procedure, with the return value and its indicator parameter as the last two parameters in the clause.

Table 3–2 shows how predefined, or base, PL/SQL datatypes are mapped to SQL datatypes, and then to host language datatypes. PL/SQL subtypes that are derived from the base types are also supported, and are mapped as indicated for the base type.

Table 3–2   Mapping PL/SQL Datatypes to SQL Datatypes

PL/SQL Datatype	Ada Language Datatype
BINARY INTEGER	SQL_STANDARD.INT
NUMBER NUMBER(P,S)	SQL_STANDARD. DOUBLE_PRECISION
RAW LONG RAW	STRING
LONG	STRING
BOOLEAN	SQL_STANDARD.INT
CHAR	SQL_STANDARD.CHAR
VARCHAR2	STRING
DATE	SQL_STANDARD.CHAR
ROWID	STRING
CURSOR	ORACLE_SQLLIB.SQL_CURSOR

> **Note:** The maximum length for the following data types are:
>
> - For STRING is 32500 bytes
> - For DATE is 2048 bytes
> - For ROWID and MLSLABEL is 256 bytes

Suppose, for example, that a procedure stored in the SCOTT schema has the parameter list

```
PROCEDURE proc1 (
        PARAM1 IN     NUMBER,
        PARAM2 IN OUT DATE,
        PARAM3    OUT DOUBLE PRECISION,
        PARAM4        CHARACTER,
        PARAM5        BINARY_INTEGER)
```

If you run the module compiler, modada, as follows:

```
modada pname=PROC1 rpc_generate=yes user=scott/tiger oname=proc1
```

then the Ada procedure specification in the generated output file proc1_.a would be created by SQL*Module as follows:

```
procedure PROC1(SQLCODE: in out sql_standard.sqlcode_type;
 sqlstate: in out sql_standard.sqlstate_type;
 PARAM1: in sql_standard.double_precision;
 PARAM1_ind: in sql_standard.smallint;
 PARAM2: in out oracle_sqllib.sql_date;
 PARAM2_ind: in out sql_standard.smallint;
 PARAM3: out sql_standard.double_precision;
 PARAM3_ind: out sql_standard.smallint;
 PARAM4: in string;
 PARAM4_ind: in sql_standard.smallint;
 PARAM5: in sql_standard.int;
 PARAM5_ind: in sql_standard.smallint);
```

Function calls are generated as procedures with the last two parameters in the generated prototype being the return parameter and the indicator variable for the return parameter. For example:

```
FUNCTION func1 (
        PARAM1 IN NUMBER) RETURN VARCHAR2
```

would have the Ada prototype:

```
procedure FUNC1(SQLCODE: in out sql_standard.sqlcode_type;
 sqlstate: in out sql_standard.sqlstate_type;
 PARAM1: in sql_standard.double_precision;
 PARAM1_ind: in sql_standard.smallint;
 mod_func_return: out string;
 mod_func_return_ind: out sql_standard.smallint) is
begin
  . . .

end FUNC1;
```

Storing Module Language Procedures

You can also use SQL*Module to create a stored package in the database from Module Language procedures. By specifying the module file in the INAME command line option (see Chapter 5, "Running SQL*Module" for details), and setting the option STORE_PACKAGE=YES, the procedures in the module file are stored in a package in the database, using the module name as the default package name. (The default name can be overridden using the PNAME option. See Chapter 5, "Running SQL*Module" for details.)

For example, the following module file:

```
MODULE        test_sp
AUTHORIZATION scott

PROCEDURE get_emp (
        :empname    CHAR(10),
        :empnumber  INTEGER,
         SQLCODE);
    SELECT ename INTO :empname
        FROM emp
        WHERE empno = :empnumber;

PROCEDURE put_emp (
        :empname    CHAR(10),
        :empnumber  INTEGER,
        :deptnumber INTEGER,
        SQLCODE);
    INSERT INTO emp (ename, empno, deptno) VALUES
                    (:empname, :empnumber, :deptnumber);
```

when stored as a package in the database would produce the following PL/SQL code for the package specification:

```
package test_sp is
procedure get_emp
    (empname out char,
    empnumber in number)
  with interface procedure get_emp
    (empname char(11),
    empnumber integer,
    sqlcode integer);
procedure put_emp
    (empname in char,
    empno in number,
    deptno in number)
  with interface procedure put_emp
    (empname char(11),
    empnumber integer,
    deptnumber integer,
    sqlcode integer);
end test_sp;
```

Note: You cannot store module procedures that contain the ALLOCATE statement, nor statements CONNECT, DISCONNECT, ENABLE THREADS, CONTEXT, nor FETCH and CLOSE statements that refer to cursor variables.

Connecting to a Database

When you write an Ada program that calls RPC interface procedures that were generated from stored procedures, you need a way to connect to a database at runtime. The steps you can take to do this are

- Write a module that contains connect and disconnect procedures. See "CONNECT Statement" on page 2-13 for the syntax of these procedures. See also the examples in the demomod sample in Chapter 6, "Demonstration Programs".

- Compile the module using SQL*Module.

Add a with clause to the host application file referencing the generated specification name.

- Compile the specification file.

- Compile the source output file.

- Link your main application.

4

Developing the Ada Application

This chapter describes the criteria that an Ada application must meet when accessing module procedures, or when calling RPC stubs generated by SQL*Module. Topics covered include

- Program Structure
- Error Handling
- Obtaining the Number of Rows Processed
- Handling Nulls
- Cursors
- Specification Files
- Calling a Procedure
- Arrays as Procedure Arguments
- Globalization Support

The sample programs in this chapter are source code listings for the Module Language procedures that are called by the sample programs in Chapter 6, "Demonstration Programs", and a set of SQL statements that create and partially populate the example tables. These sources are also available online, in the *demo* directory.

Program Structure

The developer determines the structure of an application program that uses SQL*Module. A significant advantage that you obtain from using SQL*Module is that it imposes very few special requirements or constraints on the program design, unlike some other SQL programmatic interfaces.

The code that you write is purely in the language of the host application program. There is no need for special declare sections, embedded SQL statements, and special error handling and recovery. Database operations are mostly transparent to the application program developer, being taken care of by the Module Language or PL/SQL stored procedures.

There are, however, some SQL concepts of which the host application developer must be aware

- error handling, and the use of the SQLSTATE or SQLCODE status parameter.
- the concept of *null*, and how to use indicator variables to handle it
- the concept of a cursor

Error Handling

Each Module Language procedure that is called from the host application must contain a parameter that returns status information to the application. There are two status parameters that you can use: SQLCODE and SQLSTATE. SQLCODE returns an integer value, while SQLSTATE returns a five-character string that contains an alphanumeric code.

SQLCODE is provided for compatibility with applications written to the 1989 SQL standards; new applications should use the SQLSTATE parameter.

When calling stored database procedures through an RPC stub, you include SQLCODE or SQLSTATE in the parameter list of the WITH INTERFACE clause in the procedure's package specification. See "The WITH INTERFACE Clause" on page 3-9.

SQLCODE

SQLCODE is an output parameter that can be included in a module procedure, and in the WITH INTERFACE clause in PL/SQL stored package specifications. SQLCODE returns a value that indicates whether a procedure completed successfully, completed with warnings, or did not complete due to an error.

SQLCODE returns three kinds of values:

0

Indicates that the procedure completed with no errors or warnings.

< 0

Indicates that an error occurred during execution of the procedure.

+100

Indicates that a SQL statement did not find a row on which to operate.

Negative SQLCODE values are Oracle message numbers. See the *Oracle Database Error Messages* manual for a complete list of Oracle codes and their accompanying messages. See the next section, "SQLSTATE", for mappings between Oracle error numbers and SQLSTATE values.

Obtaining Error Message Text

The procedure *error_message* in the public package *oracle_sqllib* was introduced in release 8.0. This procedure obtains the text associated with the SQLCODE of the latest error returned. The prototypes are (with and without a runtime context):

```
procedure ERROR_MESSAGE (ctx oracle_sqllib.sql_context,
                         msg_buf system.address,
                         msg_buf_len sql_standard.int);
```

and:

```
procedure ERROR_MESSAGE (msg_buf:out    system.address,
                         msg_buf_len:out sql_standard.int);
```

SQLSTATE

SQLSTATE is a five-character alphanumeric output parameter that indicates the completion status of the procedure. It is declared as SQL_STANDARD.SQLSTATE_TYPE.

SQLSTATE status codes consist of a two-character class code followed by a three-character subclass code. Aside from the class code 00 ("successful completion"), the class code denotes the category of the exception. Also, aside from the subclass code 000 ("not applicable"), the subclass code denotes a specific exception within that category. For example, the SQLSTATE value '22012' consists of class code 22 ("data exception") and subclass code 012 ("division by zero").

Each of the five characters in a SQLSTATE value is a digit (0..9) or an uppercase Latin letter (A..Z). Class codes that begin with a digit in the range 0..4 or a letter in the range A..H are reserved for the predefined conditions (those defined in the SQL92 specification). All other class codes are reserved for implementation-defined sub-conditions. All other subclass codes are reserved for implementation-defined sub-conditions. Table 4–1 shows the coding scheme.

Figure 4–1   SQLSTATE
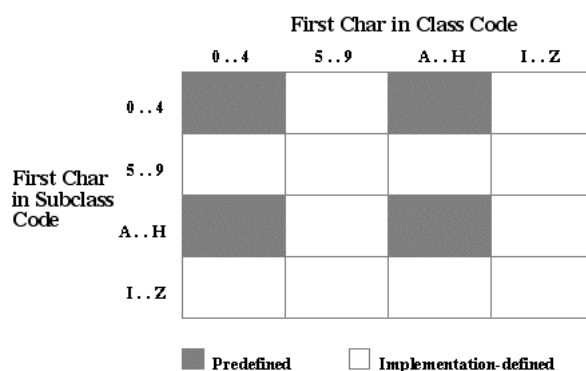
Table 4–1   Predetermined Classes

Class	Condition
02	no data
07	dynamic SQL error
08	connection exception
0A	feature not supported
21	cardinality violation
22	data exception
23	integrity constraint violation
24	invalid cursor state
25	invalid transaction state
26	invalid SQL statement name
27	triggered data change violation
28	invalid authorization specification
2A	direct SQL syntax error or access rule violation
2B	dependent privilege descriptors still exist
2C	invalid character set name
2D	invalid transaction termination
2E	invalid connection name

Table 4–1   (Cont.) Predetermined Classes

Class	Condition
33	invalid SQL descriptor name
34	invalid cursor name
35	invalid condition number
37	dynamic SQL syntax error or access rule violation
3C	ambiguous cursor name
3D	invalid catalog name
3F	invalid schema name
40	transaction rollback
42	syntax error or access rule violation
44	with check option violation
HZ	remote database access

> **Note:**   The class code HZ is reserved for conditions defined in
> International Standard ISO/IEC DIS 9579-2, *Remote Database Access.*

 Appendix D, "SQLSTATE Codes" shows how Oracle errors map to SQLSTATE status
codes. In some cases, several Oracle errors map to a status code. In other cases, no
Oracle error maps to a status code (so the last column is empty). Status codes in the
range 60000 .. 99999 are implementation-defined.

Obtaining the Number of Rows Processed

Starting with release 8.0, the function *rows_processed*, in the public package oracle_
sqllib, returns the number of rows processed by the last SQL statement.

The prototypes are:

```
function ROWS_PROCESSED return integer;
```

and

```
function ROWS_PROCESSED (ctx oracle_sqllib.sql_context) return integer;
```

where the context, *ctx*, has been allocated previously.

Handling Nulls

A database column or a SQL expression can have a value, or it can have a special
status called *null*. A null means the absence of a value. A numeric value or a special
string encoding cannot be used to indicate a null, since all allowable numeric or string
values are reserved for actual data. In a SQL*Module application, you must use an
indicator variable to test a returned value for a null, or to insert a null into a database
column.

> **Note:** The term *indicator variable* is also referred to as an *indicator parameter* when discussing a variable being passed to or retrieved from a procedure.

Indicator Variables

From the host language point of view, an indicator variable is a small integer that is passed to a procedure. In the SQL statement of the procedure, the indicator is associated with the corresponding host parameter. For example, the Module Language procedure performs a simple one-row SELECT (the host parameter in the WHERE clause is assumed to be a primary key):

```
PROCEDURE get_commission (
                    :commission   REAL,
                    :comm_ind     SMALLINT,
                    :emp_number   INTEGER,
                     SQLSTATE);
   SELECT comm INTO :commission INDICATOR :comm_ind
      FROM emp WHERE empno = :emp_number;
```

In an Ada application, you call this procedure and test for a possible null in the returned COMMISSION as follows:

```
EMPNO := 7499;
GET_COMMISSION (COMMISSION, COMM_INDICATOR, EMPNO, SQLSTATE);
if COMM_INDICATOR < 0 then
    PUT_LINE("Commission is null.");
else
    PUT("Commission is ");
    PUT(COMMISSION);
    NEW_LINE;
end if;
```

So if an indicator variable is less than zero when a procedure returns, the associated host parameter has an undefined value.

You can also associate indicator variables with input parameters, for column values that are used to insert a new row into a table, or update an existing row. If the value in the indicator variable is greater than or equal to zero, the value in the associated parameter is used as the input value. If the indicator variable is set to -1, the value in the associated parameter is ignored, and a null is inserted as the column value.

For example, the following module procedure inserts a new row into an inventory table:

```
PROCEDURE new_part (
        :part_no     INTEGER,
        :description CHAR(200),
        :bin_number  INTEGER,
        :bin_no_ind  SMALLINT,
         SQLSTATE);

   INSERT INTO inventory (part_number, description, bin_no)
      VALUES (:part_no, :description,
               :bin_number INDICATOR :bin_no_ind);
```

When you call this procedure with the parameter *bin_no_ind* set to -1, any value in the parameter *bin_number* is ignored, and a null is inserted into the BIN_NO column of the table.

If the host language parameter is a character type, and has an associated indicator variable, a returned indicator value greater than zero indicates that the returned value was truncated. The value of the indicator is the original (un-truncated) length, in bytes, of the column or expression value.

Cursors

Programs that retrieve data from a table can work in two different ways. In one case, a query might be constructed that expects either one row of data to be returned, or no row. For example, if the program performs a request such as "give me the name of the employee whose employee number is 7499", where the employee number is a primary key of the table (and hence, by definition, unique), the request either returns the name of the employee whose employee number is 7499, or returns an indication that no such employee exists in the table.

If no employee exists with that number, the query procedure returns a "no data found" indication in the SQLCODE or SQLSTATE parameter.

For Oracle to process any SQL statement, a cursor is required. However, SQL*Module *implicitly* defines a cursor for INSERT, UPDATE, and DELETE statements, as well as SELECT statements.

However for queries that can return multiple rows, an explicit cursor must be defined in the module or stored package to fetch all the rows. You can use static cursors, or cursor variables. See "Cursors" on page 4-6 for a description of cursor variables.

See the code in "Module Language Sample Program" on page 6-7 for several examples that use explicit cursors.

Specification Files

The SQL*Module compiler generates specification files. These are text files that contain declarations for the module or interface procedures that SQL*Module generates.

You must include the specification file directly in the source of your host application. The name of the specification file is the base name of the Module Language output file for SQL*Module, with a system-specific extension. These extensions are documented in "Specification File" on page 5-7.

In Ada applications, you must compile the specification file (or files) that SQL*Module generates. You then include the specification for the module procedures or stubs in each application that calls module procedures or stubs using the with context clause.

The naming of specification files is discussed in detail in Chapter 6, "Demonstration Programs".

Calling a Procedure

You call procedures generated by SQL*Module using the normal procedure call format of the host language. Procedures can only return values in parameters, including the SQLCODE and SQLSTATE parameters. The generated procedures are not functions.

Arrays as Procedure Arguments

SQL*Module supports array bind and define variables as arguments to procedures and functions:

```
PROCEDURE foo (:arrname ARRAY(n) OF type, SQLCODE);
```

where n is the size of arrname, and type is listed in "Globalization Support" on page 4-7.

For example:

```
PROCEDURE selempno (:eno ARRAY(14) of INTEGER, SQLCODE);
   SELECT empno INTO :eno FROM emp;
```

> **Note:** Host arrays are allowed in SELECT, FETCH, INSERT, UPDATE and DELETE statements only.

Restrictions:

1. Arrays may not be specified when RPC_GENERATE=yes or STORE_PACKAGE=yes. See "Stored Packages" on page 3-3 for more information. See both these command-line options in Chapter 5, "Running SQL*Module".

2. The maximum dimension of an array is 32000

3. SQL*Module does not allow multi-dimension arrays.

Globalization Support

Not all writing systems can be represented using the 7-bit or 8-bit ASCII character set. Some languages require multibyte character sets. Also, countries have different ways of punctuating numbers, and representing dates and currency symbols.

Oracle provides Globalization Support, which lets you process single-byte and multibyte character data and convert between character sets. It also lets your applications run in different language environments. With Globalization Support, number and date formats adapt automatically to the language conventions specified for a user session. Thus, Globalization Support allows users around the world to interact with Oracle in their native languages.

You control the operation of language-dependent features by specifying various Globalization Support parameters. Default values for these parameters can be set in the Oracle initialization file. The following table shows what each Globalization Support parameter specifies:

Globalization Support Parameter	Specifies ...
NLS_LANGUAGE	language-dependent conventions
NLS_TERRITORY	territory-dependent conventions
NLS_DATE_FORMAT	date format
NLS_DATE_LANGUAGE	language for day and month names
NLS_NUMERIC_CHARACTERS	decimal character and group separator
NLS_CURRENCY	local currency symbol
NLS_ISO_CURRENCY	ISO currency symbol
NLS_SORT	sort sequence

The main parameters are NLS_LANGUAGE and NLS_TERRITORY. NLS_LANGUAGE specifies the default values for language-dependent features, which include

- language for Server messages

- language for day and month names

- sort sequence

NLS_TERRITORY specifies the default values for territory-dependent features, which include

- date format

- decimal character

- group separator

- local currency symbol

- ISO currency symbol

You can control the operation of language-dependent Globalization Support features for a user session by specifying the parameter NLS_LANG as follows:

```
NLS_LANG = <language>_<territory>.<character set>
```

where *language* specifies the value of NLS_LANGUAGE for the user session, *territory* specifies the value of NLS_TERRITORY, and *character set* specifies the encoding scheme used for the terminal. An *encoding scheme* (usually called a character set or code page) is a range of numeric codes that corresponds to the set of characters a terminal can display. It also includes codes that control communication with the terminal.

You define NLS_LANG as an environment variable (or the equivalent on your system). For example, on UNIX using the C shell, you might define NLS_LANG as follows:

```
setenv NLS_LANG French_Canadian.WE8ISO8859P1
```

SQL*Module fully supports all the Globalization Support features that allow your applications to process multilingual data stored in an Oracle Database version 8 database. For example, you can run a SQL*Module-derived client application that interacts with a remote server, where the client and the server are using different

character sets, possibly with a different number of bytes each character. In these contexts, remember that specification of the lengths of string types, such as the SQL datatype CHARACTER(N), is always specified in *bytes*, not characters.

You can even pass Globalization Support parameters to the TO_CHAR, TO_DATE, and TO_NUMBER functions. For more information about Globalization Support, see the *Oracle Database Advanced Application Developer's Guide*.

5

Running SQL*Module

This chapter describes

- SQL*Module Input and Output

- Invoking SQL*Module

- Case Sensitivity in Program Names, Option Names, and Values

- How to Specify Command-Line Options

- Input Files

- Output Files

- Command-Line Options

- Compiling and Linking

SQL*Module Input and Output

This section reviews the different ways that you can use the SQL*Module compiler. This material was discussed in detail in Chapter 2, "Module Language" and Chapter 3, "Accessing Stored Procedures"; here it is presented in terms of the ways that you run the compiler, using the command-line options to get different SQL*Module functionality.

Input sources

Input to the compiler can come from two sources:

- module files written according to the SQL standard Module Language specifications, as described in Chapter 2, "Module Language" of this Guide

- stored packages and procedures in an Oracle database (see Chapter 3, "Accessing Stored Procedures")

You use a standard text editor to create module files, just as you would create a host language application.

Stored procedures can be standalone procedures, or they can be encapsulated in a stored package. You normally create PL/SQL code for stored packages and procedures using a text editor, and then store it in a database using an Oracle tool such as SQL*Plus. You can also use SQL*Module to encapsulate Module Language procedures in a package, and store them in the database.

Output Files

The *output source file* is always the host language code file that SQL*Module generates from the input source. There are also other output files, such as the *listing file* and *specification file*. You can run SQL*Module and generate *no* output source file, for example if you just want to store procedures in the database from a Module Language input file, or you just want to generate a listing file.

You compile output source files using the host language compiler, and link the resulting object files together with the host application's object files to produce the executable program. See the section "Compiling and Linking" on page 5-17 for more information about handling output files.

> **Note:** While many of the examples in this chapter assume, for simplicity, that the input and output files are in the same directory, this does not have to be the case. Input and output files can be in separate directories, and you can use the various NAME options to specify the source of input, or the destination of output.

Determining the Input Source

There are three sources of input for SQL*Module, and four ways to determine the input:

1. When compiling a module written in Module Language, the source is the Module Language code file.

2. When generating RPC stubs from stored procedures, there is no input file. The source of the input is the stored package in the database.

3. When creating a stored package in the database from a Module Language module file, the source is the Module Language file.

4. You can combine methods 1 and 2 in one invocation of SQL*Module. A package in the database is created from the Module Language module file, and an output file that contains RPC stubs to call the database package procedures is produced.

Methods 1 and 4 are the most common ways to use SQL*Module. Method 1 is described in Chapter 2, "Module Language" of this Guide, method 2 in Chapter 3, "Accessing Stored Procedures". Methods 3 and 4 are much more specialized, and are described in Chapter 3, "Accessing Stored Procedures".

STORE_PACKAGE

Determines whether SQL*Module should store a package in the database.

RPC_GENERATE

Determines whether an interface procedure output file is produced. When you specify the option RPC_GENERATE as YES, the option PNAME specifies the name of the package in the database that provides the input source.

Table 5–1 shows the how the command-line option values for STORE_PACKAGE and RPC_GENERATE, together with the values for INAME and PNAME, determine the input source.

Table 5–1   Datatypes

Input Source	Options			
	STORE_ PACKAGE	RPC_ GENERATE	INAME	PNAME
(1) Module source file	=NO	=NO	Module file name	N/A
(2) Procedure already stored in database	=NO	=YES	N/A	Stored package or procedure name
(3) Module file to create SPs in database	=YES	=NO	Module file name	Database package name (if not specified, becomes same as module filename)
(4) Store module procedures, then do (2)	=YES	=YES	Module file name	Database package name (if not specified, becomes same as module filename)

See the section "Command-Line Options" on page 5-8 for a detailed description of these options. See the section "Compiling and Linking" on page 5-17, for examples that show you how you can use these options. For an explanation of the default file naming conventions, see the sections "Input Files" on page 5-6 and "Output Files" on page 5-6.

Invoking SQL*Module

You can run the SQL*Module compiler interactively from the operating system command line, from a command in a batch file, or for some operating systems, a *makefile*. The way you invoke the compiler can be system dependent. See your system-specific Oracle documentation to find out the location on your system of the compiler and associated files, such as configuration files and the SQL runtime library.

Running the Compiler

The name of the SQL*Module compiler itself is *modada* for Ada. The SQL*Module compiler can be invoked from the operating system command line as follows:

```
modada  <option=value> ...
```

where <option=value> is a command-line argument. For example, the command

```
modada iname=my_test1.mad oname=my_test1_mod.a userid=modtest
```

compiles the module file *my_test1.mad* to produce an output file called *my_test1_mod.a*. The username is *modtest*. In this example no password was provided on the command line. As a result, SQL*Module prompts you for one when it starts. SQL*Module requires a valid username and password to compile a Module Language file. The objects referenced in the cursors and procedures in the Module file must be available in the schema named (MODTEST in this example) when you run SQL*Module.

When you use SQL*Module to generate interface procedure files that call stored procedures in the database, you must specify the same USERID as the schema that owns the stored procedures.

Case Sensitivity in Program Names, Option Names, and Values

For operating systems that are case sensitive, such as UNIX, the names of the executables are normally in lowercase. For all systems, the names of the options and their values are not case sensitive. In this Guide, the option name is in uppercase, and the value is in lower case. However, when the option value is a filename, and your operating system is case-sensitive, you must enter the filename using the correct combination of upper and lowercase

Listing Options and Default Values

If you provide no command-line arguments, or the only argument is '?', the compiler prints a list of all the options available, with their current default values. For example, the command

```
modada ?
```

runs the SQL*Module compiler for Ada and lists each option with its default value. See "Default Values" on page 5-5 for information on what determines the defaults. (Be sure to escape the '?' using '\' if you are running on a UNIX system and you are using the C shell.)

If you just want to see the default value for a single option, you can issue the command:

```
modada <OPTION>=?
```

For example, the command

```
modada OUTPUT=?
```

shows the default values for the OUTPUT option for the SQL*Module compiler for Ada.

```
modada
```

produces a short help display.

A complete description of each option is given later in this chapter.

How to Specify Command-Line Options

The value of an option is a string literal, which can represent text or numeric values. For example, for the option

```
INAME=my_test
```

the value is a string literal that specifies a filename. But for the option

```
MAXLITERAL=400
```

the value is numeric.

Some options take Boolean values, and these may be represented with the strings "yes" or "no", or "true" or "false" (in upper or lowercase). For example:

```
...  STORE_PACKAGE=YES
```

is equivalent to

```
...  STORE_PACKAGE=true
```

both of which mean that the results of the compilation should be stored as a package in the database.

The option value is always separated from the option name by an equals sign, with *no whitespace* between the name or the value and the equals sign.

Value Lists

Some options can take multiple values. Multiple option values are specified in a list. The list is a comma-delimited list of values with surrounding parentheses. Do *not* put any whitespace in the list. The following option specifies that SQL*Module should generate source code and specification output files, but not listing files:

```
...  OUTPUT=(CODE,SPECIFICATION)
```

A value list completely supersedes the value list specified by a previous default or option value list. For example, if the system configuration file contains the line

```
OUTPUT=(CODE,SPECIFICATION,LIST)
```

and there is no user configuration file, and the command line contains the option

```
...  OUTPUT=(CODE,LIST)
```

then the value of OUTPUT is (CODE,LIST). See the section "Configuration Files" on page 5-6 for how default values are determined.

If a list-valued option is specified with a single value, that is not in parentheses, the single value is *added* to the current default list. For example, if the system configuration file contains the line

```
OUTPUT=(CODE,SPECIFICATION)
```
there is no user configuration file that has an OUTPUT= option, and the command line contains the option

```
...  OUTPUT=LIST
```

then "LIST" is appended to the default list, so the value of OUTPUT is (CODE,SPECIFICATION,LIST).

Note: If NONE is a member of the OUTPUT list, then nothing would be generated, regardless of other entries in the list.

Default Values

Most of the options have default values. Three things determine the default value:

- values built into the SQL*Module compiler
- values set in the *system configuration file*
- values set in a *user configuration file*

For example, the option MAXLITERAL specifies the maximum length of strings generated by SQL*Module. The built-in SQL*Module default value for this option is 255 bytes. However, if MAXLITERAL=512 is specified in the system configuration file, the default now becomes 512. The user configuration file could set it to yet another value, which then overrides the system configuration value. Finally, if this option is set on the command line, that value will take precedence over the SQL*Module default, the system configuration file specification, and the user configuration file specification. See "Configuration Files" for more information about these files.

Some options, such as USERID, do not have a built-in default value. The built-in default values for options that have them are listed in the section "Command-Line Options" on page 5-8.

Configuration Files

A configuration file is a text file that contains SQL*Module options. Each record or line in the file contains one option, with its associated value or list of values. For example, a configuration file might contain the lines

```
BINDING=LATE
USERID=MODTEST
```

to set defaults for the BINDING and USERID options.

> **Note:** You cannot put comments in a configuration file; there is no character or character combination that lets you comment out a line.

There is one systemwide configuration file associated with each system. The system configuration file is usually maintained by the project or group leader, or the database administrator. The location of this file is system specific. For more information, see your project leader, or your system-specific Oracle documentation.

If there is no system configuration file, the compiler prints a warning message, but compilation continues normally.

In addition, each SQL*Module user can have one or more user (or local) configuration files. To activate the user configuration file, its name and path must be specified using the CONFIG= command-line option. See "Command-Line Options" on page 5-8. The user configuration file is optional.

The CONFIG= option never specifies the system configuration file. The location of the system configuration file is built into the SQL*Module compiler, and can vary from system to system.

Input Files

A SQL*Module input file is a text file containing Module Language statements. You specify the input filename using the INAME= command-line option.

Input files have default file *extensions*, also referred to as *filetypes* in some operating systems. However, not all operating systems support file extensions. If your system does not support file extensions, the last few characters of the filename might serve as the extension. Refer to your operating system documentation and to your system-specific Oracle documentation for more information about filenames and file extensions.

If you do not specify an extension for the module input file, and your operating system uses file extensions, the compiler assumes a default extension, *.mad*.

Output Files

SQL*Module can generate four types of output files:

- a source code file
- a specification (or header) file

- a listing file

- a PL/SQL source file for a stored procedure or a package

Source code files contain generated code in the host language. *modada* generates Ada code. Specification or header files contain declarations for the procedures in the code files

Source Code Output File

This file contains the host language code produced by the compiler. It is a source file in the host language, and you must compile it using your host language compiler to produce an object module. The object modules are in turn linked with the application's object modules and the SQL runtime library to form the executable program.

> **Note:** Oracle recommends that you name output files explicitly, either in a configuration file or on the command line.

Default File Names for Ada

If you do not specify an output code filename when you run *modada*, the output code filename defaults to a system-specific name. For example, on Sun workstations running the Solaris 1.0 Sun Ada compiler, the command

```
modada iname=my_test1.mad
```

generates an output code file named *my_test1.a*. On other platforms, a different name might be generated. See your system-specific Oracle documentation for complete information.

Specification File

By default, *modada* generates a specification or header file. The specification file contains declarations for the procedures in the generated output file.

Default Specification Filenames for Ada

The default specification filename is the name of the input file, or the package name, followed by a system-dependent appendix, followed by a system-dependent file extension. For example, on a Sun workstation running Solaris 1.0, the command

```
modada iname=my_test1.mad
```

generates a default specification output file with the name *my_test1s.a*. This is the value of *iname* minus the extension, with "s" appended.

On other platforms, the filename appendix and the filename extension might be different. See your system-specific Oracle documentation for complete information.

See Chapter 6, "Demonstration Programs" for language-dependent information about the content of specification files.

Listing File

If OUTPUT=LIST, SQL*Module produces a listing of the Module Language source code, with errors, if any, flagged. Error codes and messages are interspersed in the list file, at the point where the SQL*Module parser first detected the error. The line length in the listing file defaults to 255 characters. If no end-of-line character is received

before 255 characters are received, a system-specific end-of-line character or character sequence is output.

PL/SQL Source Files

When you are generating interface procedure files from a stored package or procedure, and you specify the option OUTPUT=PACKAGE, SQL*Module generates PL/SQL source code output files. If the output is from a package, two files are generated. One file has the default file extension *.pks*, and contains the package specification code. The second file has the default extension *.pkb*, and contains the package body code. See the *Oracle Database PL/SQL Language Reference* for more information on package specifications and package bodies.

Avoid Default Output Filenames

Use the ONAME and SNAME options to generate nondefault output filenames. They are described later.

Oracle strongly recommends that you use these options, rather than letting the output filenames be generated by default.

Command-Line Options

When an option is entered on the command line, its value overrides SQL*Module defaults, any values specified in a configuration file, or values specified in a module file (for example, the AUTHORIZATION clause). The order of precedence is

- command-line options
- statements in the module file preamble
- user configuration file options
- system configuration file options
- default options built into the compiler

The format of a command-line option is:

```
OPTION_NAME=VALUE
```

There should be no whitespace around the equals sign. For example:

```
modada INAME=my_app3_mod ONAME=my_app3_mod SNAME=my_app3_pkg
```

compiles the input file *my_app3_mod.mad* to produce an output file named *my_app3_mod.a*, and a specification file named *my_app3_mod_pkgs.a*.

> **Note:** The actual filename extensions are system specific. See your system-specific Oracle documentation for more information.

If the option can take a list of values and more than one value is being supplied, a comma-delimited list of values is placed inside parentheses. For example:

```
OUTPUT=(CODE,SPECIFICATION)
```

There should be no whitespace anywhere in the list.

The names as well as arguments of the command-line options can be abbreviated. Instead of

```
OUTPUT=SPECIFICATION
```

you could enter

```
OUT=SPEC
```

or even

```
OU=SP
```

since neither "OU", "SPEC", nor "SP" is ambiguous. Filenames and package names cannot be abbreviated.

The command-line options, together with their default values, are listed in Table 5–2, and are described in the remainder of this chapter.

Table 5–2   The Command-line options

Option Name	Option Purpose	Values
AUTO_CONNECT	Connect on first SQL statement if not already connected	YES | NO
BINDING	Early or late binding?	EARLY | LATE
CONFIG	Name of a user configuration file	<filename>
ERRORS	Destination of error messages	YES | NO
FIPS	Turns on FIPS flagger	YES | NO
INAME	Name of input file	<filename>
LNAME	Name of listing file	<filename>
LTYPE	Kind of listing file	NONE | SHORT | LONG
MAPPING	Resolves overloaded procedure names for the default WITH INTERFACE PROCEDURE clause	( ) | OVERLOAD
MAXLITERAL	Maximum length of string literal in generated host language code	10..1024
ONAME	Name of source code output file	<filename>
OUTPUT	Kinds of output files generated	One of, or list of two or more of (NONE | CODE | SPECIFICATION | LIST | PACKAGE
PNAME	Name of package in the database	<package_name>
RPC_GENERATE	Generate stubs from stored package or procedure?	YES | NO

Table 5–2   (Cont.) The Command-line options

Option Name	Option Purpose	Values
SELECT_ERROR	Should a query returning more than one row generate a runtime error?	YES | NO
SNAME	Name of specification output file	<filename>
SQLCHECK	Kind of compile-time checking done	NONE | SYNTAX | SEMANTICS
STORE_PACKAGE	Store module as a package in the database	YES | NO
USERID	Username and password	<string>

AUTO_CONNECT

Values

{YES | NO}

Default Value

NO

Meaning

If AUTO_CONNECT=YES, and you are not already connected to a database, when SQLLIB processes the first executable SQL statement, it attempts to connect using the userid

```
OPS$<username>
```

where username is your current operating system user or task name and nullusername is a valid Oracle userid.

When AUTO_CONNECT=NO, you must use the CONNECT statement to connect to Oracle.

Can be entered only on the command line or in a configuration file.

BINDING

Values

{EARLY | LATE}

Default Value

EARLY

Meaning

The BINDING option is used when generating interface procedure files, that is, when RPC_GENERATE=YES. Early binding means that a time stamp is derived from the

time of compilation of the stored procedure, and the time stamp is saved in the interface procedures file.

When a stored procedure is called through a stub (specified in the interface procedures file), if the current time stamp on the procedure in the database is later than that specified in the stub, the message "time stamp of <stored procedure name> has been changed" (ORA-04062) is returned.

The stored package must have WITH INTERFACE clauses specified for each procedure when RPC_GENERATE=YES, regardless of whether you choose early or late binding using the BINDING option. See the section "Early and Late Binding" on page 3-5 for more information.

CONFIG

Values
<filename>

Default Value
None.

Meaning
Specifies the name of a user configuration file that contains options. The user configuration file is a text file. Each option in the file must be on a separate line (record).

ERRORS

Values
{YES | NO}

Default Value
YES

Meaning
Specifies the destination for error message output. If ERRORS=YES, the output is both to the terminal and to the listing (*.lis*) file. If ERRORS=NO, error messages are sent only to the listing file.

FIPS

Values
{YES | NO}

Default Value
NO

Meaning
Specifies whether instances of non-compliance with the ANSI/ISO SQL standards will be flagged at compile time. If FIPS=YES, Oracle extensions to standard Module

Language and standard SQL, as well as use of standard constructs in ways that violate the SQL standard format or syntax rules, are flagged by the FIPS flagger.

INAME

Values
<filename>

Default Value
None.

Meaning
Specifies the name of the input file. If the specified filename does not contain an extension, the compiler supplies the default extension for the host language. Only one input file is allowed. If more than one INAME option is specified, the last one prevails, and the earlier ones are ignored.

If STORE_PACKAGE=NO and the PNAME option is specified, the INAME option cannot be specified. In this case, there is no input file, since the input comes from the stored package. If INAME is specified under these circumstances, SQL*Module generates a warning message and continues, if possible.

LNAME

Values
<filename>

Default Value
The base name of the listing file first defaults to the base name of INAME or, if INAME is not specified, it defaults to the name of the package specified in the PNAME option. The default file extension is *.lis*.

Meaning
Specifies the name of the listing file. This option is valid only if the LTYPE option is *not* NONE.

LTYPE

Values
{NONE | SHORT | LONG}

Default Value
LONG

Meaning
Specifies the listing type. The OUTPUT option list must contain the VALUE LIST, otherwise this option has no effect.

If the LTYPE value is NONE, no list file is generated, regardless of the setting of the OUTPUT option. If the LTYPE value is SHORT, the list file contains no code, only error messages. LTYPE=LONG generates a complete listing file, with errors and code.

> **Note:** When INAME is specified, the listing file shows Module Language code, not the generated host language code. When compiling an interface procedure, the listing output contains only error messages, regardless of the LTYPE specification. See the OUTPUT option for more information on how to generate PL/SQL output source.

MAPPING

Values
( ) | OVERLOAD

where ( ) indicates an empty string.

Default Value
Empty string.

Meaning
The MAPPING option is used when generating prototypes for the default WITH INTERFACE PROCEDURE clause. See "The Default WITH INTERFACE Clause" on page 3-12 for more information.

When MAPPING=OVERLOAD, SQL*Module resolves overloaded stored procedure and function names when generating stubs. It does this by prefixing *MODn_* to the second and subsequent procedure names, where *n* starts with 2, and increments by 1 until all stubs for all overloaded procedures of that name have been resolved.

MAXLITERAL

Values
Numeric literal, range 10 to 1024 bytes

Default Value
255 bytes

Meaning
Specifies the maximum length of string literals generated by the SQL*Module compiler, so that host language compiler limits are not exceeded. For example, if your system's compiler cannot handle string literals longer than 512 bytes, specify MAXLITERAL=512 in the system configuration file.

ONAME

Values
<filename>

Default Value
The base name of the output file first defaults to the base name of INAME. If INAME is not specified, then ONAME defaults to the name of the package specified in the

PNAME option, if present. The default file extension is system dependent, but is generally .a. The default output directory is the current directory.

Meaning

Specifies the name of the code output file. Whether an output file is actually generated depends on the values of the OUTPUT option. The OUTPUT list must contain the value CODE.

OUTPUT

Values

Any one or more of CODE, LIST, NONE, PACKAGE, SPECIFICATION

Default Values

CODE, SPECIFICATION

Meaning

Specifies what output files SQL*Module generates. The values are

CODE   An interface procedures file is generated.

LIST   A listing file is generated. See the LNAME and LTYPE options for more information.

NONE   No files are generated. This option is used to do syntactic and semantic checking of the input file, as error output is always generated.

PACKAGE   PL/SQL source files are generated. These files contain the PL/SQL package generated by SQL*Module. The default base filename is the same as the name specified in either the INAME or the PNAME option. If both are specified, the default is taken from INAME.

The default extensions are *.pks* (package specification) and *.pkb* (package body).

SPECIFICATION   A specification file containing procedure declarations is generated. The filename extension is language specific. See "Output Files" on page 5-2 for more information.

Note: If the value NONE is included in the list, then no output of any kind is generated, regardless of any other values that might be in the list.

PNAME

Values

Name of a stored package or a standalone stored procedure in the Oracle database, or the name to be given to a stored package to be created in the database when STORE_PACKAGE=YES.

Default Value

For Output (when RPC_GENERATE=YES)   There is no default value. You must specify the name of a package in the database. However, you can specify a complete path name, including an extension, to serve as a default for ONAME. In this case, the

directory hierarchy and the filename extension are ignored, and the basename is taken as the package name for database lookup.

For Input (when STORE_PACKAGE=YES)   The default value is the module name in the MODULE clause of the input file. If there is no module name, the default is taken from the INAME value.

Meaning

Specifies the name of the package stored in the database (if STORE_PACKAGE=NO), or the name of a package to be created by SQL*Module (if STORE_PACKAGE=YES). The name must be a valid database object name.

RPC_GENERATE

Values

{YES | NO}

Default Value

NO

Meaning

Specifies whether SQL*Module should produce an interface procedures file so that a host program can call stored procedures. You can use this option with STORE_PACKAGE=NO and PNAME=<package_name> to generate interface procedures for standalone or packaged procedures that are already stored in the database. You can also use this option with INAME=<filename> and STORE_PACKAGE=YES to store procedures in a module file in the database, *and* generate an interface procedures file to access them.

SELECT_ERROR

Values

 {YES | NO}

Default Value

 YES

Meaning

Specifies whether an error is generated at runtime when a SELECT or FETCH statement returns more than one row.

SNAME

Values

<filename>

Default Value

The base name of the input file, if specified, plus the appropriate extension for a specification file for the host language. For Ada, a system-specific filename addition

and extension is used, such as *ora_dcl* for VAX/OPEN VMS Ada, or **s.a* for Verdix Ada.

Meaning

Specifies the name of the specification or header file. If INAME is not specified, SNAME must be specified to get a specification file. The file is not generated if the OUTPUT option does not include SPECIFICATION in its list of values.

STORE_PACKAGE

Values

{YES | NO}

Default Value

NO

Meaning

If STORE_PACKAGE=YES, SQL*Module compiles the module file specified in the mandatory INAME option, and stores the packaged procedures in the database schema specified by the USERID option. The name of the package is specified by the PNAME option.

If you do not specify a PNAME option, the default package name becomes the name of the module, as specified in the MODULE clause of the module file. If neither the PNAME option nor the MODULE clause is specified, the package name is the base name (omitting any path specification or file extension) of the input file specified in the INAME option.

> **Note:** When STORE_PACKAGE=YES, SQL*Module performs a CREATE OR REPLACE PACKAGE statement. This statement overwrites, without any warning any package of that name in the schema.

SQLCHECK

Values

{NONE | SYNTAX | SEMANTICS}

Default Value

SEMANTICS

Meaning

Determines the way SQL*Module processes the input file when INAME is specified. This option has no meaning if there is no input file.

NONE SQL*Module processes the command line, issues any error messages produced by configuration file or command-line options, then exits without compiling any input and does not produce any output files.

SYNTAX SQL*Module compiles the input file specified in the INAME option, using its own SQL parser. Errors detected are flagged, but no source code, specification, or listing output files are produced.

SEMANTICS The input file is compiled on the server side, all syntactic and semantic errors are flagged, and all appropriate output files are generated.

USERID

Values

<string>

Default Value

None

Meaning

Specifies an Oracle username and, optionally, a password and a database to connect to. The syntax of this option is

```
USERID=USER_NAME[/PASSWORD][@DATABASE_NAME]
```

SQL*Module must be able to connect to a server when compiling an input file, to parse the SQL and PL/SQL statements, do syntactic and semantic checking, and to store packages if required. If the password is omitted, SQL*Module prompts for one. If a database is not specified, the default (local) database for the user is used.

If you do not specify the USERID option, the default becomes the user name (and possibly the password) specified in the AUTHORIZATION clause of the Module Language input file, or the USERID value specified in a configuration file.

Note: SQL*Module always prompts for a password if one has not been supplied in a configuration file, in an AUTHORIZATION clause in the module file, or on the command line. So, there is no need to hard code passwords into text files.

Compiling and Linking

To produce an executable program, you must compile source code output files that SQL*Module generates, then link these together with the compiled object files of any sources that call modules or interface procedures, with SQLLIB, and with other Oracle libraries. The details are necessarily both system and language dependent. The tables in the next three sections show a few examples.

An Example (Module Language)

There is a Module Language file to be compiled. No stored database packages are involved. The steps to take are shown in Table 5–3.

Note: This example is specific to VAX/OPEN VMS. For other Ada implementations, using a linker for all Ada files might be required.

Table 5–3   Development Scenario

Step	File Name	How Developed	Action to Take
1	tst_app_drv.ada	by Ada developer	compile into Ada library using host Ada compiler
2	tst_app_mod.mad	by SQL developer	compile using SQL*Module
3	tst_app_mod.ora_dd	generated by SQL*module in Step 2	compile into Ada library using host Ada compiler
4	tst_app_mod.ada	generated by module from Step 2	compile into Ada library using host Ada compiler; make sure to with this package in *tst_app_drv.ada*
5	tst_app_drv.o	extracted from Adalib	link (with SQLLIB)
6	tst_app_drv	linked from step 5	run and test

6

Demonstration Programs

This chapter provides information about using SQL*Module host applications written in Ada. This chapter also includes sample programs that demonstrate how you can use SQL*Module with an Ada application.

Topics covered are:

- The SQL_STANDARD Package
- Sample Applications

The SQL_STANDARD Package

You must use the datatypes defined in the supplied SQL_STANDARD package. The SQL_STANDARD package defines the packages, Ada bindings to the SQL datatypes, and the subtypes that are used for SQL*Module with Ada. You must compile the supplied SQL_STANDARD package into your Ada library, and with this package in each program unit that calls procedures generated from Module Language source, or that calls interface procedures.

The SQL_STANDARD package is system specific. See your system-specific Oracle documentation for the location of this file on your system.

SQLCODE

The standard type of the SQLCODE parameter for Ada is SQL_STANDARD.SQLCODE_TYPE.

SQLSTATE

The standard type of the SQLSTATE parameter for Ada is SQL_STANDARD.SQLSTATE_TYPE. It is a five-character string.

Sample Programs

The Module Language sample programs are based on an example database for a small college. This section demonstrates the tables that are used in the application, and a module that contains cursors and procedures that query and update the tables.

The database contains tables that maintain records about

- students
- courses
- classes (instances of courses)

■ enrollment in classes

■ instructors

■ departments

The SQL statements are used to create the tables used in the demonstration application. You can create the sample database, and fill it with some preliminary data, by using SQL*Plus or SQL*DBA to execute these scripts.

These scripts, and all other sample code files, are shipped with SQL*Module. They are in the demo directory on your system.

Sample Tables

The tables and sequence number generators are created by the MKTABLES.SQL script. At the end of this script, five other scripts are called to partially populate the tables. These five scripts are listed following MKTABLES.SQL.

MKTABLES.SQL

```
REM Create all tables for the sample college database application.

REM Drop existing tables
REM Remove REMs next 6 lines when running under SQL*Plus

REM CLEAR SCREEN
REM Prompt WARNING!! About to re-create the SQL*Module example tables.
REM Prompt All previously entered data will be lost.
REM Prompt If you really want to do this, type ENTER or Return.
REM Prompt  Else, type your CANCEL (INTR) character to exit
REM Pause   this script now.

REM Prompt Dropping tables...

DROP TABLE students     CASCADE CONSTRAINTS;
DROP TABLE instructors  CASCADE CONSTRAINTS;
DROP TABLE courses      CASCADE CONSTRAINTS;
DROP TABLE classes      CASCADE CONSTRAINTS;
DROP TABLE enrollment   CASCADE CONSTRAINTS;
DROP TABLE departments  CASCADE CONSTRAINTS;

DROP SEQUENCE student_id_seq;
DROP SEQUENCE instructor_id_seq;
DROP SEQUENCE class_number_seq;
DROP SEQUENCE enrollment_seq;

CREATE SEQUENCE student_id_seq        START WITH 1000;
CREATE SEQUENCE instructor_id_seq     START WITH 100000;
CREATE SEQUENCE class_number_seq      START WITH 100;
CREATE SEQUENCE enrollment_seq        START WITH 100;

REM Prompt Creating tables...

CREATE TABLE departments  (name        VARCHAR2(16) NOT NULL,
                           id          NUMBER(6) PRIMARY KEY,
                           location    NUMBER(4),
                           chairperson NUMBER(6),
                           budget      NUMBER(9,2)
                           );
CREATE TABLE instructors  (last_name   VARCHAR2(15) NOT NULL,
```

```
                          first_name    VARCHAR2(15) NOT NULL,
                          mi            VARCHAR2(3),
                          id            NUMBER(6) PRIMARY KEY,
                          hire_date     DATE,
                          dept          NUMBER(6)
                          NOT NULL REFERENCES departments(id),
                          salary        NUMBER(9,2),
                          rank          VARCHAR2(20)
                          );

CREATE TABLE students     (last_name    VARCHAR2(15) NOT NULL,
                          first_name    VARCHAR2(15) NOT NULL,
                          mi            VARCHAR2(3),
                          id            NUMBER(6) PRIMARY KEY,
                          status        VARCHAR2(5) NOT NULL,
                          date_of_birth DATE,
                          matric_date   DATE,
                          grad_date     DATE,
                          major         NUMBER(6)
                          REFERENCES departments(id),
                          advisor_id    NUMBER(6)
                          REFERENCES instructors(id)
                          );

CREATE TABLE courses      (dept          NUMBER(6)
                              NOT NULL REFERENCES departments(id),
                          id            NUMBER(6),
                          name          VARCHAR2(38) NOT NULL
                          );

CREATE TABLE classes      (class_number   NUMBER(6) PRIMARY KEY,
                          course_number   NUMBER(6) NOT NULL,
                          dept            NUMBER(6) NOT NULL,
                          max_enrollment  NUMBER(4) NOT NULL,
                          building_number NUMBER(4),
                          room_number     NUMBER(5),
                          instructor      NUMBER(6),
                          quarter         NUMBER(1),
                          year            NUMBER(4)
                          );

CREATE TABLE enrollment   (e_sn         NUMBER(6) PRIMARY KEY,
                          class_no      NUMBER(6) NOT NULL,
                          student_id    NUMBER(6) NOT NULL,
                          grade         NUMBER(3,2),
                          comments      VARCHAR2(255)
                          );

REM Prompt INSERTing sample data in tables...

@@departmt.sql
@@instrucs.sql
@@students.sql
@@courses.sql
@@enrolmnt.sql
```

DEPARTMT.SQL

```
DELETE FROM departments;
```

```
INSERT INTO departments VALUES ('BIOLOGY', 100, 2510, null,100000);

INSERT INTO departments VALUES ('CHEMISTRY', 110, 2510, null, 50000);

INSERT INTO departments VALUES ('COMPUTER SCIENCE', 120, 2530, null, 110000);

INSERT INTO departments VALUES ('ELECTRIC. ENG.', 130, 2530, null, 145000);

INSERT INTO departments VALUES ('FINE ARTS', 140, 2520, null, 10000);

INSERT INTO departments VALUES ('HISTORY', 150, 2520, null, 20000);

INSERT INTO departments VALUES ('MATHEMATICS', 160, 2580, null, 5000);

INSERT INTO departments VALUES ('MECH. ENG.', 170, 2520, null, 100000);

INSERT INTO departments VALUES ('PHYSICS', 180, 2560, null, 300000);
```

INSTRUCS.SQL

```
DELETE FROM instructors;

REM Add some faculty to the college

INSERT INTO instructors VALUES ('Webster', 'Milo', 'B', 9000,
        '01-SEP-49', 140, 40000, 'PROFESSOR');

INSERT INTO instructors VALUES ('Crown', 'Edgar', 'G', 9001,
        '03-SEP-70', 150, 35000, 'PROFESSOR');

INSERT INTO instructors VALUES ('Golighty', 'Claire', 'M', 9002,
        '24-AUG-82', 120, 33000, 'ASSISTANT PROFESSOR');

INSERT INTO instructors VALUES ('Winterby', 'Hugh', '', 9003,
        '10-SEP-82', 120, 43000, 'PROFESSOR');

INSERT INTO instructors VALUES ('Whipplethorpe', 'Francis', 'X',
        9004, '01-SEP-78', 170, 50000, 'PROFESSOR');

INSERT INTO instructors VALUES ('Shillingsworth', 'Susan', 'G',
        9005, '22-AUG-87', 160, 65000, 'PROFESSOR');

INSERT INTO instructors VALUES ('Herringbone', 'Leo', 'R', 9006,
        '02-JAN-81', 110, 40000, 'ASSOCIATE PROFESSOR');

INSERT INTO instructors VALUES ('Willowbough', 'George', 'T',
        9007, '04-SEP-86', 180, 37000, 'ASSOCIATE PROFESSOR');

INSERT INTO instructors VALUES ('Higham', 'Earnest', 'V', 9008,
        '10-JUN-76', 100, 55000, 'PROFESSOR');
```

STUDENTS.SQL

```
DELETE FROM students;

INSERT INTO students VALUES ('Brahms', 'Susan', 'F',
        student_id_seq.nextval, 'FT', '10-JUN-75', sysdate, null, null, null);

INSERT INTO students VALUES ('Hiroki', 'Minoru', '',
        student_id_seq.nextval, 'FT', '12-AUG-71', sysdate, null, null, null);
```

```
INSERT INTO students VALUES ('Hillyard', 'James', 'T',
        student_id_seq.nextval, 'FT', '11-SEP-74', sysdate, null, null, null);

INSERT INTO students VALUES ('Kaplan', 'David', 'J',
        student_id_seq.nextval, 'FT', '02-MAR-74', sysdate, null, null, null);

INSERT INTO students VALUES ('Jones', 'Roland', 'M',
        student_id_seq.nextval, 'FT', '23-JAN-75', sysdate, null, null, null);

INSERT INTO students VALUES ('Rubin', 'Naomi', 'R',
        student_id_seq.nextval, 'PT', '23-FEB-54', sysdate, null, null, null);

INSERT INTO students VALUES ('Gryphon', 'Melissa', 'E',
        student_id_seq.nextval, 'FT', '08-JUL-75', sysdate, null, null, null);

INSERT INTO students VALUES ('Chen', 'Michael', 'T',
        student_id_seq.nextval, 'FT', '22-OCT-72', sysdate, null, null, null);
```

COURSES.SQL

```
DELETE FROM courses;

REM Add a few courses for demo purposes

--  HISTORY
INSERT INTO courses VALUES (150, 101,
        'INTRODUCTION TO VENUSIAN CIVILIZATION');

INSERT INTO courses VALUES (150, 236,
        'EARLY MEDIEVAL HISTORIOGRAPHY');

INSERT INTO courses VALUES (150, 237,
     'MIDDLE MEDIEVAL HISTORIOGRAPHY');

INSERT INTO courses VALUES (150, 238,
     'LATE MEDIEVAL HISTORIOGRAPHY');

--  MATHEMATICS
INSERT INTO courses VALUES (160, 101, 'ANALYSIS I');

INSERT INTO courses VALUES (160, 102, 'ANALYSIS II');

INSERT INTO courses VALUES (160, 523, 'ADVANCED NUMBER THEORY');

INSERT INTO courses VALUES (160, 352, 'TOPOLOGY I');

--  COMPUTER SCIENCE
INSERT INTO courses VALUES (120, 210, 'COMPUTER NETWORKS I');

INSERT INTO courses VALUES (120, 182, 'OBJECT-ORIENTED DESIGN');

INSERT INTO courses VALUES (120, 141, 'INTRODUCTION TO Ada');

INSERT INTO courses VALUES (120, 140, 'ADVANCED 7090 ASSEMBLER');
```

EMROLMNT.SQL

```
REM  Create some classes and enroll some students in
REM  them, to test the procedures that access
REM  the ENROLLMENT table.
```

```
DELETE FROM classes;

REM  Department 150 is HISTORY

INSERT INTO classes VALUES (900, 101, 150, 300, 2520, 100, 9001, 1, 1990);

INSERT INTO classes VALUES (901, 236, 150, 20, 2520, 111, 9001, 3, 1990);

INSERT INTO classes VALUES (902, 237, 150, 15, 2520, 111, 9001, 4, 1990);

INSERT INTO classes VALUES (903, 238, 150, 10, 2520, 111, 9001, 1, 1991);

REM  Department 120 is COMPUTER SCIENCE
INSERT INTO classes VALUES (910, 210, 120, 60, 2530, 34, 9003, 1, 1990);

INSERT INTO classes VALUES (911, 182, 120, 120, 2530, 440, 9003, 1, 1991);

INSERT INTO classes VALUES (912, 141, 120, 60, 2530, 334, 9003, 2, 1990);

INSERT INTO classes VALUES (913, 140, 120, 300, 2530, 112, 9003, 1, 1989);

REM  Now enroll Susan and Michael in some courses.

DELETE FROM enrollment
        WHERE student_id =
        (SELECT id FROM students
                WHERE first_name = 'Susan'
                AND last_name = 'Brahms');

DELETE FROM enrollment
        WHERE student_id =
        (SELECT id FROM students
                WHERE first_name = 'Michael'
                AND last_name = 'Chen');

INSERT INTO enrollment VALUES (enrollment_seq.nextval,
        900, 1000, 3.0, 'Good');

INSERT INTO enrollment VALUES (enrollment_seq.nextval,
        901, 1000, 3.5, 'Very Good');

INSERT INTO enrollment VALUES (enrollment_seq.nextval,
        902, 1000, 4.0, 'Excellent');

INSERT INTO enrollment VALUES (enrollment_seq.nextval,
        903, 1000, 2.0, 'Fair');

INSERT INTO enrollment VALUES (enrollment_seq.nextval,
        910, 1007, 3.0, ' ');

INSERT INTO enrollment VALUES (enrollment_seq.nextval,
        911, 1007, 3.0, ' ');

INSERT INTO enrollment VALUES (enrollment_seq.nextval,
        912, 1007, 4.0, ' ');
INSERT INTO enrollment VALUES (enrollment_seq.nextval,
        913, 1007, 2.0, ' ');
```

Module Language Sample Program

```
-- SQL*Module demonstration module.
-- Contains procedures to maintain the college database.

-- PREAMBLE

MODULE        demomod
LANGUAGE      Ada
AUTHORIZATION modtest

------------------------------------------------------------------
---------------------- STUDENTS TABLE----------------------
------------------------------------------------------------------

-- The following cursors and procedures access the STUDENTS table
-- or the STUDENT_ID_SEQ sequence number generator.

-- Declare a cursor to select all students
-- in the college.

DECLARE GET_STUDENTS_CURS CURSOR FOR

    SELECT last_name, first_name, mi, id, status,
           major, advisor_id
        FROM students

-- Define procedures to open and close this cursor.

PROCEDURE open_get_students_curs (
        SQLCODE);

    OPEN GET_STUDENTS_CURS;

PROCEDURE close_get_students_curs (
        SQLCODE);

    CLOSE GET_STUDENTS_CURS;
-- Define a procedure to fetch using the
-- get_students_curs cursor.

PROCEDURE get_all_students (
        :lname            CHAR(15),
        :fname            CHAR(15),
        :mi               CHAR(3),
        :mi_ind           SMALLINT,
        :id               INTEGER,
        :status           CHAR(5),
        :major            INTEGER,
        :major_ind        SMALLINT,      -- indicator for major
        :adv              INTEGER,
        :adv_ind          SMALLINT,      -- indicator for advisor
        SQLCODE);

    FETCH get_students_curs
        INTO :lname, :fname, :mi INDICATOR :mi_ind,
             :id, :status, :major INDICATOR :major_ind,
             :adv INDICATOR :adv_ind;

-- Add a new student
-- to the database.  Some of the columns in the
```

```
            -- table are entered as null in this procedure.
            -- The UPDATE_STUDENT procedure is used to fill
            -- them in later.

            PROCEDURE add_student (
                    :last_name      CHARACTER(15),
                    :first_name     CHARACTER(15),
                    :mi             CHARACTER(3),
                    :mi_ind         SMALLINT,
                    :sid            INTEGER,
                    :status         CHARACTER(5),
                    :date_of_birth  CHARACTER(9),
                    :dob_ind        SMALLINT,
                    SQLCODE);

                INSERT INTO students VALUES (
                    :last_name,
                    :first_name,
                    :mi :mi_ind,
                    :sid,
                    :status,
                    :date_of_birth :dob_ind,
                    sysdate,                        -- use today's date
                                                    -- for start date
                    null,                           -- no graduation date yet
                    null,                           -- no declared major yet
                    null                            -- no advisor yet
                    );
            -- Update a student's record to add or change
            -- status, major subject, advisor, and graduation date.

            PROCEDURE update_student (
                    :sid            INTEGER,        -- student's id number
                    :major          INTEGER,        -- dept number of major
                    :major_ind      SMALLINT,       -- indicator for major
                    :advisor        INTEGER,        -- advisor's ID number
                    :advisor_ind    SMALLINT,
                    :grd_date       CHARACTER(9),
                    :grad_date_ind  SMALLINT,
                    SQLCODE);

                UPDATE students SET
                    grad_date = :grd_date INDICATOR :grad_date_ind,
                    major = :major INDICATOR :major_ind,
                    advisor_id = :advisor INDICATOR :advisor_ind
                    WHERE id = :sid;

            PROCEDURE delete_student (
                    :sid            INTEGER,
                    SQLCODE);

                DELETE FROM students
                    WHERE id = :sid;

            -- Get an ID number for a new student
            -- using the student_id sequence generator.  This
            -- is done so that the ID number can be returned
            -- to the add_student routine that calls
            -- ENROLL.
```

```
PROCEDURE get_new_student_id (
        :new_id          INTEGER,
        SQLCODE);

    SELECT student_id_seq.nextval
        INTO :new_id
        FROM dual;

--  Return the name
--  of a student, given the ID number.

PROCEDURE get_student_name_from_id (
        :sid             INTEGER,
        :lname           CHAR(15),
        :fname           CHAR(15),
        :mi              CHAR(3),
        SQLCODE);

    SELECT last_name, first_name, mi
        INTO :lname, :fname, :mi
        FROM students
        WHERE id = :sid;

-----------------------------------------------------------------
----------------------- INSTRUCTORS TABLE --------------------
-----------------------------------------------------------------

--  Define a procedure to return an instructor's last
--  name, given the ID number.

PROCEDURE get_instructor_name_from_id (
        :iid             INTEGER,
        :lname           CHAR(15),
        :fname           CHAR(15),
        :imi             CHAR(3),
        :mi_ind          SMALLINT,
        SQLCODE);

    SELECT last_name, first_name, mi
        INTO :lname, :fname, :imi INDICATOR :mi_ind
        FROM instructors
        WHERE id = :iid;

-----------------------------------------------------------------
----------------------- DEPARTMENTS TABLE --------------------
-----------------------------------------------------------------

--  Define procedure to return the name of a department
--  given its ID number.

PROCEDURE get_department_name_from_id (
        :did             INTEGER,
        :dept_name       CHARACTER(16),
        SQLCODE);

    SELECT name
        INTO :dept_name
        FROM departments
        WHERE id = :did;
```

```
              ------------------------------------------------------------------
              ----------------------- COURSES TABLE -----------------------
              ------------------------------------------------------------------

              -- (none defined yet)

              ------------------------------------------------------------------
              ----------------------- CLASSES TABLE -----------------------
              ------------------------------------------------------------------

              -- Add a class to the classes table.

              PROCEDURE add_class (
                      :class_no        INTEGER,
                      :dept_no         INTEGER,
                      :course_no       INTEGER,
                      :max_students    INTEGER,
                      :instr_id        INTEGER,
                      :quarter         INTEGER,
                      :year            INTEGER,
                      SQLCODE);

                  INSERT INTO classes VALUES (
                      :class_no,
                      :course_no,
                      :dept_no,
                      :max_students,
                      null,                               -- building number and
                      null,                               -- room not yet assigned
                      :instr_id,
                      :quarter,
                      :year
                      );

              -- Drop a class.

              PROCEDURE delete_class (
                      :class_no        INTEGER,
                      SQLCODE);

                  DELETE FROM classes
                      WHERE class_number = :class_no;

              -- Get an ID number for a new class.
              -- A class is an instance of a course.
              -- Use the class_number_seq sequence generator.

              PROCEDURE get_new_class_id (
                      :new_id          INTEGER,
                      SQLCODE);

                  SELECT class_number_seq.nextval
                      INTO :new_id
                      FROM dual;

              ------------------------------------------------------------------
              --------------------- ENROLLMENT TABLE -----------------------
              ------------------------------------------------------------------

              -- Declare a cursor to return information about all
```

```
        -- classes a given student has or is enrolled in his
        -- or her college career.

        -- In this college, letter grades are assigned
        -- numbers, in the following format:
        -- A   4.0
        -- B+  3.5
        -- B   3.0
        -- C+  2.5
        -- C   2.0
        -- D   1.0
        -- F   0.0

DECLARE get_enroll_curs CURSOR FOR

    SELECT courses.name,
           classes.instructor,
           classes.year,
           classes.quarter,
           enrollment.grade,
           enrollment.comments
        FROM courses, classes, enrollment
        WHERE courses.id = classes.course_number
              AND classes.class_number = enrollment.class_no
              AND enrollment.student_id = :sid

-- Define a procedure to open the GET_ENROLL_CURS cursor.
-- Note that this procedure requires an IN parameter to set
-- the student ID number (sid).

PROCEDURE open_get_enroll_curs (
        :sid        INTEGER,
        SQLCODE);

    OPEN GET_ENROLL_CURS;

-- CLOSE the get_enroll_curs cursor

PROCEDURE close_get_enroll_curs (
        SQLCODE);

    CLOSE get_enroll_curs;

-- FETCH from the courses, classes, and enrollment table
-- using the get_enroll_curs cursor

PROCEDURE get_enroll_by_student (
        :course_name    CHARACTER(38),
        :instructor     INTEGER,
        :year           INTEGER,
        :quarter        INTEGER,
        :grade          REAL,
        :grade_ind      SMALLINT,
        :comments       CHARACTER(255),
        SQLCODE);

    FETCH get_enroll_curs
        INTO :course_name,
             :instructor,
             :year,
```

```
                :quarter,
                :grade INDICATOR :grade_ind,
                :comments;

-- Enroll a student in a class.

PROCEDURE enroll_student_in_class (
        :class_number  INTEGER,
        :sid           INTEGER,
        SQLCODE);

     INSERT INTO enrollment VALUES (
         enrollment_seq.nextval,
         :class_number,
         :sid,
         null,                          -- no grade yet
         ' '                            -- no comments yet
         );

-----------------------------------------------------------------
---------------------- UTILITY PROCEDURES --------------------
-----------------------------------------------------------------

-- Commit a transaction.

PROCEDURE do_commit(
        SQLCODE);

     COMMIT WORK;

-- Connect to a database

PROCEDURE do_connect (
        :dbname      CHARACTER(14),
        :username    CHARACTER(14),
        :passwd      CHARACTER(14),
        SQLCODE);

     CONNECT TO :dbname USER :username USING :passwd;

-- Disconnect

PROCEDURE do_disconnect (
        SQLCODE);

     DISCONNECT CURRENT;

-- Roll a transaction back.

PROCEDURE do_rollback (
        SQLCODE);

     ROLLBACK WORK;
```

Calling a Stored Procedure

The sample stored package defined can be used to demonstrate how to call a stored procedure from an Ada application. The package source is GPAPKG.SQL, and it is in your demo directory. See the program "DEMCALSP.A" on page 6-28, written in the

host language, that calls the GET_GPA_IF procedure in this package. Each of these host programs is also on-line, in your demo directory.

```
--  Create the specification for a package
--  that contains the GET_GPA stored procedure.
--  Use the WITH INTERFACE clause so that
--  the package procedure can be called from a 3GL.

--  Note that the procedure parameters have PL/SQL
--  datatypes, but in the WITH INTERFACE clause
--  SQL datatypes must be used, and they must be
--  constrained if required (for example, CHARACTER(15)).
--  The WITH INTERFACE clause enables you to
--  specify error-handling parameters, such as SQLSTATE,
--  as well as indicator parameters.  These are filled
--  in as the procedure executes.

--  The calling host 3GL application calls the procedure
--  named in the WITH INTERFACE clause.  This
--  would usually be given the same name as the procedure
--  in the body.  Here it is given a different name, to
--  demonstrate that (1) you can do this, and (2) it is
--  the WITH INTERFACE clause name that gets
--  generated in the interface procedure as the procedure to call.

--  Note that this package will create
--  the package and procedure names in uppercase. So the
--  module compiler will generate interface procedures that have
--  the names
--  in uppercase, which means that you must call them using
--  upper case in your host program. If you prefer lowercase,
--  simply change the package and procedure names to be
--  quoted lowercase, for example:
--
--    CREATE OR REPLACE PACKAGE "gpa_pkg" AS ...

CREATE OR REPLACE PACKAGE GPA_PKG AS

    PROCEDURE GET_GPA(student_id       IN     NUMBER,
                      student_last_name IN OUT CHARACTER,
                      gpa               OUT    NUMBER)
    WITH INTERFACE
    PROCEDURE GET_GPA_IF
                      (student_id       INTEGER,
                      student_last_name CHARACTER(15)
                                         INDICATOR sname_ind,
                      sname_ind         SMALLINT,
                      gpa               REAL,
                      sqlstate          CHARACTER(5),
                      sqlcode           INTEGER);
END;

--  Create the package body.  There is no need for
--  a WITH INTERFACE clause in the body.
--  The GET_GPA procedure computes the cumulative GPA
--  over all courses that the student has taken, and returns
--  the computed value.  If the student has received no
--  grades yet, a null is returned (through the indicator
--  parameter).
```

```
CREATE OR REPLACE PACKAGE BODY GPA_PKG AS

    PROCEDURE GET_GPA(student_id        IN     NUMBER,
                      student_last_name IN OUT CHARACTER,
                      gpa               OUT    NUMBER) IS

-- The cursor selects all the classes that
-- the student has enrolled in.

    CURSOR get_enroll_curs(sid IN NUMBER) IS
        SELECT enrollment.grade
        FROM   enrollment
        WHERE  enrollment.student_id = sid
          AND  enrollment.grade IS NOT NULL;

-- Declare local variables.
-- gpa_temp needed because gpa is an OUT parameter
    n        NUMBER   := 0;
    grade    NUMBER;
    gpa_temp NUMBER   := 0;

    BEGIN
        gpa := 0.0;

-- Get the last name;
-- if not found, the no_data_found
-- predefined exception is raised.
        SELECT last_name
          INTO student_last_name
          FROM students
          WHERE id = student_id;

-- Otherwise, open the cursor and FETCH.
        open get_enroll_curs(student_id);
        loop
            FETCH get_enroll_curs INTO grade;
            exit when get_enroll_curs%notfound;
            gpa_temp := gpa_temp + grade;
            n := n + 1;
        end loop;

        close get_enroll_curs;

        if n > 0 then
            gpa := gpa_temp / n;
        end if;

    exception

-- The SQLCODE parameter in the WITH INTERFACE
-- parameter list will not be set to +100 because
-- the exception is handled here, but the indicator
-- variable will be set to -1 because of the null
-- assignment.
        when no_data_found then
            student_last_name := null;
    end GET_GPA;

END;
```

Sample Applications

This section contains sample applications that may aid development.

DEMOHOST.A

```
--      Module Language demonstration program for Ada.
--      For an explanation of the tables that are accessed
--      and the Module Language procedures that
--      are called in this program, see  Sample Programs.
--
--      The module language code that contains the procedures called
--      by this program, and SQL scripts to create and populate
--      the tables used, are included in the source distribution.
--

with
-- The required SQL standard package.
    sql_standard,

-- The module language procedures package.
    demomod,

-- Other I/O packages...
    text_io,
    float_text_io,
    integer_text_io;

use
-- use the standard I/O packages.
    text_io,
    sql_standard,
    float_text_io,
    integer_text_io;

procedure DEMOHOST is

-- instantiate new packages for I/O on SQL_STANDARD datatypes
    package STD_INT_IO is
        new text_io.integer_io(SQL_STANDARD.INT);
    use STD_INT_IO;

    package SQLCODE_IO is
        new text_io.integer_io(SQL_STANDARD.SQLCODE_TYPE);
    use SQLCODE_IO;


    package STD_SMALLINT_IO is
        new text_io.integer_io(SQL_STANDARD.SMALLINT);
    use STD_SMALLINT_IO;

    package STD_FLOAT_IO is
        new text_io.float_io(SQL_STANDARD.REAL);
    use STD_FLOAT_IO;


-- declare main procedure variables and exceptions


-- handle command input
    type COMMAND is
```

```
                  (AC, AS, DC, DS, ES, SE, SS, US, HELP, QUIT, BYE);

         package COMMAND_IO is
             new text_io.enumeration_io(COMMAND);
         use COMMAND_IO;

         COM_LINE      : COMMAND;

-- make SQLCODE global since program structure allows this
         SQLCODE       : SQL_STANDARD.SQLCODE_TYPE;
         ANSWER        : string(1..4);
         LENGTH        : integer;
         SERVICE_NAME  : SQL_STANDARD.CHAR(1..14);
         USERNAME      : SQL_STANDARD.CHAR(1..14);
         PASSWORD      : SQL_STANDARD.CHAR(1..14);

-- declare top-level exceptions
         CONNECT_ERROR : exception;
         SQLCODE_ERROR : exception;


-- define procedures

-- get a user command
      procedure GET_COMMAND(CMD : out COMMAND) is
      begin
           loop
           begin
               new_line(2);
               put("Select an option: ");
               get(CMD);
               return;
           exception
               when data_error =>
                   put_line
                     (ascii.bel & "Invalid option, try again.");
           end;
           end loop;
      end GET_COMMAND;


      procedure MENU is
      begin
          new_line(5);
          put_line("                 *** COLLEGE RECORDS ***");
          new_line;
          put_line("AC  -  add a class to curriculum");
          put_line("AS  -  enroll a new student in the college");
          put_line("DC  -  drop a class from curriculum");
          put_line("DS  -  drop a student");
          put_line("ES  -  enroll a student in a class");
          put_line("SE  -  show complete enrollment records");
          put_line("SS  -  show all students");
          put_line("US  -  update a student's record");
          put_line("HELP - redisplay this menu");
          put_line("QUIT - quit program");
          new_line(3);
      end MENU;
```

```
-- Procedure to get an integer value from the user,
-- prompting first.
procedure GET_STANDARD_INT(PROMPT : string;
                           VALUE  : out SQL_STANDARD.INT) is

begin
    put(prompt);
    get(integer(VALUE));
    skip_line;
end GET_STANDARD_INT;


-- Get a text string from the user, prompting first.
-- The string is blank-padded.
procedure GET_STANDARD_TEXT(PROMPT : in     string;
                            VALUE  : out    SQL_STANDARD.CHAR;
                            LENGTH : in out integer) is
    OLD_LENGTH : integer;

begin
    OLD_LENGTH := LENGTH;
    put(PROMPT);
    VALUE := (1..LENGTH => ' ');
    get_line(string(VALUE), LENGTH);

    if LENGTH = OLD_LENGTH then
        skip_line;
    end if;

end GET_STANDARD_TEXT;


-- The following procedures, all beginning with the prefix
-- "CALL_", are called from the main procedure,
-- and in turn call Module Language procedures, defined
-- in the DEMOMOD.mad file.

procedure CALL_ADD_CLASS is

    CLASS_NUMBER      : SQL_STANDARD.INT;
    DEPARTMENT_NUMBER : SQL_STANDARD.INT;
    COURSE_NUMBER     : SQL_STANDARD.INT;
    MAX_ENROLLMENT    : SQL_STANDARD.INT;
    INSTRUCTOR_ID     : SQL_STANDARD.INT range
                            1000..SQL_STANDARD.INT'last;
    QUARTER           : SQL_STANDARD.INT range 1..4;
    YEAR              : SQL_STANDARD.INT range 1900..2100;


begin
    new_line(2);
    put_line("Add a new class to the schedule");
    new_line(2);

    DEMOMOD.GET_NEW_CLASS_ID(CLASS_NUMBER, SQLCODE);

    if SQLCODE /= 0 then
        put("Cannot generate new class number. CODE is ");
        put(SQLCODE);
        new_line;
```

```
            put_line("  Call your database administrator.");
            return;
        else
            put("New class number is ");
            put(CLASS_NUMBER);
            new_line;
        end if;

        loop
        begin
            new_line;
            GET_STANDARD_INT
                ("Enter dept ID: ", DEPARTMENT_NUMBER);

            GET_STANDARD_INT
              ("Enter course ID number: ", COURSE_NUMBER);
            GET_STANDARD_INT
              ("maximum enrollment: ", MAX_ENROLLMENT);
            GET_STANDARD_INT
              ("instructor ID number: ", INSTRUCTOR_ID);
            GET_STANDARD_INT
              ("quarter (1=spring, 2=summer, ...: ", QUARTER);
            GET_STANDARD_INT("year (4 digits please): ", YEAR);

            DEMOMOD.ADD_CLASS(CLASS_NUMBER, COURSE_NUMBER,
                            DEPARTMENT_NUMBER, MAX_ENROLLMENT,
                            INSTRUCTOR_ID,
                            QUARTER, YEAR, SQLCODE);
            if SQLCODE /= 0 then
                put("Error adding class.  CODE is ");
                put(SQLCODE);
                new_line;
            else
                put_line("New class added.");
            end if;
            exit;
        exception
            when CONSTRAINT_ERROR =>
                new_line;
                put_line("Last input not valid.  Try again.");
                new_line;
        end;
        end loop;
    end CALL_ADD_CLASS;


    procedure CALL_ADD_STUDENT is
        ERROR_COUNT   : integer := 0;
        SIZE          : integer;
        NEW_ID        : SQL_STANDARD.INT;
        MI_IND        : SQL_STANDARD.SMALLINT;
        TEMP_STRING   : string(1..80);
        FIRST_NAME    : SQL_STANDARD.CHAR(1..15);
        LAST_NAME     : SQL_STANDARD.CHAR(1..15);
        MI            : SQL_STANDARD.CHAR(1..3);
        DATE_OF_BIRTH : SQL_STANDARD.CHAR(1..9);
        DOB_IND       : SQL_STANDARD.SMALLINT;
        STATUS        : SQL_STANDARD.CHAR(1..5);
        LENGTH        : integer;
```

```
begin
    new_line(2);
    put_line("Add a new student to the database.");
    new_line(2);


    DEMOMOD.GET_NEW_STUDENT_ID(NEW_ID, SQLCODE);
    if SQLCODE /= 0 then
        put_line("Cannot generate ID number for student.");
        put("CODE is ");
        put(SQLCODE);
        new_line;
        put_line("Call your database administrator.");
        return;
    end if;

    skip_line;
    loop
        begin
            new_line;
            LENGTH := 15;
            GET_STANDARD_TEXT("  Last name: ", LAST_NAME,
                                    LENGTH);

            LENGTH := 15;
            GET_STANDARD_TEXT("  First name: ", FIRST_NAME,
                                    LENGTH);

            LENGTH := 3;
            GET_STANDARD_TEXT("  Middle initial: ", MI,
                                    LENGTH);

            if LENGTH = 0 then
                MI_IND := -1;
            else
                MI_IND := 0;
            end if;

            LENGTH := 9;
            GET_STANDARD_TEXT("  Date of birth (DD-MON-YY): ",
                                    DATE_OF_BIRTH, LENGTH);

            if LENGTH = 0 then
                DOB_IND := -1;
            else
                DOB_IND := 0;
            end if;

            LENGTH := 5;
            GET_STANDARD_TEXT("  Status (FT, PT, JYA, ...): ",
                                    STATUS, LENGTH);

            DEMOMOD.ADD_STUDENT(LAST_NAME,
                                        FIRST_NAME,
                                        MI, MI_IND,
                                        NEW_ID,
                                        STATUS,
                                        DATE_OF_BIRTH,
                                        DOB_IND,
                                        SQLCODE);
```

```
                    if SQLCODE /= 0 then
                        new_line;
                        put("Error adding student. CODE is ");
                        put(SQLCODE, width => 5);
                    else
                        new_line;
                        put("Student added.  ID number is");
                        put(NEW_ID, width => 6);
                    end if;
                    new_line(3);
                    return;
            exception
                when constraint_error =>
                    ERROR_COUNT := ERROR_COUNT + 1;
                    if ERROR_COUNT > 3 then
                        put_line
                        ("Too many errors. Back to main program.");
                         exit;
                    end if;
                    put_line("Invalid value.  Try again.");
                when others =>
                    put_line("Data error or other error.");
                    exit;
            end;
        end loop;
end CALL_ADD_STUDENT;


procedure CALL_DROP_CLASS is
    CLASS_NUMBER      : SQL_STANDARD.INT;

begin
    new_line(2);
    put_line("Drop a class");
    new_line(2);

    GET_STANDARD_INT
      ("  Enter class ID number: ", CLASS_NUMBER);

    DEMOMOD.DELETE_CLASS(CLASS_NUMBER, SQLCODE);

    if SQLCODE /= 0 then
        new_line;
        put("Error dropping the class.  CODE is ");
        put(SQLCODE);
        new_line;
        put_line("Call your database administrator.");
    else
        put_line("Class dropped.");
    end if;
end CALL_DROP_CLASS;


procedure CALL_DROP_STUDENT is
    LAST_NAME, FIRST_NAME   : SQL_STANDARD.CHAR(1..15);
    MI                      : SQL_STANDARD.CHAR(1..3);
    STUDENT_ID              : SQL_STANDARD.INT;
    ANSWER                  : string(1..12);
    ALEN                    : integer;
```

```
begin
    new_line(2);
    put_line("Drop a student from the college.");
    new_line(2);

    GET_STANDARD_INT
       ("  Enter student ID number: ", STUDENT_ID);
    DEMOMOD.GET_STUDENT_NAME_FROM_ID(STUDENT_ID,
                                     LAST_NAME,
                                     FIRST_NAME, MI,
                                     SQLCODE);
    if SQLCODE /= 0 then
        new_line;
        put("Error getting student information.  CODE is ");
        put(SQLCODE);
        new_line;
        put_line("Call your database administrator.");
        return;
    end if;

    put_line("Student's name is--");
    put_line(string(FIRST_NAME & MI & LAST_NAME));
    put("Do you really want to do this? ");
    get_line(ANSWER, ALEN);
    if ANSWER(1) = 'Y' or ANSWER(1) = 'y' then
        DEMOMOD.DELETE_STUDENT(STUDENT_ID, SQLCODE);
        if SQLCODE /= 0 then
            put_line("Error dropping student.  CODE is ");
            put(SQLCODE);
            return;

        else
            put_line
             (string(LAST_NAME) & " has been dropped!");
        end if;
    else
        put_line("OK, student will not be dropped.");
    end if;

end CALL_DROP_STUDENT;


procedure CALL_ENROLL_STUDENT is
    CLASS_NUMBER, STUDENT_ID   : SQL_STANDARD.INT;
    LAST_NAME, FIRST_NAME      : SQL_STANDARD.CHAR(1..15);
    MI                         : SQL_STANDARD.CHAR(1..3);

begin
    new_line(2);
    put_line("Enroll a student in a class.");
    new_line(2);

    GET_STANDARD_INT("  Enter student ID: ", STUDENT_ID);
    GET_STANDARD_INT("  Enter class   ID: ", CLASS_NUMBER);
    DEMOMOD.GET_STUDENT_NAME_FROM_ID(STUDENT_ID,
                                     LAST_NAME,
                                     FIRST_NAME,
                                     MI,
                                     SQLCODE);
    if SQLCODE /= 0 then
```

```
                new_line;
                put_line("That student ID does not exist.");
                put("CODE is ");
                put(SQLCODE);
                new_line;
                put_line("Recheck and try again.");
            else
                put_line
                  ("  The student's name is " & string(LAST_NAME));
                put("  Enrolling...");
                DEMOMOD.ENROLL_STUDENT_IN_CLASS(CLASS_NUMBER,
                                                STUDENT_ID,
                                                SQLCODE);

                if SQLCODE /= 0 then
                    new_line;
                    put("Error occurred enrolling student. CODE is ");
                    put(SQLCODE);
                    new_line;
                    put_line("Check class ID number and try again.");

                else
                    put_line("done");
                end if;
            end if;
        end CALL_ENROLL_STUDENT;


        procedure CALL_SHOW_ENROLLMENT is
            COURSE_NAME                 : SQL_STANDARD.CHAR(1..38);
            INSTR_ID, SID, YEAR, QUARTER : SQL_STANDARD.INT;
            GRADE, GPA                  : SQL_STANDARD.REAL;
            GRADE_IND                   : SQL_STANDARD.SMALLINT;
            COMMENTS                    : SQL_STANDARD.CHAR(1..255);
            GRADE_COUNT, ROW_COUNT      : integer;

        begin
            new_line(2);
            put_line("Show enrollment in all courses for a student.");
            new_line(2);

            GET_STANDARD_INT
              ("  Enter student ID number (try 1000): ", SID);

            DEMOMOD.OPEN_GET_ENROLL_CURS(SID, SQLCODE);
            if SQLCODE /= 0 then
                new_line;
                put("Error opening cursor. CODE is ");
                put(SQLCODE);
                new_line;
                put_line("Call your database administrator.");
            else
                GPA := 0.0;
                GRADE_COUNT := 0;
                ROW_COUNT := 0;

                put("COURSE TITLE                             ");
                put_line("INSTR ID   YEAR    QUARTER    GRADE");

                loop
                    DEMOMOD.GET_ENROLL_BY_STUDENT(COURSE_NAME,
```

```
                                            INSTR_ID,
                                            YEAR, QUARTER,
                                            GRADE, GRADE_IND,
                                            COMMENTS,
                                            SQLCODE);
            if SQLCODE = 100   then
                exit;
            elsif SQLCODE /= 0 then
                new_line;
                put_line("Error fetching data.  CODE is ");
                put(SQLCODE);
                new_line;
                put_line("Call your database administrator.");
                exit;
            else
                ROW_COUNT := ROW_COUNT + 1;
                put(string(COURSE_NAME));
                put(INSTR_ID, width => 6);
                put(YEAR, width => 11);
                put(QUARTER, width => 6);
                if GRADE_IND >= 0 then
                    GRADE_COUNT := GRADE_COUNT + 1;
                    GPA := GPA + GRADE;
                    put(GRADE, fore => 7, aft => 2, exp => 0);
                end if;
            end if;
            new_line;
        end loop;

        if GRADE_COUNT > 0 and SQLCODE = 100 then
            new_line;
            GPA := GPA / REAL(GRADE_COUNT);
            put("Overall GPA is ");
            put(GPA, fore => 1, aft => 2, exp => 0);
        end if;

        DEMOMOD.CLOSE_GET_ENROLL_CURS(SQLCODE);
        if SQLCODE /= 0 then
            new_line;
            put("Error closing cursor. CODE is ");
            put(SQLCODE);
            new_line;
        end if;
    end if;

end CALL_SHOW_ENROLLMENT;


procedure CALL_SHOW_STUDENTS is
    LAST_NAME, FIRST_NAME      : SQL_STANDARD.CHAR(1..15);
    MI                         : SQL_STANDARD.CHAR(1..3);

    INSTR_LAST_NAME            : SQL_STANDARD.CHAR(1..15);
    INSTR_FIRST_NAME           : SQL_STANDARD.CHAR(1..15);
    INSTR_MI                   : SQL_STANDARD.CHAR(1..3);

    MI_IND, INSTR_MI_IND       : SQL_STANDARD.SMALLINT;
    SID, MAJOR, ADVISOR, INSTR : SQL_STANDARD.INT;
    MAJOR_IND, ADVISOR_IND     : SQL_STANDARD.SMALLINT;
    STATUS                     : SQL_STANDARD.CHAR(1..5);
```

```
begin
    new_line(2);
    put_line("   ----- STUDENTS CURRENTLY ENROLLED -----");
    new_line(2);

    put("LAST NAME      FIRST NAME    MI  ID NO  STATUS");
    put_line(" MAJOR  ADVISOR");
    DEMOMOD.OPEN_GET_STUDENTS_CURS(SQLCODE);
    if SQLCODE /= 0 then
        new_line;
        put("Error opening cursor. CODE is ");
        put(SQLCODE);
        new_line;
        put_line("Call your database administrator.");
        return;
    end if;

    loop
        DEMOMOD.GET_ALL_STUDENTS(LAST_NAME,
                                    FIRST_NAME,
                                    MI, MI_IND,
                                    SID, STATUS,
                                    MAJOR, MAJOR_IND,
                                    ADVISOR, ADVISOR_IND,
                                    SQLCODE);
        if SQLCODE = 100 then
            exit;
        elsif SQLCODE /= 0 then
            new_line;
            put_line("Error fetching data.  CODE is ");
            put(SQLCODE);
            new_line;
            put_line("Call your database administrator.");
            exit;
        else
            put(string(LAST_NAME));
            put(string(FIRST_NAME));
            put(string(MI));
            put(SID, width => 5);
            put("   ");
            put(string(STATUS));
            put("  ");

            if MAJOR_IND < 0 then
                put("(NONE)");
            else
                put(MAJOR);
            end if;
            if ADVISOR_IND = 0 then
                DEMOMOD.GET_INSTRUCTOR_NAME_FROM_ID
                                    (ADVISOR,
                                     INSTR_LAST_NAME,
                                     INSTR_FIRST_NAME,
                                     INSTR_MI, INSTR_MI_IND,
                                     SQLCODE);

                if SQLCODE = 0 then
                    put(" " & string(INSTR_LAST_NAME));
                else
                    put("[err = ");
```

```
                        put(SQLCODE);
                        put("]");
                    end if;
                else
                    put(" (NONE)");
                end if;
            end if;
            new_line;
    end loop;

    DEMOMOD.CLOSE_GET_STUDENTS_CURS(SQLCODE);
    if SQLCODE /= 0 then
        new_line;
        put("Error closing cursor. CODE is ");
        put(SQLCODE);
        new_line;
        put_line("Call your database administrator.");
        new_line;
    end if;

end CALL_SHOW_STUDENTS;


procedure CALL_UPDATE_RECORD is
    SID, ADVISOR, MAJOR    : SQL_STANDARD.INT;
    GRAD_DATE              : SQL_STANDARD.CHAR(1..9);
    ADVISOR_IND, MAJOR_IND : SQL_STANDARD.SMALLINT;
    GRAD_DATE_IND          : SQL_STANDARD.SMALLINT;
    LENGTH                 : integer;
    LAST_NAME              : SQL_STANDARD.CHAR(1..20);
    FIRST_NAME             : SQL_STANDARD.CHAR(1..20);
    MI                     : SQL_STANDARD.CHAR(1..3);

begin
    new_line(2);
    put_line("Update a student's records.");
    new_line(2);

    GET_STANDARD_INT("  Enter student ID number: ", SID);
    DEMOMOD.GET_STUDENT_NAME_FROM_ID(SID,
                                     LAST_NAME,
                                     FIRST_NAME,
                                     MI,
                                     SQLCODE);

    if SQLCODE /= 0 then
        new_line;
        put_line("That student ID does not exist.");
        new_line;
        put_line("Recheck and try again.");
        return;
    else
        put_line
        ("  The student's last name is " & string(LAST_NAME));
        new_line;
    end if;

    put("  Change major?  If so, enter new department ");
    GET_STANDARD_INT("number.  If not, enter 0: ", MAJOR);
```

```
            if MAJOR = 0 then
                MAJOR_IND := -1;
            else
                MAJOR_IND := 0;
            end if;

            put("  New advisor?  If so, enter the instructor ID ");
            GET_STANDARD_INT("number.  If not, enter 0: ", ADVISOR);

            if ADVISOR = 0 then
                ADVISOR_IND := -1;
            else
                ADVISOR_IND := 0;
            end if;

            put_line
    ("  Has the student graduated.  If so, enter date (DD-MON-YY)");
            LENGTH := 9;
            GET_STANDARD_TEXT
               ("  If not, press RETURN: ", GRAD_DATE, LENGTH);

            if LENGTH = 0 then
                GRAD_DATE_IND := -1;
            else
                GRAD_DATE_IND := 0;
            end if;


            DEMOMOD.UPDATE_STUDENT(SID,
                                   MAJOR, MAJOR_IND,
                                   ADVISOR, ADVISOR_IND,
                                   GRAD_DATE, GRAD_DATE_IND,
                                   SQLCODE);
            if SQLCODE /= 0 then
                new_line;
                put("Error updating records.  Code is ");
                put(SQLCODE);
                new_line;
                put_line("Call your database administrator.");
            else
                new_line;
                put_line("Records updated. ");
            end if;

    end CALL_UPDATE_RECORD;

-------------------------------------------------------------------
-------------------- main ------------------------------------------
-------------------------------------------------------------------

begin

    SQLCODE_IO.default_width := 6;

    SERVICE_NAME := "inst1_alias   ";
    USERNAME     := "modtest       ";
    PASSWORD     := "yes           ";
    DEMOMOD.DO_CONNECT(SERVICE_NAME, USERNAME, PASSWORD, SQLCODE);
    if SQLCODE /= 0 then
        raise connect_error;
```

```
        end if;
        put_line("Connected to ORACLE.");
        new_line;
        MENU;

        loop
            GET_COMMAND(COM_LINE);
            case COM_LINE is
                when AC => CALL_ADD_CLASS;
                when AS => CALL_ADD_STUDENT;
                when DC => CALL_DROP_CLASS;
                when DS => CALL_DROP_STUDENT;
                when ES => CALL_ENROLL_STUDENT;
                when SE => CALL_SHOW_ENROLLMENT;
                when SS => CALL_SHOW_STUDENTS;
                when US => CALL_UPDATE_RECORD;
                when HELP => MENU;
                when QUIT | BYE =>
                    skip_line;
                    new_line(5);
                    put("Commit all changes [yn]: ");
                    LENGTH := 4;
                    get_line(ANSWER, LENGTH);
                    if (ANSWER(1..1) = "y") then
                        DEMOMOD.DO_COMMIT(SQLCODE);
                        put_line("Changes committed.");
                    else
                        DEMO_MOD.DO_ROLLBACK;
                        put_line("Changes discarded.");
                    end if;
                    new_line(2);
                    put_line("G'Day!");
                    new_line(4);
                    exit;
            end case;
        end loop;
        DEMOMOD.DO_DISCONNECT(SQLCODE);
        if SQLCODE /= 0 then
            put("Error disconnecting. SQLCODE is ");
            put(SQLCODE);
            put_line("Exiting anyway.");
        end if;
exception
    when CONNECT_ERROR =>
        put_line("Error connecting to ORACLE.");
        new_line(4);
    when SQLCODE_ERROR =>
        put("Error fetching data.  CODE is ");
        put(sqlcode);
        new_line(4);
        DEMOMOD.DO_DISCONNECT(SQLCODE);
    when others =>
        put_line("Unhandled error occurred. Fix the program!");
        new_line(4);

end DEMOHOST;
```

DEMCALSP.A

```
-- demcalsp.a
--
-- Sample program that demonstrates how to call a
-- database stored procedure using the WITH INTERFACE
-- PROCEDURE clause.
--
-- The stored package is in the file GPAPKG.SQL.

-- Include the required specs. Demomod must be included
-- since it contains the connect and disconnect procedures.

with TEXT_IO,
     SQL_STANDARD,
     GPA_PKG,
     DEMOMOD,
     FLOAT_TEXT_IO,
     INTEGER_TEXT_IO;

use  TEXT_IO,
     SQL_STANDARD,
     FLOAT_TEXT_IO,
     INTEGER_TEXT_IO;

procedure DEMCALSP is

-- Define the required I/O packages for SQL_STANDARD.
    package STD_INT_IO is
        new TEXT_IO.INTEGER_IO(SQL_STANDARD.INT);
    use STD_INT_IO;

    package SQLCODE_IO is
        new TEXT_IO.INTEGER_IO(SQL_STANDARD.SQLCODE_TYPE);
    use SQLCODE_IO;

    package STD_SMALLINT_IO is
        new TEXT_IO.INTEGER_IO(SQL_STANDARD.SMALLINT);
    use STD_SMALLINT_IO;

    package STD_FLOAT_IO is
        new TEXT_IO.FLOAT_IO(SQL_STANDARD.REAL);
    use STD_FLOAT_IO;

    STUDENT_ID        : SQL_STANDARD.INT;
    STUDENT_LAST_NAME : SQL_STANDARD.CHAR(1..15);
    NAME_IND          : SQL_STANDARD.SMALLINT;
    GPA               : SQL_STANDARD.REAL;
    PASSWORD          : SQL_STANDARD.CHAR(1..12);
    SERVICE_NAME      : SQL_STANDARD.CHAR(1..12);
    USERNAME          : SQL_STANDARD.CHAR(1..12);
    SQLCODE           : SQL_STANDARD.SQLCODE_TYPE;
    SQLSTATE          : SQL_STANDARD.SQLSTATE_TYPE;

    CONNECT_ERROR     : exception;
    SQLCODE_ERROR     : exception;

begin

    PASSWORD     := "yes         ";
    SERVICE_NAME := "inst1_alias ";
```

```
        USERNAME     := "modtest      ";

        DEMOMOD.DO_CONNECT(SERVICE_NAME, USERNAME, PASSWORD, SQLCODE);
        if SQLCODE /= 0 then
            raise CONNECT_ERROR;
        end if;
        new_line(2);
        put_line("Get grade point average--");
        new_line;

        loop
        begin
            new_line;
            put("Enter student ID number (try 1000) (0 to quit): ");
            get(STUDENT_ID);
            new_line;
            exit when STUDENT_ID = 0;

-- Call the stored procedure.
            GPA_PKG.GET_GPA_IF(STUDENT_ID, STUDENT_LAST_NAME,
                               NAME_IND, GPA, SQLSTATE, SQLCODE);
            if SQLCODE /= 0 then
                raise SQLCODE_ERROR;
            end if;

            if NAME_IND = 0 then
                new_line;
                put("Last name is " & string(STUDENT_LAST_NAME));
                put("Overall GPA is");
                put(GPA, fore => 4, aft => 2, exp => 0);
            else
                put("There is no student with ID number");
                put(STUDENT_ID, width => 5);
                new_line;
            end if;
        exception
            when SQLCODE_ERROR =>
                new_line;
                put("Error fetching data, SQLCODE is ");
                put(SQLCODE, width => 5);
        end;
        end loop;

-- Disconnect from the server.
        DEMOMOD.DO_DISCONNECT(SQLCODE);
        if SQLCODE /= 0 then
            put("Error disconnecting. SQLCODE is ");
            put(SQLCODE);
            put_line("Exiting anyhow.");
        end if;

exception
    when CONNECT_ERROR =>
        put("Error connecting to Oracle.");

end DEMCALSP;
```

A

New Features

This appendix contains a list of the new features in release 8.0 of SQL*Module for Ada.

New Statements

- CONNECT. (See "CONNECT Statement" on page 2-13.)
- SET CONNECTION. (See "SET CONNECTION Statement" on page 2-14.)
- DISCONNECT. (See "DISCONNECT Statement" on page 2-14.)
- ENABLE THREADS. (See "ENABLE THREADS" on page 2-15.)
- CONTEXT ALLOCATE. (See "CONTEXT ALLOCATE" on page 2-16.)
- CONTEXT FREE. (See "CONTEXT FREE" on page 2-16.)

Other New Features

- New datatype for multi-tasking, SQL_CONTEXT. (See "SQL_CONTEXT Datatype" on page 2-16.)
- Support for procedure declaration with arrays. (See "Arrays as Procedure Arguments" on page 4-7.)
- Function for obtaining the rows processed. (See "Obtaining the Number of Rows Processed" on page 4-4.)
- Function for obtaining error message text. (See "Obtaining Error Message Text" on page 4-2.)
- Dynamic SQL. (See "Dynamic SQL" on page 3-9.)

B

Module Language Syntax

This appendix describes the syntax of Module Language, using syntax diagrams.

For the complete syntax of all SQL statements, including syntax diagrams, see *Oracle Database PL/SQL Language Reference*.

Module Language Syntax Diagrams

Syntax diagrams use lines and arrows to show how procedure names, parameters, and other language elements are sequenced to form statements. Follow each diagram in the direction shown by the lines and arrows.

In these syntax diagrams, Module Language keywords appear in uppercase; parameters or other variable items appear in lowercase. Delimiters and terminators (such as '(', ',', and so on) appear in their literal form inside circles.

If the syntax diagram contains more than one possible path, you must select the path appropriate to your application.

If you have the choice of more than one keyword or parameter, your options appear in a vertical list. If any of the parameters in a vertical list appears on the main path, then one of them is required. That is, you must choose one of the parameters, but not necessarily the one that appears on the main path.

Single required parameters appear on the main path, that is, on the horizontal line you are currently traveling. If parameters appear in a vertical list the main path, they are optional, that is, you need not choose one of them. In the AUTHORIZATION clause of the module preamble, the username is mandatory, but the password and database to connect to are optional, as shown in this diagram:
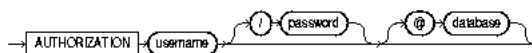
Loops let you repeat the syntax contained within them as many times as you like.

Here is the syntax diagram for a module:
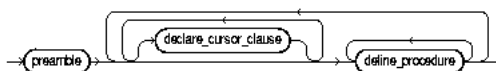
Thus, a module consists of a preamble, followed by zero or more cursor declarations, followed by one or more procedures.

Preamble

The syntax of the preamble is:
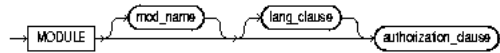
The illustration shows the preamble syntax. The keyword MODULE is mandatory, followed by the optional mod_name, the optional language clause, and the mandatory authorization clause.

****

LANGUAGE Clause

The following diagram shows the syntax of the optional language clause (lang_clause) of the preamble:
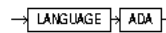
The illustration shows the syntax for the language clause. It contains the mandatory keyword LANGUAGE, followed by the mandatory keyword ADA.

****

AUTHORIZATION Clause

The following diagram shows the syntax of the AUTHORIZATION clause:
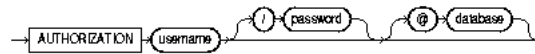
Cursors

The syntax of the cursor declaration is:
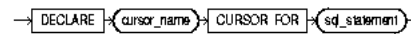
Procedure Definitions

The SQL92 syntax for a procedure definition is:
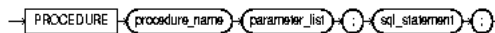
where the parameter list is defined as:
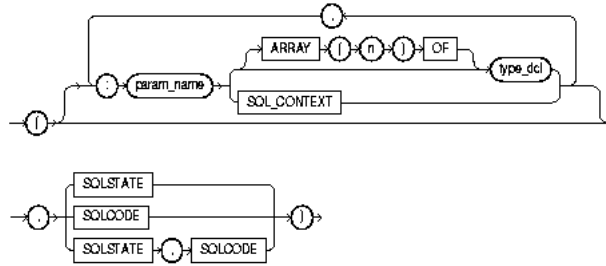
Where type_dcl is defined as:
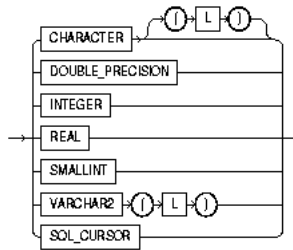
Length L has values: 1< L<32500. n is the size of the array.

You can place the SQLSTATE or the SQLCODE status parameters anywhere in the parameter list; they are conventionally shown at the end of the parameter list in this Guide, as indicated in this syntax diagram. You must include either SQLSTATE (recommended) or SQLCODE (for backward compatibility with SQL89). You can include both, in any order. Do not place a colon before the status parameter, and do not include a datatype after it.

WITH INTERFACE CLAUSE

The syntax of a procedure declaration that includes a WITH INTERFACE clause is formally defined as:
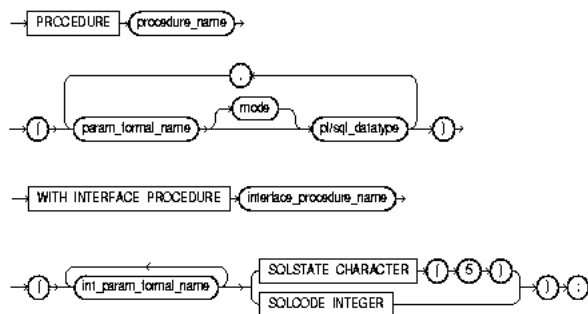
The int_param_formal_name is defined as:
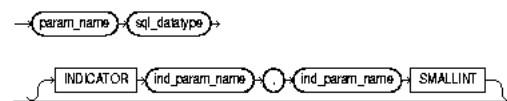
The SQL datatype in the WITH INTERFACE clause must be compatible with the corresponding PL/SQL datatype in the procedure declaration parameter list.

The syntax of the mode attribute is:
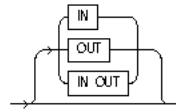
If mode is omitted, the value is IN.

C

Reserved Words

The words listed in this appendix are reserved by Oracle or by SQL*Module. For PL/SQL reserved words, see *Oracle Database PL/SQL Language Reference*.

Module Reserved Words

The following words are reserved by Oracle and by SQL*Module. You cannot use them to name a module, nor to name cursors, procedures, or procedure parameters in a module.

ADA, ARRAY, ALL, ALLOCATE, ALTER, AND, ANGLE_BRK, ANSIC, ANY, AREASIZE, ARRAYLEN, AS, ASC, AT, AUDIT, AUTHORIZATION, AUTO, AVG, BEGIN, BETWEEN, BIND, BREAK, BY, C, CASE, CAT, CHAR, CHARACTER, CLOSE, COBOL, CODE, COMMENT, COMMIT, CONNECT, CONNECTION, CONTEXT, CONST, CONTINUE, CREATE, CURRENT, CURRVAL, CURSOR, DATABASE, DATE, DEC, DECI, DECIMAL, DECLARE, DEFAULT, DEFINE, DEFINED, DELETE, DESC, DESCRIBE, DESCRIPTOR, DISCONNECT, DISPLAY, DISTINCT, DO, DOUBLE, DROP, ELSE, ENABLE, END, ENDEXEC, ENDIF, ENUM, ERRORS, ESCAPE, EXCLUSIVE, EXEC, EXECUTE, EXECORACLE, EXECORACLEELSE, EXECORACLEENDIF, EXECSQL, EXECSQLBEGIN, EXECSQLEND, EXECSQLTYPE, EXECSQLVAR, EXECUTE, EXISTS, EXTERN, FETCH, FIPS, FLOAT, FOR, FORCE, FORTRAN, FOUND, FREE, FROM, FULL, FUNCTION, GET, GO, GOTO, GRANT, GROUP, HAVING, HOLDCURSOR, IAF, IDENTIFIED, IF, IFDEF, IFNDEF, IMMEDIATE, IN, INCLUDE, INCSQL, INDICATOR, INSERT, INT, INTEGER, INTERSECT, INTO, IS, ISOLATION, KRC, LANGUAGE, LEVEL, LIKE, LIMITED, LIST, LOCK, LOCKLONG, MAX, MAXLITERAL, MAXOPENCURSORS, MIN, MINUS, MOD, MODE, MODULE, NEXTVAL, NO, NOAUDIT, NONE, NOT, NOTFOUND, NOWAIT, NULL, NUMBER, NUMERIC, OF, ONLY, OPEN, OPTION, OR, ORACA, ORACLE, ORACLE_C, ORDER, PACKAGE, PASCAL, PLI, PRECISION, PREPARE, PRIOR, PROCEDURE, PUT, RAW, READ, REAL, REBIND, REENTRANT, REFERENCE, REGISTER, RELEASE, RELEASE_CURSOR, REM, RENAME, RETURN, REVOKE, ROLLBACK, ROW, ROWID, ROWNUM, SAVEPOINT, SECTION, SEGMENT, SELECT, SELECTERROR, SEMANTICS, SERIALIZABLE, SET, SHARE, SHORT, SIGNED, SIZEOF, SMALLINT, SOME, SQL, SQL2, SQL89, SQLCHECK, SQLCODE, SQLERRM, SQLERROR, SQLROWS, SQLSTATE, SQLWARNING, SQL_CONTEXT, SQL_CURSOR, START, STATEMENT, STATIC, STDDEV, STOP, STRING, STRUCT, SUM, SWITCH, SYNTAX, SYSDATE, TABLE, THREADS, TO, TRANSACTION, TYPEDEF, UID, UNDEF, UNION, UNIQUE, UNSIGNED, UPDATE, USE, USER, USING, VALIDATE, VALUES, VARCHAR, VARCHAR2, VARIABLES, VARIANCE, VARNUM, VARRAW, VARYING, VOID, VOLATILE, WHEN, WHENEVER, WHERE, WHILE, WITH, WORK, WORKWRITE, WRITE, XOR_EQ, XOR_WQ, YES

D

SQLSTATE Codes

This appendix contains a table of the SQLSTATE codes and the conditions and errors associated with them.

SQLSTATE Codes

Code	Condition	Oracle Error
00000	successful completion	ORA-00000
01000	warning	
01001	cursor operation conflict	
01002	disconnect error	
01003	null value eliminated in set function	
01004	string data - right truncation	
01005	insufficient item descriptor areas	
01006	privilege not revoked	
01007	privilege not granted	
01008	implicit zero-bit padding	
01009	search condition too long for info schema	
0100A	query expression too long for info schema	
02000	no data	ORA-01095
		ORA-01403
07000	dynamic SQL error	
07001	using clause does not match parameter specs	
07002	using clause does not match target specs	
07003	cursor specification cannot be executed	
07004	using clause required for dynamic parameters	
07005	prepared statement not a cursor specification	
07006	restricted datatype attribute violation	
07007	using clause required for result fields	
07008	invalid descriptor count	SQL-02126

Code	Condition	Oracle Error
07009	invalid descriptor index	
08000	connection exception	
08001	SQL client unable to establish SQL connection	
08002	connection name in use	
08003	connection does not exist	SQL-02121
08004	SQL server rejected SQL connection	
08006	connection failure	
08007	transaction resolution unknown	
0A000	feature not supported	ORA-03000 .. 03099
0A001	multiple server transactions	
21000	cardinality violation	ORA-01427
		SQL-02112
22000	data exception	
22001	string data - right truncation	ORA-01401
		ORA-01406
22002	null value - no indicator parameter	ORA-01405
		SQL-02124
22003	numeric value out of range	ORA-01426
		ORA-01438
		ORA-01455
		ORA-01457
22005	error in assignment	
22007	invalid date-time format	
22008	date-time field overflow	ORA-01800 .. 01899
22009	invalid time zone displacement value	
22011	substring error	
22012	division by zero	ORA-01476
22015	interval field overflow	
22018	invalid character value for cast	
22019	invalid escape character	ORA-00911
		ORA-01425
22021	character not in repertoire	
22022	indicator overflow	ORA-01411

Code	Condition	Oracle Error
22023	invalid parameter value	ORA-01025
		ORA-01488
		ORA-04000 .. 04019
22024	unterminated C string	ORA-01479 .. 01480
22025	invalid escape sequence	ORA-01424
22026	string data - length mismatch	
22027	trim error	
23000	integrity constraint violation	ORA-00001
		ORA-02290 .. 02299
24000	invalid cursor state	ORA-01001 .. 01003
		ORA-01410
		ORA-08006
		SQL-02114
		SQL-02117
		SQL-02118
		SQL-02122
25000	invalid transaction state	
26000	invalid SQL statement name	
27000	triggered data change violation	
28000	invalid authorization specification	
2A000	direct SQL syntax error or access rule violation	
2B000	dependent privilege descriptors still exist	
2C000	invalid character set name	
2D000	invalid transaction termination	
2E000	invalid connection name	
33000	invalid SQL descriptor name	
34000	invalid cursor name	
35000	invalid condition number	
37000	dynamic SQL syntax error or access rule violation	
3C000	ambiguous cursor name	
3D000	invalid catalog name	
3F000	invalid schema name	

Code	Condition	Oracle Error
40000	transaction rollback	ORA-02091 .. 02092
40001	serialization failure	
40002	integrity constraint violation	
40003	statement completion unknown	
42000	syntax error or access rule violation	ORA-00022
		ORA-00251
		ORA-00900 .. 00999
		ORA-01031
		ORA-01490 .. 01493
		ORA-01700 .. 01799
		ORA-01900 .. 02099
		ORA-02140 .. 02289
		ORA-02420 .. 02424
		ORA-02450 .. 02499
		ORA-03276 .. 03299
		ORA-04040 .. 04059
		ORA-04070 .. 04099
44000	with check option violation	ORA-01402
60000	system errors	ORA-00370 .. 00429
		ORA-00600 .. 00899
		ORA-06430 .. 06449
		ORA-07200 .. 07999
		ORA-09700 .. 09999

Code	Condition	Oracle Error
61000	resource error	ORA-00018 .. 00035
		ORA-00050 .. 00068
		ORA-02376 .. 02399
		ORA-04020 .. 04039
62000	path name server and detached process errors	ORA-00100 .. 00120
		ORA-00440 .. 00569
63000	Oracle*XA and two-task interface errors	ORA-00150 .. 00159
		SQL-02128
		ORA-02700 .. 02899
		ORA-03100 .. 03199
		ORA-06200 .. 06249
64000	control file, database file, and redo file errors; archival and media recovery errors	ORA-00200 .. 00369
		ORA-01100 .. 01250
65000	PL/SQL errors	ORA-06500 .. 06599
66000	SQL*Net driver errors	ORA-06000 .. 06149
		ORA-06250 .. 06429
		ORA-06600 .. 06999
		ORA-12100 .. 12299
		ORA-12500 .. 12599
67000	licensing errors	ORA-00430 .. 00439
69000	SQL*Connect errors	ORA-00570 .. 00599
		ORA-07000 .. 07199

Code	Condition	Oracle Error
72000	SQL execute phase errors	ORA-01000 .. 01099
		ORA-01400 .. 01489
		ORA-01495 .. 01499
		ORA-01500 .. 01699
		ORA-02400 .. 02419
		ORA-02425 .. 02449
		ORA-04060 .. 04069
		ORA-08000 .. 08190
		ORA-12000 .. 12019
		ORA-12300 .. 12499
		ORA-12700 .. 21999
82100	out of memory (could not allocate)	SQL-02100
82101	inconsistent cursor cache: unit cursor/global cursor mismatch	SQL-02101
82102	inconsistent cursor cache: no global cursor entry	SQL-02102
82103	inconsistent cursor cache: out of range cursor cache reference	SQL-02103
82104	inconsistent host cache: no cursor cache available	SQL-02104
82105	inconsistent cursor cache: global cursor not found	SQL-02105
82106	inconsistent cursor cache: invalid Oracle cursor number	SQL-02106
82107	program too old for runtime library	SQL-02107
82108	invalid descriptor passed to runtime library	SQL-02108
82109	inconsistent host cache: host reference is out of range	SQL-02109
82110	inconsistent host cache: invalid host cache entry type	SQL-02110
82111	heap consistency error	SQL-02111
82112	unable to open message file	SQL-02113
82113	code generation internal consistency failed	SQL-02115
82114	reentrant code generator gave invalid context	SQL-02116
82115	invalid hstdef argument	SQL-02119

Code	Condition	Oracle Error
82116	first and second arguments to sqlrcn both null	SQL-02120
82117	invalid OPEN or PREPARE for this connection	SQL-02122
82118	application context not found	SQL-02123
82119	connect error; can't get error text	SQL-02125
82120	precompiler/SQLLIB version mismatch.	SQL-02127
82121	FETCHed number of bytes is odd	SQL-02129
82122	EXEC TOOLS interface is not available	SQL-02130
90000	debug events	ORA-10000 .. 10999
99999	catch all	all others
HZ000	remote database access	

E

System-Specific References

This appendix contains a complete list of the features of the SQL*Module compiler and its libraries that are system specific.

System-Specific Aspects of SQL*Module

This section describes the system-specific aspects of the SQL*Module compiler features and their libraries that are system specific.

Supported Compilers

The Ada compiler that you can use to compile the code generated by SQL*Module is platform specific. See your system-dependent documentation.

Character Case in Command Lines

Operating systems differ in case sensitivity. See "Case of Package and Procedure Names" on page 3-4 for more information.

Location of Files

The location in the directory hierarchy of the SQL*Module executable, the system configuration file, and the SQLLIB and OCILIB libraries can differ from system to system. See "Invoking SQL*Module" on page 5-3 and "Configuration Files" on page 5-6.

Filename Extensions

The default filename extensions that SQL*Module generates for output files might vary from platform to platform. See "Output Files" on page 5-6.

Ada Output Files

The default filenames and filename extensions for Ada depend on the supported compiler for your system. See "Source Code Output File" on page 5-7 and "Listing File" on page 5-7.

Command Line

The command line interpreter makes assumptions about defaults that are system specific. See your system-specific Oracle documentation.

Ada SQL_STANDARD Package

The SQL_STANDARD package that is shipped with the Ada version of SQL*Module can differ from system to system, depending on the requirements of your Ada compiler. See your system-specific Oracle documentation.

Index