Forms™ Advanced Techniques

Release 4.5

Forms™ Advanced Techniques

Release 4.5

Part No. A32506–2

ORACLE®

Forms Advanced Techniques, Release 4.5

Part No. A32506–2

Contributing Authors: Ken Chu, Gina Lim
Contributors: Mark Clark, David Navas, Eric Newman, David Walker

Preface

The *Forms Advanced Techniques Manual* provides information necessary to help you use Forms 4.5. This preface includes the following topics:

- Forms Documentation Set
- Audience
- Related Publications
- Your Comments Are Welcome

Forms Documentation Set

The documentation set for Forms Version 4.5 consists of the following documents:

Document	*Part Number*
Forms Documentation Set, Version 4.5	A32503
Getting Started with Forms, Version 4.5	A32504
Forms Developer's Guide, Version 4.5	A32505
Forms Advanced Techniques, Version 4.5	A32506
Forms Reference Manual, Version 4.5, Vol. 1 and Vol. 2	A32507
Forms Messages and Codes, Version 4.5	A32508

Audience

All the manuals in the Forms Version 4.5 documentation set are written for the application developer.

Related Publications

As an application designer using Version 4.5 of Forms, you should also be familiar with the following documents:

Document	*Part Number*
Procedure Builder Developer's Guide	A32485
Oracle Terminal User's Guide, Version 2.0	A11700
Oracle7 Server Messages and Codes Manual	A12379
Oracle7 Server SQL Language Reference Manual, Version 7.0	778–70–1292
PL/SQL User's Guide and Reference, Version 2.0	800–20–1292
Forms documentation for your operating system	

Your Comments Are Welcome

We value and appreciate your comments as an Oracle user and reader of the manuals. As we write, revise, and evaluate our documentation, your opinions are the most important input we receive. At the back of our printed manuals is a Reader's Comment Form, which we encourage you to use to tell us what you like and dislike about this manual or other Oracle manuals. If the form is not available, please use the following address or FAX number.

Forms Documentation Manager
Oracle Corporation
500 Oracle Parkway
Redwood City, CA 94065
U.S.A.
FAX: 415–506–7200

Contents

CHAPTER

1

Handling Runtime Errors

This chapter describes error–handling techniques for code that you write in form, menu, and library modules. It includes the following topics:

Handling Runtime Errors in Triggers

When you compile a trigger at design time or generate a form module, the compiler detects any errors that would prevent the trigger from executing successfully at runtime. Other errors, however, manifest only at runtime, and are often impossible to avoid.

When a runtime error occurs, you need to detect and respond to the error in your trigger code so it does not interrupt the flow of the application. Triggers should be written to handle runtime errors gracefully, rather than allowing them to disrupt the work of the form operator.

Oracle Forms default functionality helps you handle runtime errors by detecting errors as they occur, issuing error messages, setting the values of error variables, and, when appropriate, rolling back transactions that cannot be completed successfully. You can supplement default error handling by writing triggers that respond to errors in an application–specific manner.

Note: This section assumes you are familiar with PL / SQL error handling techniques. Refer to the *PL/SQL User's Guide and Reference* for more information.

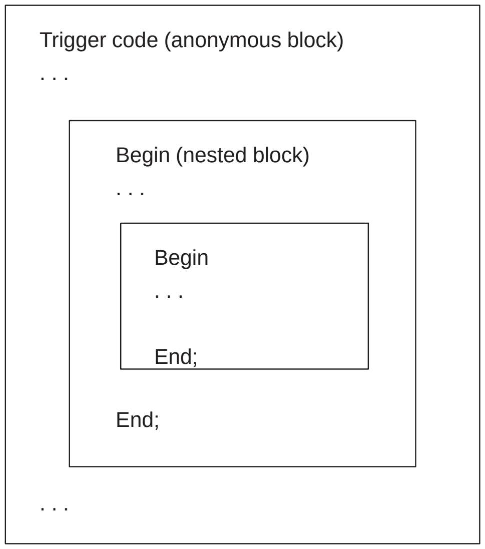
Using PL/SQL Exception Handling in Triggers

Errors in PL / SQL code are called *exceptions.* When an error occurs, an exception is *raised.* To handle raised exceptions, you can write *exception handlers.* In Oracle Forms, you can use exception handlers to handle runtime errors that occur in the following types of statements:

- SQL statements
- PL / SQL statements
- calls to user–named subprograms

When one of these statements in a trigger or PL / SQL block raises an exception, normal processing stops and control transfers to the trigger's exception handling section.

How Exceptions Propagate in Triggers PL / SQL blocks can be nested as in–line blocks in Oracle Forms triggers. For example, a trigger might include a block that encloses another block.



Nested PL/SQL Blocks within a Trigger

When the inner block raises an exception for which there is no exception handler, that block terminates and the exception propagates, or ”falls out,” to the enclosing block. If a handler is not found in the enclosing block, the exception propagates again, finally reaching the outermost block of the trigger. If the outer block does not handle the exception, the trigger fails, and Oracle Forms returns a runtime error.

Exceptions Raised in User–Named Subprograms For purposes of exception handling, calls to user–named subprograms are treated as in–line PL / SQL blocks within a trigger. That is, an unhandled exception raised in a user–named subprogram propagates to the block in which the subprogram was called.

Results of Trigger Failure

Once a trigger fires, it can end with either success or failure. A trigger fails when it raises an unhandled exception. When this occurs, Oracle Forms aborts the trigger and performs the appropriate post–failure processing. The specific processing that Oracle Forms performs depends on the type of trigger that failed.

The following table shows the post–failure processing for three types of triggers:

Trigger Type	*Result of Trigger Failure*
Key – * (any key trigger)	Trigger failure is ignored.
Pre–Item	Oracle Forms attempts to return input focus to the source item.
On–Insert	Oracle Forms rolls back all database changes posted by the current commit process and attempts to navigate to the first item in the current record of the block in error.
When–Validate–Item	Item validation fails. Oracle Forms sets the error location. When possible, input focus is returned to the source item.

When a trigger fails, Oracle Forms tries to undo any conditions that would cause a runtime error as a result of the trigger's failure.

For example, the Pre–Item trigger fires when Oracle Forms attempts to move the input focus from a source item to a target item. When a Pre–Item trigger fails, navigation cannot be completed successfully, so Oracle Forms returns the input focus to the source item.

Similarly, when a commit processing trigger such as On–Insert fails, Oracle Forms performs the appropriate rollback to ensure data integrity.

In the trigger descriptions in the online Help and in Chapter 2 of the *Oracle Forms Reference Manual*, each trigger has an "On Failure:" entry that describes its failure behavior.

Handling Exceptions Raised in Triggers

When you write triggers that include SQL statements, PL/SQL statements, or calls to user–named subprograms, you should consider how you will handle any exceptions that are raised. Depending on the behavior desired, you can respond to exceptions in one of three ways:

- Write exception handlers in the trigger to handle all raised exceptions.
- Take advantage of Oracle Forms default post–failure trigger processing by deliberately omitting exception handlers; when an exception is raised, the trigger fails, and the appropriate post–failure processing occurs.
- Write exception handlers for only the specific exceptions you want to handle in the trigger, and leave any other exceptions unhandled to deliberately cause the trigger to fail.

Responding to Errors

When an error occurs, you can use the following built–in subprograms to get information about the error, including the error number, type, and message:

- ERROR_TYPE (returns CHAR)
- ERROR_CODE (returns NUMBER)
- ERROR_TEXT (returns CHAR)
- DBMS_ERROR_CODE (returns NUMBER)
- DBMS_ERROR_TEXT (returns CHAR)

Evaluating the Success or Failure of Built–ins

A built–in subprogram can have one of three outcomes: success, failure, or fatal error. When a built–in subprogram fails, a runtime error occurs, and Oracle Forms responds by issuing the appropriate error message. However, no exception is raised in the trigger. For this reason, the trigger itself does not fail, and any subsequent statements in the trigger are executed.

When you call built–in subprograms in triggers, you will often want to trap the success or failure of the subprogram so that you can either correct any errors or cause the trigger to fail explicitly. This is particularly true when subsequent trigger statements depend on the successful outcome of a preceding built–in subprogram.

To trap the success or failure of a built–in subprogram, use the following Oracle Forms built–in functions:

- FORM_SUCCESS (returns BOOLEAN)
- FORM_FAILURE (returns BOOLEAN)
- FORM_FATAL (returns BOOLEAN)

These functions report on the outcome of the most recently executed built–in subprogram. In the following When–Button–Pressed trigger, the function FORM_SUCCESS is used to test the outcome of the GO_BLOCK built–in procedure:

```
Go_Block('xyz_block');
    IF NOT Form_Success THEN
      ––handle the error
    END IF;
```

This test detects both fatal and failure–type errors, and so is more encompassing than FORM_FAILURE or FORM_FATAL used alone.

Handling Errors in Built–in Subprograms

When a built–in subprogram fails or causes a fatal error, you might want to direct the trigger from which the subprogram was called to fail, rather than allowing Oracle Forms to continue processing subsequent trigger statements.

In a previous topic, you saw that a trigger fails when it raises an unhandled exception. However, because errors in built–in subprograms do not raise exceptions, you must raise an exception explicitly in your code to cause a trigger to fail:

```
/*  When–Button–Pressed Trigger:  */

Go_Block('xyz_block');
   IF NOT Form_Success THEN
     RAISE Form_Trigger_Failure;
   END IF;
END IF;
```

Because this trigger does not have an exception handling section, raising an exception causes the trigger to fail, and Oracle Forms performs the post–failure processing appropriate for the trigger. Although any valid exception can be raised to cause a trigger to fail, the example uses the built–in exception FORM_TRIGGER_FAILURE.

Raising the FORM_TRIGGER_FAILURE Exception

The FORM_TRIGGER_FAILURE exception is a predefined PL/SQL exception available only in Oracle Forms. Because it is predefined, you can raise this exception without having to first define it in the declarative section of a trigger or user–named subprogram.

Indeed, this exception is used so often in Oracle Forms that it is common to write a user–named procedure that can be used to test the outcome of built–in procedures and functions:

```
/*  user–named subprogram:  */

PROCEDURE Check_Builtin IS
BEGIN
   IF NOT Form_Success THEN
     RAISE Form_Trigger_Failure;
   END IF;
END;
```

You can define the procedure in a form or an attached library and then call it from any trigger in your form:

```
/*  When–Button–Pressed Trigger:  */

Go_Block('xyz_block');
Check_Builtin;
```

Note: Do not use the internal error code associated with the FORM_TRIGGER_FAILURE exception, because the internal error code can change without notice.

Handling Errors in User–Named Triggers

User–named triggers are invoked by calling the EXECUTE_TRIGGER built–in subprogram:

```
/*  Built-in Trigger:  */

statement a;
Execute_Trigger('my_user_named_trigger');
statement b;
```

When an unhandled exception is raised in a user–named trigger, the user–named trigger fails, but the exception does not propagate to the calling trigger. Rather, Oracle Forms treats the failure as an error in the built–in procedure EXECUTE_TRIGGER, and sets the return values of the built–in error functions accordingly. Thus, the outcome of a user–named trigger can be trapped in the same way as a call to any other built–in subprogram; that is, by evaluating the built–in error functions:

```
/*  Built-in Trigger:  */

statement a;
Execute_Trigger('my_usernamed_trigger');
IF NOT Form_Success THEN
   RAISE Form_Trigger_Failure;
END IF;
statement b;
```

Error Handling for Stored Procedures

There are three primary methods for trapping ORACLE errors that are returned from the kernel during the processing of your PL / SQL code:

- checking DBMS_ERROR_TEXT and DBMS_ERROR_CODE built–in subprograms within a form–level ON–ERROR trigger
- creating appropriate user–defined exceptions
- evaluating the SQLCODE and SQLERRM functions in a WHEN OTHERS exception handler

Checking DBMS_ERROR_TEXT, DBMS_ERROR_CODE

From within an ON–ERROR trigger, you can check to see if the ERROR_CODE function reports any of the following database–related errors:

```
40501: ORACLE err – unable to reserve record for update or delete
40502: ORACLE err – unable to read list of values
40505: ORACLE err – unable to perform query
40506: ORACLE err – unable to check for record uniqueness
40507: ORACLE err – unable to fetch next query record
40508: ORACLE err – unable to INSERT record
40509: ORACLE err – unable to UPDATE record
40510: ORACLE err – unable to DELETE record
40512: ORACLE err – unable to issue SAVEPOINT command
40513: ORACLE err – unable to get date/time from database
40504: ORACLE err – unable to execute a gname trigger
40511: ORACLE err occurred while executing a gname trigger
```

Once you know that some database error has caused form processing to fail, you can interrogate the DBMS_ERROR_TEXT and DBMS_ERROR_CODE functions to determine exactly what server error has occurred. The full text of the error message is available in the return value for DBMS_ERROR_TEXT.

User–Defined Exceptions

Although PL / SQL includes many pre–defined exceptions—such as NO_DATA_FOUND, DUP_VAL_ON_INDEX, and VALUE_ERROR—they will never completely cover the range of ORACLE errors you may need to trap. So PL / SQL provides the facility to define your own exceptions and associate them with the occurrence of a particular Oracle error of your choice. The following code illustrates how to use EXCEPTION_INIT to tell PL / SQL to report an error of your choice. For more information refer to the *PL/SQL User's Guide and Reference, Version 2.0.*

Example:

```
/*
**  Example of declaring your own error–driven exceptions
*/
DECLARE
   /*
   ** First declare the name of the exception
   */
   cannot_del_parent  EXCEPTION;

   /*
   ** Then associate it with the ORA–2292 error
   ** which is "violated integrity constraint XYZ –
   ** child record found". Note error number is negative.
   */
   PRAGMA Exception_Init (cannot_del_parent, –2292);

BEGIN
   DELETE FROM PARENT_TABLE
   WHERE PRIMARY_KEY_FIELD = :BLOCK.PK;
   /*
   ** If we get here, then things went ok.
   */

EXCEPTION
   /*
   ** If our error arises, then this exception
   ** will be raised. We can deal with it elegantly.
   */
   WHEN cannot_del_parent THEN
      Message('You cannot remove open '||
      'order number'||:block.pk);
      RAISE Form_Trigger_Failure;
END;
```

This method is best when the ORACLE error number itself is enough to allow your application to determine what happened. User–defined error messages can be returned from database triggers, procedures, or functions, as shown earlier with RAISE_APPLICATION_ERROR. Creating corresponding user–defined exceptions is a natural counterpart to trapping the errors you raise.

However, some errors returned by the kernel contain the name of the constraint (out of many possible ones) that has been violated, always returning a single ORACLE error number. An example of this would be:

```
ORA–02290: violated check constraint (SCOTT.C_CK)
```

Trapping SQLCODE and SQLERRM

In this case, we need access to the error message itself to gain knowledge about what went wrong. The WHEN OTHERS clause must be used so the SQLCODE and SQLERRM can be captured and evaluated. These two PL/SQL functions, which mimic their PRO*Language analogs, are only available within an exception handler, and are most useful in a WHEN OTHERS clause. In addition, the function called *strip_constraint_name* will accept the text of an Oracle error and return in capital letters the name of the constraint that was violated. Consider two examples:

Example 1:

```
/*
**  Example of using SQLCODE/SQLERRM in WHEN OTHERS */
*/
DECLARE
   lv_sqlcode NUMBER; /* Place to hold SQLCODE */
   lv_sqlerrm VARCHAR2(240); /* Place to hold SQLERRM */
   lv_constr  VARCHAR2(41); /* Place for Constraint Name */
BEGIN

   UPDATE PARENT_TABLE
      SET SOME_FIELD = 5
   WHERE PRIMARY_KEY_FIELD = :BLOCK.PK;

   /*
   ** If we get here, then things went ok.
   */

EXCEPTION
  /*
   ** If an error arises, the exception handler gets control
   */
   WHEN OTHERS THEN
      lv_sqlcode := SQLCODE;
      lv_sqlerrm := SQLERRM;
      IF (lv_sqlcode = –2290) THEN
         /*
         ** Strip out the name of the violated constraint
         */
         lv_constr := strip_constraint_name(lv_sqlerrm);

         IF (lv_constr = ’SCOTT.C_CK’) THEN
                        Message(’Code must be A,B, or C’);
                        RAISE Form_Trigger_Failure;
         END IF;
      END IF;
END;
```

Example 2:

```
/*      STRIP_CONSTRAINT_NAME:  Returns constraint name from error
**  Constraint name should appear enclosed by parentheses
**  in the Oracle errors 02290–02292 and 02296–02299.
**  Return the text between the parentheses when passed
**  the error message text.
*/
FUNCTION strip_constraint_name( errmsg VARCHAR2 )
RETURN VARCHAR2
IS
  lv_pos1 NUMBER;
  lv_pos2 NUMBER;
BEGIN
  lv_pos1 := INSTR(errmsg, '(');
  lv_pos2 := INSTR(errmsg, ')');
  IF (lv_pos1 = 0 OR lv_pos2 = 0 ) THEN
      RETURN(NULL);
    ELSE
       RETURN(UPPER(SUBSTR(errmsg, lv_pos1+1,
                               lv_pos2–lv_pos1–1)));
   END IF;
END;
```

To trap ORACLE errors that are a result of database block INSERT, UPDATE, and DELETE operations, you must code the respective ON–INSERT, ON–UPDATE, and / or ON–DELETE triggers to actually perform the DML operations so that you can trap the errors.

Errors related to the features discussed above that could be trapped by the first example are:

- After a SET ROLE:

```
ORA–01919: Role ROLENAME does not exist
ORA–01979: Missing or invalid password
```

- After relevant DML:

```
ORA–00001: Duplicate key in index
ORA–01400: Mandatory NOT NULL column missing
ORA–01407: cannot update mandatory (NOT NULL)
          column to NULL
ORA–04088: error during execution of trigger
          (OWNER.TRIGGERNAME)
ORA–04092: trigger may not commit or rollback
```

- On return from a database procedure, or a trigger, or a function:

```
 ORA–06550: PL/SQL Error occurred
 ORA–06501: PROGRAM_ERROR if attempting to run against V6.
 ORA–20000:
 ...
 ...(user–defined error range)
...
ORA–20999:
```

Errors related to the features discussed above that could be trapped by the second example are:

- After INSERT / UPDATE with bad foreign key:

```
 ORA–02291:violated integrity constraint (OWNER.CONSTRAINT)–
          parent key not found
```

- After DELETE without CASCADE when dependent children exist:

```
 ORA–02292:violated integrity constraint (OWNER.CONSTRAINT)–
          child record found
```

- After INSERT / UPDATE:

```
 ORA–02290: violated check constraint (OWNER.CONSTRAINT)
```

- On return from database procedure / trigger / function:

```
 ORA–06550: PL/SQL Error occurred
 ORA–06501: PROGRAM_ERROR if attempting to run against V6.
 ORA–20000:
 ...
       ...   (user–defined error range)

      ...

ORA–20999:
```

CHAPTER

2

Stored Procedures and Database Triggers

This chapter discusses support in the Oracle7 Server for stored procedures and database triggers. The topics include:

About Stored Procedures

Oracle Forms supports application partitioning. When designing applications for deployment against the Oracle7 Server, you can include calls to server–side, stored PL / SQL subprograms (stored procedures and stored functions) directly in the PL / SQL code of your Oracle Forms triggers and user–named routines. This includes the ability to execute procedures and functions defined within a package, as well as the ability to access (drag and drop) any of the subprograms at either the local or remote database server.

Processing within the form is on hold until the stored procedure or function completes execution, so the network and database load must be considered with regard to response time. The first time a user executes a stored procedure or function, the executable code is cached in the Oracle7 SGA—the shared global area. Because the code is cached, subsequent uses of the executable are faster. The first time any subprogram within a package is referenced, the entire package is loaded and becomes shareable.

Use a database procedure instead of an Oracle Forms procedure when:

- The procedure provides standard functionality that other tools should share—such as validation and calculations.
- The procedure performs a significant number of DML operations—perhaps to have them performed in a bundle by the server.
- If new PL / SQL Version 2 features such as TABLES, RECORDS, or TYPES are required—since Oracle Forms Version 4.5 includes PL / SQL Version 1.1.

The executable form .FMX file can be significantly larger if references to stored procedures are in large packages.

Restrictions When Using Stored Procedures

Observe the following restrictions when working with stored procedures:

- Since Oracle Forms creates the *access routine* for each stored subprogram to which your trigger or user–named routine refers, it is *not* possible to write a form containing calls to stored procedures or functions that will generate against both Oracle Version 6.0 and Version 7.0. Once a form includes a reference to at least one stored subprogram, then it must be generated against a Version 7.0 database, otherwise compilation errors result when the names of stored subprograms cannot be resolved by the PL/SQL compiler.
- Character (CHAR or VARCHAR2) values passed in to a stored procedure or function, passed out from a stored procedure, or returned from a stored function may not exceed 2000 characters. Actual character parameters exceeding 2000 characters will be truncated to 2000 before passing to the stored procedure or function. Any OUT character parameters or character return values exceeding 2000 characters will be truncated to a length of 2000. The truncation that may occur on IN, OUT or return character values from a procedure or function is performed without raising the PL/SQL VALUE_ERROR exception, and without raising an Oracle Forms truncation error.
- Calling stored procedures and functions is not supported from a menu PL/SQL context.
- Changes made to stored subprogram definitions may not be usable by the form developer until after re–establishing a connection to the server.

Standard Packages with Oracle7 Server

The following packages are created when the Oracle7 Server kernel is installed:

- dbms_alert
- dbms_ddl
- dbms_describe
- dbms_lock
- dbms_mail
- dbms_output

- dbms_pipe
- dbms_session
- dbms_snapshot
- dbms_standard
- dbms_transaction
- dbms_utility

See the appendix of the *Oracle7 Server Application Developer's Guide* that describes these packages, their contents and usage.

Only the package dbms_standard is a standard extension. This means that its procedure names are inserted into the scope just outside a package or top–level procedure, but before the kernel's package STANDARD.

Public synonyms for supplied packages are created during execution, and EXECUTE privilege is granted to public.

You can invoke procedures included in any of these packages from Oracle Forms by using the syntax <package>.<procedure>; for example:

```
DBMS_SESSION.SET_ROLE('role');
```

For information on the procedures included in these packages refer to the *Oracle7 Application Developer's Guide*.

Creating and Modifying Stored Procedures

You can create, edit, compile, and browse stored procedures directly from the Oracle Forms Designer if you have the appropriate privileges.

Execute Privilege	Allows the user to execute the public subprogram defined in a stored program unit. Public subprograms are listed in the package specification.
Create Privilege	Allows the user to create, modify, and compile stored program units.
Compile Privilege	Allows the user to compile a stored program unit.
Drop Privilege	Allows the user to drop a stored program unit.

To create a stored procedure:

1. In the Navigator, expand the Database Objects node, then select the Stored Procedures node and choose Navigator–>Create.

   The New Program Unit dialog appears.

2. Specify name and type, either procedure, function, package spec or package body.

   The Stored Program Unit Editor appears. For information about using the Stored Program Unit Editor, see "The Stored Program Unit Editor" later in this chapter.

3. In the Stored Program Unit Editor, define and compile the desired program units.

   When you generate a form, menu, or library containing references to stored procedures, functions, or packages, Oracle Forms must perform the following:

   - Ensure that the stored subprogram to which you make reference exists on the server during compilation.
   - Include an intermediate *access routine* for each stored subprogram you reference. There will be one access routine for each procedure, function or package that is called. If you call functions or procedures within a package, they will use the access routine associated with the entire package. The access routines handle the integration between PL/SQL Version 1 and Version 2, and are included in the .FRM file of your generated form. These access routines are not externalized.

To edit a stored procedure:

1. In the Navigator, expand the Database Objects and Stored Procedures nodes.

   Oracle Forms displays any stored procedures that you either own or have privileges to execute, compile, or drop.

2. Double–click on the desired stored procedure.

   The Stored Program Unit Editor is displayed.

3. Modify the stored procedure as desired and then save it.

Stored Program Unit Editor

Using the Stored Program Unit Editor, you can create, edit, and compile the source text for stored procedures.

Stored Program Unit Editor Commands and Fields

Title	The title reflects the owner and name of the currently displayed stored program unit in the form: stored program unit – <owner>.<programunit_name>
New	The New command allows the user to enter the source text for a new program unit. A new program unit is only visible within the Stored Program Unit Editor. New program units are not stored in the database until the first compile operation.
New	The New button is only enabled if the current user has create privileges for stored program units owned by the user currently selected in the Owner combo box.
Save	The Save command synchronizes the source text displayed in the editor with the source text stored in the database and compiles the stored program unit on the server–side.

Revert	The Revert command discards any changes made in the source text pane and restores the stored program unit to its previous state. The restored state is the most recent of the following: • after the last navigation to this stored program unit or its creation with the New button • after the last selection of the Compile button. The Revert button is only enabled when there have been changes to the source text pane since one of the above states.
Drop	The Drop command drops an existing stored program unit or aborts the creation of a new one (after seeking confirmation via an alert). After dropping the stored program unit, the editor displays the preceding stored program unit in the Name combo box if one exists; otherwise, it displays the next stored program unit. If there are no stored program units for the current owner, the editor is empty. The Drop button is enabled only if the stored program unit is new or the current user has drop privileges for the currently displayed stored program unit.
Close	The Close command closes the Stored Program Unit Editor window. If the source text of the currently displayed stored program unit has been modified, the editor presents an Unsaved Changes alert.
Owner Combo Box	The Owner combo box identifies the user who owns the stored program unit currently displayed in the editor. Selecting a new entry in the Owner combo box updates the Name combo box and displays the first stored program unit in the Name list (if one exists).

Name Combo Box	The Name combo box displays the name of the currently selected stored program unit. The Name combo box contains the names of all accessible stored program units owned by the user currently selected in the Owner combo box. A program unit is accessible if the current user has execute privileges on it. Selecting a new entry in the Name combo box updates the editor to display the newly selected stored program unit. If the source text of the current stored program unit has been modified, the editor presents an Unsaved Changes alert.
Source Text Pane	The source text pane displays the source text of the current stored program unit. It is editable only if the current user has create privileges for the stored program unit; otherwise, it is read–only.
Compilation Message Pane	The Compilation Message pane displays any compilation messages associated with the current stored program unit. Clicking on a compilation message moves the text cursor in the source pane to the line and column associated with the message. The Compilation Message pane is hidden when the current stored program unit has no associated error messages.
Status Bar	The Status Bar displays information about the state of the current stored program unit.

Calling Stored Procedures

A database procedure is a PL / SQL block designed to be executed by the server–side PL / SQL engine. It may accept inputs, and may return outputs, neither of which is mandatory. It runs under the security domain (or schema) of the *creator* of the procedure, not the current user.

The current user needs EXECUTE privileges on the procedure to use it. One important difference between PL / SQL procedures for Oracle Forms and procedures for the database is that server–side procedures do not understand references to Oracle Forms bind variables (such as :BLOCK.ITEMNAME, :GLOBAL.VARNAME, or :SYSTEM.CURSOR_ITEM). Any data that procedures need for processing must be passed into the ”black box” by way of parameters of appropriate datatype, or by package variables, or by selecting from tables.

You should structure your Oracle Forms user–named routines to accept inputs and return results in parameters. This will make the eventual migration of the procedure into the database as painless as adding the word CREATE in front of the PROCEDURE declaration (in addition to running the resulting script in SQL*PLUS).

Syntax Call a stored procedure or function from within Oracle Forms exactly as you would invoke a user–named routine:

```
DECLARE
   ld DATE;
   ln NUMBER;
   lv VARCHAR2(30);
BEGIN
/*
** Calling Form–Level Procedure/Function
*/
   forms_procedure_name(ld,ln,lv);
   ld := forms_function_name(ln,lv);

/*
** Calling Database Procedure/Function
*/
   database_procedure_name(ld,ln,lv);
   ld := database_function_name(ln,lv);
END;
```

Supported Datatypes for Parameters/Return Values Stored procedures, functions, and packages are created using Version 2.0 of PL / SQL within the Oracle7 database. Although Oracle Forms Version 4.5 is built on PL / SQL Version 1.1, the following PL / SQL Version 2.0 datatypes are also supported for parameters and as function return values:

- VARCHAR2—maximum of VARCHAR2(2000)
- NUMBER
- DATE
- BOOLEAN

Recall that your stored procedures and functions can be written *internally* using all new PL / SQL Version 2.0 datatypes and functionality. The above restriction on datatypes applies only to the interface that your stored routines have with their Version 1.1 PL / SQL counterparts, namely parameters and function return values.

However, if you reference a package, regardless of *which* procedures you may use in your Oracle Forms triggers or procedures, then *all* of the subprograms in the package must use *only* the supported datatypes above for parameters.

A restriction has been made on referring to stored subprograms that contain parameters defined as:

```
    TABLE%ROWTYPE
```

or

```
    TABLE.COLUMN%TYPE
```

These are, therefore, unsupported datatype specifications even though their expansion may *refer* to a supported data type for a parameter of a stored procedure accessed from Oracle Forms.

An attempt to reference a stored procedure or stored function that uses unsupported parameter or return–value datatypes will result in the failure to recognize the stored subprogram, and an error is reported when the form is generated:

```
PL/SQL error 313 at line xxx, column yyy
’PROCNAME’ not declared in this scope
```

or

```
PL/SQL error 201 at line xxx, column yyy
identifier ’FUNCTNAME’ must be declared.
```

Default Values for Formal Parameters Default values for formal parameters are not supported. However, you can create a stored package that contains:

- overloaded procedure specification in the package spec
- a private implementation using default parameters in the package body

Example: A form could invoke the package procedure 'Test.Test' with zero, one, or two VARCHAR2 arguments. After creating the package spec and body shown in the example below, the form could invoke:

```
   test.test;
   test.test('Hi');
   test.test('Hi', 'There');
```

This example shows the code on the server side:

```
   CREATE PACKAGE Test IS
PROCEDURE Test;                -- Available to Forms4.5 Client
PROCEDURE Test(a VARCHAR2 );   -- Available to Forms4.5 Client
PROCEDURE Test(a VARCHAR2,
               b VARCHAR2);    -- Available to Forms4.5 Client
END Test;
CREATE PACKAGE BODY Test IS
   PROCEDURE Private_Test( a in VARCHAR2 := 'Hello'
       b in VARCHAR2 := 'There')  IS

   BEGIN
      Dbms_Output.Put_Line(a);
      Dbms_Output.Put_Line(b);
   END;

PROCEDURE Test IS
BEGIN
   Private_Test;
END;

   PROCEDURE Test(a VARCHAR2) IS
   BEGIN
      Private_Test (a);
   END;

   PROCEDURE Test(a VARCHAR2 , b VARCHAR2) IS
   BEGIN
      Private_Test (a, b);
   END;

END Test;
```

Supported Constructs When invoking a stored procedure or function, only the following subset of possible usages is supported:

- ProcName(arg1,...,argN)
- FuncName(arg1,...,argN)
- PackName.ProcName(arg1,...,argN)
- PackName.FuncName(arg1,...,argN)

Accessing Subprograms in Another User's Schema To access a subprogram (i.e., procedure or function) in another user's schema or one in a remote database, you must create a synonym to hide the username or Db_Link name from the PL / SQL compiler such that the result takes the form:

- ProcSynonym(arg1,...,argN)
- FuncSynonym(arg1,...,argN)
- PackSynonym.ProcName(arg1,...,argN)
- PackSynonym.FuncName(arg1,...,argN)

You can create synonyms to nickname remote stored programs, hiding the username or Db_Link:

- Subprogram@DbLink
- Package@DbLink
- Package.Subprogram@DbLink
- Schema.Subprogram@DbLink
- Schema.Package
- Schema.Package@DbLink
- Schema.Package.Subprogram@DbLink

Where *subprogram* is either a procedure or a function.

Example: To call the package function 'LIBOWNER.LIB_HR.GET_SSN', you could create a synonym for the LIB_HR package that includes the schema name (the owner name) as follows:

```
CREATE SYNONYM lib_hr_syn
FOR libowner.lib_hr;
```

Then invoke the function from within your form as follows:

```
ss_num := lib_hr_syn.get_ssn(:Emp.Empno);
```

If the package function is at a remote site accessible through a database link named *basel*, for example, then you could create a synonym for the package, including the database link name:

```
CREATE SYNONYM basel_lib_hr_syn
FOR libowner.lib_hr@basel;
```

and invoke the function within your PL/SQL code as:

```
ss_num := basel_lib_hr_syn.get_ssn(:Emp.Empno);
```

Alternately, you could create a synonym for the function itself, hiding both the schema and Db_Link information:

```
CREATE SYNONYM basel_lib_hr_get_ssn_syn
FOR libowner.lib_hr.get_ssn@basel;
```

and invoke the function from Oracle Forms as:

```
ss_num := basel_lib_hr_get_ssn_syn(:Emp.Empno);
```

Of course, any of the synonyms above could have been created as PUBLIC SYNONYMS if appropriate.

Name Resolution When a form is generated, the PL/SQL compiler may encounter the name of an identifier that could be a procedure or function. The PL/SQL compiler uses a precedence mechanism to resolve ambiguities.

If an identifier such as PROCNAME is encountered that has the structure of a procedure or function, the compiler will use the first match found in the following search order:

1. Is it defined within the current PL/SQL block?
2. Is it a standard PL/SQL command?
3. Is it an Oracle Forms built–in subprogram procedure or function?
4. Is it a user–named procedure or function?
5. Is it defined in package DBMS_STANDARD on the serverside?
6. Does the current user have access to any such procedure or function on the server side?

If the answer is ”no” to all of the above, then the compiler signals an error:

```
PL/SQL error 313 at line xxx, column yyy
’PROCNAME’ not declared in this scope
or
PL/SQL error 201 at line xxx, column yyy
identifier ’FUNCTNAME’ must be declared.
```

About Database Triggers

A *database trigger* is nearly identical in concept to the Oracle Forms trigger. The difference lies in the event that causes the trigger to fire and the location where the subsequent code is performed.

Database triggers are PL/SQL blocks that are associated with a given table; they fire upon the execution of UPDATE, INSERT, or DELETE operations against that table. They may fire BEFORE or AFTER each row the operation affects, or each statement. The combinations give a maximum possibility of twelve triggers for any table. While a trigger executes, it runs under the security domain (schema) of its creator, not the current user.

Within the body of the database trigger, your PL/SQL code may refer to both the old and the new values of the columns being affected. For an INSERT, the old values are non–existent, while for a DELETE, the new values do not exist. This makes data validation simple to implement, and auditing changed values extremely easy.

A database trigger can perform complex data verification that could not be feasibly declared as a constraint. If a database trigger fails with an error, the triggering statement (i.e., the INSERT, UPDATE, or DELETE that fired the trigger) is rolled back. In a simple example, you could write the following trigger to prevent updates on the EMP table during weekends, unless the current user exists in a special WEEKEND_UPDATE_OK table, in which case the update is allowed.

Example:

```
CREATE TRIGGER no_weekend_updates
BEFORE UPDATE
   ON EMP
DECLARE
   day_of_week NUMBER(2) := TO_NUMBER(TO_CHAR(SYSDATE,'D'));
   dummy CHAR(1);
BEGIN
   IF (day_of_week in (1,7))  /* Sunday,Saturday */
   THEN
      BEGIN
         SELECT 'X'/* Check Exceptions Table */
                   INTO dummy
                   FROM WEEKEND_UPDATE_OK
         WHERE USERID = USER;
      EXCEPTION
         WHEN NO_DATA_FOUND THEN /*Not Exception*/
                  RAISE_APPLICATION_ERROR(–20011,
                      'Try again on Monday!');
      END;
   END IF;
END;
```

Besides providing arbitrarily complex data validation, database triggers can also be used to perform any cause–and–effect sequence. This makes database triggers particularly well suited for data auditing operations, data replication, and distributed data integrity checking (since constraints cannot reference remote databases). If database triggers are written to supplant Oracle Forms–side triggers (e.g. for table auditing) then the forms–side functionality will have to be disabled to avoid duplicating table operations.

One alternative is to use a package variable as a *flag* to communicate between the form and the database–side triggers (or procedures). In that manner, a decision can be made within the trigger or procedure on whether a particular operation might have already been performed by the Oracle Forms code.

Triggers (as well as stored procedures and functions) raise errors with the RAISE_APPLICATION_ERROR procedure. The RAISE_APPLICATION_ERROR procedure assigns an error number within the special range 20000–20999 and provides an error message. Since it is your PL/SQL database trigger that prepares the error message, the message can vary. Within the Oracle Forms application, these errors can be trapped with the methods mentioned in the next section.

Recall that database triggers fire in response to a DML statement such as INSERT, UPDATE, or DELETE. So during normal Oracle Forms commit–time processing, as Oracle Forms issues INSERT statements to add newly entered records to the database, UPDATE statements to effect changes by the operator to existing records, and DELETE statements to remove records deleted by the operator from the database; database triggers will fire if enabled.

For example, consider the scenario where the following database triggers have been defined and enabled on the EMP table:

- BEFORE DELETE
- BEFORE DELETE FOR EACH ROW
- AFTER DELETE
- AFTER DELETE FOR EACH ROW

When the Oracle Forms operator deletes a record that has been queried, the following sequence of events occurs:

1. Oracle Forms locks the record to be deleted.
2. Operator presses [Commit].
3. Oracle Forms fires the PRE–COMMIT trigger.
4. Oracle Forms fires the PRE–DELETE trigger.
5. Oracle Forms issues a DELETE statement to delete the row.
6. The database fires the BEFORE DELETE trigger.
7. The database fires the BEFORE DELETE FOR EACH ROW trigger.
8. The database deletes the record.
9. The database fires the AFTER DELETE FOR EACH ROW trigger.
10. The Database AFTER DELETE trigger fires.
11. Oracle Forms fires the POST–DELETE trigger.
12. Oracle Forms fires the POST–COMMIT trigger.

Recall that any error raised during Commit processing causes Oracle Forms to roll back the currently committing transaction to the savepoint that was issued when the Commit sequence began. If any Database trigger fails (by raising an unhandled error or calling a RAISE_APPLICATION_ERROR) as a result of a DML statement that Oracle Forms has issued automatically, then an ”Oracle Error Occurred ...” message will appear. An error raised as a result of a DML statement in your PL/SQL trigger, on the other hand, can be handled gracefully, as discussed below. If not handled, the error will produce the message: ”XXX–YYYYYY trigger raised unhandled exception.”

Creating and Editing Database Triggers

You can create, edit, compile, and browse database triggers directly from the Oracle Forms Designer if you have the appropriate privileges.

Execute Privilege	Allows the user to execute the database trigger.
Create Privilege	Allows the user to create, modify, and compile database triggers.
Compile Privilege	Allows the user to compile a database trigger.
Drop Privilege	Allows the user to drop a database trigger.

To create a database trigger:

1. In the Navigator, expand the Database Objects and Tables nodes, then select and expand the desired table. Select the Triggers node and then choose Navigator–>Create.

   The Database Trigger Editor appears. For information about using the Database Trigger Editor, see “The Database Trigger Editor” later in this chapter.

2. In the Database Trigger Editor, define and compile the desired program units.

To edit a database trigger:

1. In the Navigator, expand the Database Objects and Tables nodes, then select and expand the desired table.

   Oracle Forms displays any database triggers associated with the current module.

2. Double–click on the desired database trigger.

   The Database Trigger Editor appears.

3. In the Database Trigger Editor, modify the database trigger as desired and then compile it.

Database Trigger Editor

Table Owner Combo Box	The Table Owner combo box identifies the user who owns the database trigger currently displayed in the editor.
	Selecting a new entry in the Table Owner combo box updates the Table combo box and displays the first database trigger in the Name list (if one exists).
Table Combo Box	The Table combo box displays the tables owned by the current user.
Name Combo Box	The Name combo box displays the name of the currently selected database trigger.
	The Name combo box contains the names of all accessible database triggers owned by the user currently selected in the Owner combo box. A database trigger is accessible if the current user has execute privileges on it.
	Selecting a new entry in the Name combo box updates the editor to display the newly selected database trigger. If the source text of the current database trigger has been modified, the editor presents an Unsaved Changes alert.
Triggering	When defining a database trigger, you can specify the trigger timing, that is, you can specify when the trigger action is to be executed in relation to the triggering statement: before or after the triggering statement.
Statement	A triggering event or statement is the SQL statement that causes a trigger to be fired. A triggering event can be an INSERT, UPDATE, or DELETE statement for a specific table.
For Each Row	When you create a database trigger, you can specify a trigger restriction. A trigger restriction is an option available for triggers that are fired for each row. Its function is to conditionally control the execution of a trigger. A trigger restriction is specified using a WHEN clause.
Trigger Bocy	Allows you to specify the PL/SQL code to be used as the body of the database trigger.

New	Creates a new code object of the same type and scope as the current code object. For example, when the current object is a trigger attached to an item, choosing New displays the Triggers LOV that allows you to create a new trigger attached to the same item. When the current object is a program unit, choosing New invokes the New Program Unit dialog.
Save	Compiles and saves the code in the Source Code field. The compiler detects syntax errors, semantic errors, and references to non–existent objects, procedures, and functions. When you compile successfully, the status line displays the message <Successfully Compiled>.
Revert	Undoes any changes that were made to the source code since the editor was invoked or since the last Apply or Revert command. Revert is disabled when there are no unapplied changes.
Drop	The Drop command drops an existing database trigger or aborts the creation of a new one (after seeking confirmation via an alert). After dropping the database trigger, the editor displays the preceding database trigger in the Name combo box if one exists; otherwise, it displays the next database trigger. If there are no database triggers for the current owner, the editor is empty. The Drop button is enabled only if the database trigger is new or the current user has drop privileges for the currently displayed database trigger.
Close	Closes the Database Trigger Editor.
Clicking a compilation message	Moves the insertion point to the line at which an error was detected in the Source Code field.

Declarative Database Constraints

A *database constraint* is a rule that governs the logical integrity of your data. The rule is declared at table creation time, or can be added (with some restrictions) after the fact to an existing table.

Entity Constraints

Entity constraints can declaratively enforce a column (or ordered group of columns) to be NOT NULL, UNIQUE within the table, have a DEFAULT value if none is specified on INSERT, and / or satisfy a more complex logical condition given in a CHECK clause. These constraints provide intra–table integrity.

Referential Constraints

Referential constraints provide the logical link between a master and a detail table by establishing primary and foreign key relationships. Database constraints protect your data by automatically enforcing at the database level the rules you have declared. If you create a record in a master table and one detail record, for example, the default operation of database constraints restricts updating of the primary key of the master record. Default database constraints also prevent deleting of the master record (unless the ON DELETE CASCADE option is used when declaring all of the foreign key references).

Strategies for Constraint Checking

Prior to Oracle7 Server, nearly all data integrity was provided at the application level—for Oracle Forms, through triggers. The goal of application–side logic is to stop bad data before it happens. Prior to Oracle7 Server, the RDBMS might dutifully accept whatever data was passed along through the application. To understand more about the similarities and differences of the Oracle database versions, refer to the Oracle7 Server *Migration Guide*.

The Oracle7 ability to perform implicit data verification at the kernel level means that current systems that had been coded to guarantee data integrity by the application will in essence be checking the same things twice. To optimize, you might be tempted to strip out all application logic and leave all constraint checking to the database. However, you should weigh the benefits that constraints on both sides provide and determine if checking things twice is too costly.

Database constraints provide the ability to centrally protect the integrity of the data without necessarily coding the logic into each one of the tools. However, the reason that logic was coded into the application in the first place was to give the user immediate feedback on errors, facilitating their rapid correction with online help and appropriate error messages.

No user would appreciate entering a batch of fifty new orders, only to press the [Commit] key and learn from ORACLE that the fourth order violated an integrity constraint.

You will most likely prefer to check the data both at data entry time and at commit time, recognizing that the small overhead of checking is well worth the additional security it provides. Database Constraints checking is more efficient than its application–based counterpart since constraints are processed intelligently by the kernel, completely within the realm (and RAM) of the kernel, and without additional network trips back to the client.

When you upgrade from Version 6 to Oracle7 Server, all constraints that may have been declared in Version 6 are initially disabled by default. If you want to take advantage of database constraints, be aware that all of the enabled constraints will be checked for each record that is INSERTED, UPDATED, and DELETED (as appropriate). In particular, newly enabled constraints can suddenly return new errors to your application as they reject invalid data. The developer may wish to add extra error handling into the current application to handle the CONSTRAINT violations that may arise.

Note that if a column has been declared to have a DEFAULT value, the kernel will assign the DEFAULT value only when the record is INSERTed, provided that the given column is not listed among the columns in the INSERT clause. Because Oracle Forms 4.5 always lists ALL of the columns (i.e., database fields) in a block for INSERTS and UPDATES, the DEFAULT will never automatically be assigned unless the user either removes the field in question from the block (or marks it as non–database), or else codes an ON–INSERT trigger to override the normal Oracle Forms insert processing.

Also, be aware that some forms–based operations may become unnecessary when constraints are enabled. For example, deleting detail records when a corresponding master is removed, is internally handled by the kernel if the ON DELETE CASCADE option is specified for the parent–child foreign key relationship.

The Integrity Constraints option in the Oracle Forms New Block window continues to function under Oracle7 as it did in V6.

Master/Detail Blocks and Referential Integrity

Applications that include master–detail blocks coordinated via the automatically generated triggers and procedures in SQL*Forms 3.0 should require no modification when running against tables with Declarative Referential Integrity constraints enabled. Oracle Forms default logic will prevent the deletion of a master record when outstanding detail records exist, unless the design specified the Cascading Deletes option when creating the detail block. In that case, Oracle Forms uses a PRE–DELETE trigger to *first* delete the detail records, then deletes the master.

The order of operations was not critical running under ORACLE Version 6, but an application that had moved the standard PRE–DELETE logic to the POST–DELETE trigger instead would encounter problems running against Oracle7 Server when Primary / Foreign–Key relationships have been declared and enabled on the server–side.

Waiting until the POST–DELETE trigger to delete the detail records will cause Oracle Forms to issue the DELETE statement for the master record while detail records remain, and an error will be generated by the kernel:

```
ORA–02292: violated integrity constraint (OWNER.CONSTRAINT)
      –– detail record found
```

However, if the foreign key relationship is specified with the ON DELETE CASCADE option, then no problem will arise, and the Oracle Forms POST–DELETE trigger will be needlessly performing an extra DELETE statement to remove the detail records that the Cascade Delete of the master already deleted.

So potentially any INSERT, UPDATE, or DELETE in your applications could generate an error caused by violating an enabled constraint. To add more sophisticated error handling to your existing forms (in the wake of having enabled many new server–side constraints) see the Error Handling Chapter.

CHAPTER

3

User Exit Interface to Foreign Functions

This chapter examines the user exit interface for invoking foreign functions. The topics covered in this chapter include:

About the User Exit Interface

Foreign functions are subprograms written in a 3GL programming language that allow you to customize your Oracle Forms applications to meet the unique requirements of your users. Foreign functions are often used to enhance performance or provide additional functionality to Oracle Forms.

In Oracle Forms, you can invoke a foreign function from a user exit interface. A user exit interface allows you to call a foreign function by using the USER_EXIT built–in from a trigger or a user–named subprogram. Invoking a foreign function from the USER_EXIT built–in returns an integer value to Oracle Forms indicating success, failure, or a fatal error. Following the execution of the USER_EXIT built–in, the values of the error variables in Oracle Forms—FORM_FAILURE, FORM_FATAL, and FORM_SUCCESS—are set accordingly.

Foreign functions that you invoke from a user exit interface are contained in an Oracle Forms dynamic link library or linked with Oracle Forms Runform. Creating a user exit interface to a foreign function requires you to link additional files to Oracle Forms dynamic link libraries or Oracle Forms Runform. The additional files provide information about the user exit interfaces and the entry points that allow Oracle Forms to invoke foreign functions from a user exit interface.

The implementation of a user exit interface to foreign functions in Microsoft Windows is covered in the section “A User Exit Interface to Foreign Functions on Microsoft Windows.” For information on implementing a user exit interface to foreign functions in other environments, refer to the Oracle Forms documentation for your operating system.

Note: An alternative approach for calling a foreign function is from a PL / SQL interface. The ORA_FFI built–in package provides a PL / SQL interface for invoking foreign functions from Oracle Forms. For more information on the ORA_FFI built–in package, refer to chapter 13, “PL / SQL Interface to Foreign Functions.”

About Foreign Functions

Foreign functions are subprograms written in a 3GL programming language for customizing Oracle Forms applications. Foreign functions can interact with Oracle databases, and Oracle Forms variables and items. Although it is possible to access Oracle Forms variables and items, you cannot call Oracle Forms built–in subprograms from a foreign function.

Foreign functions can be used to perform the following tasks:

- Replace default Oracle Forms processing when running against a non–Oracle data source using transactional triggers.
- Perform complex data manipulation.
- Pass data to Oracle Forms from operating system text files.
- Manipulate LONG RAW data.
- Pass entire PL / SQL blocks for processing by the server.
- Control real time devices, such as a printer or a robot.

  Note: You should not perform host language screen I / O from a foreign function. This restriction exists because the runtime routines that a host language uses to perform screen I / O conflict with the routines that Oracle Forms uses to perform screen I / O. However, you can perform host language *file* I / O from a foreign function.

Types of Foreign Functions

You can develop the following types of foreign functions:

- Oracle Precompiler foreign functions
- OCI (ORACLE Call Interface) foreign functions
- non–ORACLE foreign functions

You can also develop foreign functions that combine both the ORACLE Precompiler interface and the OCI.

Oracle Precompiler Foreign Functions An Oracle Precompiler foreign function incorporates the Oracle Precompiler interface. This interface allows you to write a subprogram in one of the following supported host languages with embedded SQL commands:

- Ada
- C
- COBOL
- FORTRAN
- Pascal
- PL/I

Note: Not all operating systems support all of the listed languages. For more information on supported languages, refer to the Oracle Forms documentation for your operating system.

With embedded SQL commands, an Oracle Precompiler foreign function can access Oracle databases as well as Oracle Forms variables and items. You can access Oracle Forms variables and items by using a set of Oracle Precompiler statements that provide this capability.

Because of the capability to access both Oracle databases and Oracle Forms variables and items, most of your foreign functions will be Oracle Precompiler foreign functions. For more information on the Oracle Precompiler interface, refer to the *Programmer's Guide to the Oracle Precompilers*.

Oracle Call Interface (OCI) Foreign Functions An OCI foreign function incorporates the Oracle Call Interface. This interface allows you to write a subprogram that contains calls to Oracle databases. A foreign function that incorporates only the OCI (and not the Oracle Precompiler interface) cannot access Oracle Forms variables and items. For more information on the OCI, refer to the *Programmer's Guide to the Oracle Call Interface*.

Non–Oracle Foreign Functions A non–Oracle foreign function does not incorporate either the Oracle Precompiler interface or the OCI. For example, a non–Oracle foreign function might be written entirely in the C language. A non–Oracle foreign function cannot access Oracle databases, or Oracle Forms variables and items.

Oracle Precompiler Statements

All Oracle Precompiler foreign functions can use host language statements to perform procedural operations. Precompiler foreign functions can also use the following types of statements to perform additional functions such as accessing the database and manipulating Oracle Forms variables and items.

Statement	*Use*
EXEC SQL	Performs SQL commands.
EXEC TOOLS GET	Retrieves values from Oracle Forms to a foreign function.
EXEC TOOLS SET	Sends values from a foreign function to Oracle Forms.
EXEC TOOLS MESSAGE	Passes a message from a foreign function to display in Oracle Forms.
EXEC TOOLS GET CONTEXT	Obtains context information previously saved in a foreign function.
EXEC TOOLS SET CONTEXT	Saves context information from one foreign function for use in subsequent foreign function invocations.
EXEC ORACLE	Executes Oracle Precompiler options.

An Oracle Precompiler foreign function source file includes host programming language statements and Oracle Precompiler statements with embedded SQL statements. Precompiling an Oracle Precompiler foreign function replaces the embedded SQL statements with equivalent host programming language statements. After precompiling, you have a source file that you can compile with a host language compiler. For more information on a specific precompiler, refer to the appropriate precompiler documentation for your environment.

EXEC SQL Statement

An EXEC SQL statement is a SQL command prefixed with "EXEC SQL." EXEC SQL statements allow you to perform any SQL command in an Oracle Precompiler foreign function. Use EXEC SQL statements to select or manipulate data in the database from a foreign function.

Syntax: `EXEC SQL` *`sql_statement`*`;`

where *`sql_statement`* is any valid Oracle SQL statement, except for the restricted commands noted in this section.

You do not need to perform an explicit CONNECT in an Oracle Precompiler foreign function because Oracle Forms establishes the connection automatically. However, Oracle Server does support concurrent connects. For more information, refer to the *Programmer's Guide to the Oracle Precompilers*.

Restrictions: You adhere to the following restrictions when you use SQL commands in an Oracle Precompiler foreign function:

- Do not issue a SQL COMMIT or ROLLBACK statement from within a foreign function if there are changes in a form that have not been posted to the database when the foreign function is called.
- Do not issue any command that would implicitly cause a database commit, such as a DDL command within a foreign function, if there are changes in a form that have not been posted to the database when the foreign function is called.

EXEC TOOLS GET Statement

An EXEC TOOLS GET statement retrieves a value from Oracle Forms into an Oracle Precompiler foreign function. Specifically, it places the value of an Oracle Forms item or variable into a host language variable. Once the foreign function retrieves a value from Oracle Forms, the foreign function can use that value for calculation, manipulation, or updating.

Syntax:

```
EXEC TOOLS GET form_variable_1[, form_variable_2, ...]
   INTO :host_variable_1[, :host_variable_2, ...];
```

where:

form_variable_n	Specifies the name of the Oracle Forms item or variable from which you are reading a value.
host_variable_n	Specifies the name of the host language variable into which you are reading a value.

Notes: The *form_variable* can be a reference to any of the following items:

- a fully–qualified item *(block.item)*
- a form parameter
- an Oracle Forms system variable
- an Oracle Forms global variable
- a host language variable (prefixed with a colon) whose value is any of the above choices

Refer to the *Programmer's Guide to the Oracle Precompilers* for any restrictions on host language variables.

Restrictions: It is not possible to get or set values directly into a record group from a foreign function.

Example:

```
/*
** Example:  Read an item name from a block (empblock.empname)
*/
EXEC SQL BEGIN DECLARE SECTION;
char itm_buff[255];    /* buffer for item value */
VARCHAR itm_name[255]; /* Forms item name       */
EXEC SQL END DECLARE SECTION;

strcpy(itm_name.arr,"EMBLOCK.EMPNAME");
itm_name.len=strlen("EMBLOCK.EMPNAME");
EXEC TOOLS GET :itm_name
INTO :itm_buff;
```

EXEC TOOLS SET Statement

An EXEC TOOLS SET statement sends a value from an Oracle Precompiler foreign function to Oracle Forms. Specifically, it places the value of a constant or the value of a host language variable into an Oracle Forms item or variable.

Any value that an EXEC TOOLS SET statement passes to a form item displays after the foreign function returns processing control to the form that called the foreign function (providing, of course, that the item has the Displayed item property set to True).

Syntax:

```
EXEC TOOLS SET form_variable[, ...]
    VALUES ({:host_variable | constant}[, ...]);
```

where:

form_variable	Specifies the name of the Oracle Forms item or variable into which you are reading a value.
host_variable	Specifies the name of the host language variable from which you are reading a value.
constant	Specifies the constant that you are reading. Do not precede a constant with a colon.

Notes: The *form_variable* can be a reference to any of the following items:

- a fully–qualified item *(block.item)*
- a form parameter
- an Oracle Forms system variable
- an Oracle Forms global variable
- a host language variable (prefixed with a colon) whose value is any of the above choices

Refer to the *Programmer's Guide to the Oracle Precompilers* for any restrictions on host language variables.

Represent host variables and constants in standard SQL format:

Value	*Result*
:holder1	Inserts the value of the host variable, holder1 (preceded by a semi–colon).
'Summit Sporting Goods'	Inserts the constant string value, Summit Sporting Goods (enclosed in single quotes).
413	Inserts the constant numeric value, 413 (no quotes for numeric values).

Example:

```
/*
** Example:  Write ’SMITH’ into emp.ename
*/
EXEC SQL BEGIN DECLARE SECTION;
char itm_buff[255];  /*buffer for item value */
VARCHAR itm_name[255]; /* Forms item name */
EXEC SQL END DECLARE SECTION;

strcpy(itm_name.arr,”EMP.ENAME”);
itm_name.len = strlen(”EMP.ENAME”);
strcpy(itm_buff,”SMITH”);
EXEC TOOLS SET :itm_name
VALUES (:itm_buff);
```

EXEC TOOLS MESSAGE Statement

An EXEC TOOLS MESSAGE statement displays a message on the Oracle Forms message line.

Syntax: `EXEC TOOLS MESSAGE (`*`message [severity]);`*

where:

message	Specifies the message you are passing to Oracle Forms. The message can be a quoted string or a host language variable.
severity	Specifies the severity level of the message you are passing to Oracle Forms.

Example:

```
/*
** Example:  Error message for text item
*/
EXEC TOOLS MESSAGE
   ’Incorrect argument: Expecting item name. Please re–enter.’;
```

EXEC TOOLS GET CONTEXT Statement

An EXEC TOOLS GET CONTEXT statement obtains context information (a pointer name) previously saved using EXEC TOOLS SET CONTEXT and reads it into a foreign function.

Syntax:

```
EXEC TOOLS GET CONTEXT context_name [, ...]
   INTO :host_variable_1[, :host_variable_2, ...];
```

where:

context_name	Specifies the name of the context you are saving.
host_variable_n	Specifies the name of the host language variable into which you are reading the context value.

Notes: The *context_name* can be a reference to one of the following items:

- a host language variable (prefixed with a colon)
- a constant

Example:

```
/*
** Example:  Get previously saved context information
*/
EXEC SQL BEGIN DECLARE SECTION;
char *pctx;
EXEC SQL END DECLARE SECTION;
EXEC TOOLS GET CONTEXT appl_1 INTO :pctx;
```

EXEC TOOLS SET CONTEXT Statement

An EXEC TOOLS SET CONTEXT statement saves context information from one foreign function for use in subsequent foreign function invocations. Use EXEC TOOLS SET CONTEXT instead of creating a global variable to hold information. EXEC TOOLS SET CONTEXT allows you to assign a meaningful text name to a pointer representing a location in memory. You can retrieve the pointer using EXEC TOOLS GET CONTEXT.

Syntax:

```
EXEC TOOLS SET CONTEXT host_name[, ...]
    BY context_name [, ...];
```

where:

host_name	Specifies the host language variable containing the information to be saved.
context_name	Specifies the name of the saved context.

Notes: The *context_name* can be a reference to one of the following items:

- a host language variable (prefixed with a colon)
- a constant

Example:

```
/*
** Example:  Save context information for later use
*/
EXEC SQL BEGIN DECLARE SECTION;
char my_context[20];
EXEC SQL END DECLARE SECTION;
strcpy(my_context, ”context_1”);
EXEC TOOLS SET CONTEXT :my_context BY appl_1;
```

EXEC ORACLE Statement

An EXEC ORACLE statement is a statement that is not standard SQL and is used to execute Oracle Precompiler options. For more information, refer to the *Programmer's Guide to the Oracle Precompilers*.

Creating a User Exit Interface to Foreign Functions

Creating a user exit interface to a foreign function involves the following:

- Creating an IAPXTB control structure that registers each user exit interface
- Integrating a user exit interface with Oracle Forms

Creating an IAPXTB Control Structure

The IAPXTB control structure is a data structure that contains information regarding all foreign functions that can be invoked from a user exit interface. The IAPXTB control structure designates the entry points necessary for linking your foreign functions to Oracle Forms. The following table describes each column in the IAPXTB control structure:

Column	*Content*
NAME	This column specifies a user exit name for a foreign function that can be invoked from a user exit interface. (This is not necessarily the name of the file that contains the foreign function or the name of the foreign function that is called.) Note that some host languages are case sensitive.
FUNCTION	This column specifies the name of the foreign function.
TYPE	This column specifies the language in which the foreign function is written. Valid values include: XITCC for C XITCOB for COBOL, XITFOR for FORTRAN XITPLI for PL/I XITPAS for PASCAL XITAda for Ada

You must enter one entry in the IAPXTB control structure for every foreign function that can be invoked from a user exit interface. This is true for all foreign functions that can be invoked from a user exit interface, whether a foreign function is in a file that is precompiled and compiled by itself, or precompiled and compiled with several other foreign functions. You should maintain all foreign functions that can be invoked from a user exit interface for a production system in one IAPXTB control structure. You should keep test versions of your foreign functions in a separate IAPXTB control structure.

To create the IAPXTB control structure:

1. Define the IAPXTB data structure.

   Make sure you define the data structure with three fields representing the name of the user exit interface, the name of the foreign function, and the language used to develop the foreign function.

2. Enter the data for each foreign function.
3. Compile the IAPXTB source file to generate an object code file. Retain the IAPXTB object code file for integrating the user exit interface with Oracle Forms.

Specific information about creating the IAPXTB control structure in Microsoft Windows is available in the section “A User Exit Interface to Foreign Functions on Microsoft Windows.” For information about creating the IAPXTB control structure in other environments, refer to the Oracle Forms documentation for your operating system.

Integrating a User Exit Interface with Oracle Forms

Integration of a user exit interface to Oracle Forms depends on the operating system on which you are working and the language in which you choose to write the foreign function.

On Microsoft Windows, you create a dynamic link library with foreign function object code files and IAPXTB control structure object code files. When a foreign function is invoked from a user exit interface, a dynamic link library loads into memory. For more information on integration of foreign functions in Microsoft Windows, refer to the section “A User Exit Interface to Foreign Functions on Microsoft Windows.”

For other environments, you must rebuild the Oracle Forms Runform executable by linking the object code files from the foreign function and IAPXTB control structure to Oracle Forms object code files. For more information on linking object code files and generating a Oracle Forms Runform executable, refer to the Oracle Forms documentation for your operating system.

To integrate a user exit interface with Oracle Forms:

1. Identify the foreign function object code file.
2. Identify the IAPXTB control structure object code file.
3. Link the foreign function object code file and the IAPXTB control structure object code file with either a dynamic link library or Oracle Forms Runform.

Invoking a Foreign Function from a User Exit Interface

After creating a user exit interface to a foreign function, you can invoke the foreign function using a user exit interface from Oracle Forms. To invoke a foreign function from a user exit interface, you call the USER_EXIT built–in subprogram from a trigger or from a user–named subprogram.

When you invoke a foreign function from a user exit interface, Oracle Forms temporarily passes processing control to the foreign function. When execution of the foreign function is complete, Oracle Forms regains processing control.

Syntax:

```
USER_EXIT(user_exit_string);
USER_EXIT(user_exit_string, error_string);
```

The USER_EXIT built–in calls the foreign function named in the *user_exit_string*.

Parameters:

user_exit_string	Specifies a user exit name for a foreign function that you want to call from a user exit interface, including any parameters. Maximum length of the user_exit_string is 255 characters.
error_string	Specifies a user–defined error message that Oracle Forms displays if the call to the foreign function fails. Maximum length of the error_string is 255 characters.

Restrictions: When you specify the user exit name that represents a foreign function (in the user_exit_string of the USER_EXIT built–in subprogram) that name must follow the rules of your operating system and host language. Be aware that these rules might include case sensitivity. You should also note that only one foreign function can be invoked per USER_EXIT built–in call.

Passing Parameter Values to a Foreign Function from Oracle Forms

You can pass parameters to a foreign function that is invoked from a user exit interface. You can pass parameter values by defining user_exit_string and error_string in the USER_EXIT built–in subprogram.

When you define user_exit_string and error_string in Oracle Forms, the foreign function recognizes the values as corresponding to a command line value and an error message value. For instance, Oracle Forms treats the value of user_exit_string as a string variable. The value of user_exit_string is the command line to a foreign function.

The following are foreign function parameters and their corresponding Oracle Forms definitions for each foreign function invoked from a user exit interface:

Foreign Function Parameter	*Oracle Forms Definition*
Command Line	user_exit_string.
Command Line Length	length (in characters) of user_exit_string.
Error Message	error_string.
Error Message Length	length (in characters) of error_string.
In–Query	a boolean value that reflects whether the foreign function was invoked from a user exit interface when the form was in Enter Query mode.

Although Oracle Forms automatically invokes the foreign function from the user_exit_string, Oracle Forms does not automatically parse the user_exit_string. It is the responsibility of the foreign function to parse the user_exit_string.

You can pass any number of parameters to a foreign function from the user_exit_string of the USER_EXIT built–in. Use this feature to pass information such as item names. For example, to pass two parameters, PARAM1 and PARAM2, to the CALCULATE_VALUES foreign function, you specify the following statement:

```
User_Exit('CALCULATE_VALUES PARAM1 PARAM2');
```

Returning a Value from a Foreign Function to Oracle Forms

When the execution of the Oracle Forms USER_EXIT built–in subprogram is complete, an integer value is returned to Oracle Forms. This integer value indicates whether the USER_EXIT built–in subprogram executed with success, failure, or a fatal error. (For example, if you are creating a foreign function in C, Oracle Forms provides the constants in a ".h" file.)

Error variables in Oracle Forms—FORM_FAILURE, FORM_FATAL, and FORM_SUCCESS—are set according to the value that is returned from the USER_EXIT built–in subprogram. You can query the error variables to determine the success or failure of the execution of the USER_EXIT built–in subprogram just as you would for any built–in subprogram. The trigger that calls the USER_EXIT built–in subprogram determines how to handle the return condition.

For example, you might want to check the value of FORM_SUCCESS after executing a foreign function from a user exit interface:

```
User_Exit('my_exit');
IF NOT Form_Success THEN
   handle the error
END IF;
```

A User Exit Interface to Foreign Functions on Microsoft Windows

This section describes aspects of Oracle Forms that are specific to its use in Microsoft Windows. For information about other environments, refer to the Oracle Forms documentation for your operating system.

In Microsoft Windows, a foreign function that can be invoked from a user exit interface is contained in a dynamic link library(DLL). A DLL is a library that loads into memory only when the contained code is invoked, and a DLL can be shared by multiple applications. Before proceeding, you should be familiar with the procedure for building DLLs, as described in your compiler manual.

Note: Some C runtime functions are not available in .DLL files. For more information, refer to your compiler documentation.

To create an Oracle Precompiler foreign function that can be invoked from a user exit interface on Microsoft Windows:

1. Write an Oracle Precompiler foreign function using the Oracle precompiler statements to embed SQL commands in your source code.
2. Precompile the foreign function source code with an Oracle precompiler.
3. Compile the output from the Oracle precompiler to generate a foreign function object code file. Be sure to specify the large memory model on your compiler.
4. Create an IAPXTB control structure and compile the source code to generate an IAPXTB object code file.
5. Build a DLL with the foreign function and IAPXTB control structure object code files.
6. Make sure you include the name of the DLL in the FORMS45_USEREXITS variable of the ORACLE.INI file, or rename the DLL to F45XTB.DLL. If you rename the DLL to F45XTB.DLL, replace the existing F45XTB.DLL in the \ORAWIN\BIN directory with the new F45XTB.DLL.
7. Invoke the foreign function from a user exit interface in Oracle Forms.

Microsoft Windows User Exit Interface Files

During the installation of Oracle Forms for Windows, a group of files related to the user exit interface (with the exception of F45XTB.DLL) are copied to the \ORAWIN\FORMS45\USEREXIT directory. You may not require all of the files contained in this directory. All foreign functions that can be invoked from a user exit interface from Oracle Forms for Windows must be contained in a DLL.

F45XTB.DLL	is the default file containing foreign functions that can be invoked from a user exit interface. This file is a DLL that ships with Oracle Forms, and does not initially contain user–defined foreign functions. This file is placed in the \ORAWIN\BIN directory during installation. When you create new foreign functions, replace the existing F45XTB.DLL file with a new F45XTB.DLL.

To assist you in creating the IAPXTB control structure, Oracle Forms provides you with two IAPXTB control structure source files, UE_XTB.C and UE_XTBN.C. Each file serves as a template for creating an IAPXTB control structure. Modify a IAPXTB control structure source file to include the foreign functions you define. Include the appropriate file in your project. You only need one of the two source files to create the IAPXTB control structure.

UE_XTB.C	is a file that contains an example of an entry for the IAPXTB control structure. Modify this file and add your foreign function entries. This is the file that is in the UE_SAMP.MAK project file.
UE_XTBN.C	is a file that contains an *empty* IAPXTB control structure. Add your foreign function entries to create an IAPXTB control structure. This is the file that is in the UE_XTBN.MAK project file.

The following files are project files that contain all the required files to create a DLL containing foreign functions that you can invoke from a user exit interface in Oracle Forms.

UE_SAMP.MAK	is a project file that includes the IAPXTB control structure from the UE_XTB.C file. Building this project generates UE_SAMP.DLL. You can rename the DLL from UE_SAMP.DLL to F45XTB.DLL and replace the existing F45XTB.DLL in the \ORAWIN\BIN directory, or you can add UE_SAMP.DLL to the list of DLLs defined by the FORMS45_USEREXITS parameter in the ORACLE.INI file.
UE_XTBN.MAK	is a project file that includes the IAPXTB control structure from the UE_XTBN.C file. Building this project generates UE_XTBN.DLL. This is the project file that is used to generate the initial F45XTB.DLL that resides in the \ORAWIN\BIN directory. You can rename the DLL from UE_XTBN.DLL to F45XTB.DLL and replace the existing F45XTB.DLL in the \ORAWIN\BIN directory, or you can add UE_XTBN.DLL to the list of DLLs defined by the FORMS45_USEREXITS parameter in the ORACLE.INI file.

In addition to your foreign function object code files and an IAPXTB control structure object code file, you need the following files in your project file to generate a user exit interface DLL (These files are included in UE_SAMP.MAK and UE_XTBN.MAK):

F45XTB.DEF	contains definitions you need to build your own DLL.
OSSWEP.OBJ	is the Dynamic Link Library Windows Entry Point .OBJ file you need to build your own DLL. (You may replace OSSWEP.OBJ with an .OBJ file of your own.)
UEZ.OBJ	is an .OBJ file that you link to your own .OBJ files.

Compiling Microsoft Windows Foreign Functions

When compiling your foreign functions, be sure to specify the large memory model. Refer to your Microsoft Windows compiler documentation for additional information, such as restrictions on building a Microsoft Windows DLL, and the use of #define statements. (Your foreign function code may need to include the UE.H file to access typedefs and #define statements.)

Creating the IAPXTB Control Structure for Microsoft Windows

You can create the IAPXTB control structure by using the UE_XTB.C or the UE_XTBN.C file as a template. Alternatively, you can create your own IAPXTB control structure in a self–defined file.

Note: In Microsoft Windows, an IAPXTB control structure is required for building each DLL that contains foreign functions that can be invoked from a user exit interface. You should only include entries in the IAPXTB control structure for corresponding foreign functions that are contained in a DLL.

The following is an excerpt from a slightly modified UE_XTB.C file:

```
extern exitr iapxtb[] = { /* Holds exit routine pointers */
    “UE_Name”, UE_Funct, XITCC,
    “USEREXECSQL”, uxsql, XITCC,
    (char *)0, 0, 0    /* zero entry marks the end */
} /* end iapxtb */
```

The file includes the user exit name UE_Name. In this example, the following line was added to the original UE_XTB.C file:

```
    "UE_Name", UE_Funct, XITCC,
```

UE_Name	is the user exit name for use by the USER_EXIT built–in subprogram to invoke the foreign function from Oracle Forms.
UE_Funct	is the name of the foreign function that temporarily takes over processing control from Oracle Forms.
XITCC	specifies the C programming language that is used to develop the foreign function.

Building a Microsoft Windows Dynamic Link Library

Two project files, UE_SAMP.MAK and UE_XTBN.MAK, are guides to help you create a DLL containing foreign functions that can be invoked from a user exit interface.

You also have the option of defining your own project file. When creating your own project file, remember to include a IAPXTB control structure object code file, the foreign function object code files, and the required files for integrating foreign functions that can be invoked from a user exit interface in Oracle Forms.

One of the required files for integrating foreign functions with Oracle Forms is the F45XTB.DEF file. Use the F45XTB.DEF file to export foreign functions. Some export statements for Oracle Forms already exist. Do not modify the existing export statements, because the functions are used by Oracle Forms to access user exit interfaces.

The UE_SAMP.MAK project file is used here as an example. In addition to the object code files containing your foreign functions, UE_SAMP.MAK includes the following files:

\orawin\forms45\userexit\uez.obj
\orawin\forms45\userexit\osswep.obj
\orawin\forms45\userexit\f45xtb.def
\orawin\forms45\userexit\ue_xtb.c

You must also link the following files:

LDLLCEW.LIB
LIBW.LIB
OLDNAMES
\orawin\forms45\userexit\F45R.LIB

If you are creating an Oracle Precompiler foreign function, you must link the following libraries:

\orawin\pro20\userexit\sql16win.lib
\orawin\pro20\userexit\sqx16win.lib

You create UE_SAMP.DLL after building the UE_SAMP.MAK project file. You can rename UE_SAMP.DLL to F45XTB.DLL, make a backup copy of F45XTB.DLL located in \ORAWIN\BIN, and replace the existing F45XTB.DLL with the new F45XTB.DLL. Alternatively, you can add UE_SAMP.DLL to the list of DLLs defined by the FORMS45_USEREXITS parameter in the ORACLE.INI file.

Defining Foreign Functions in Multiple Dynamic Link Libraries

Foreign functions developed for use in Oracle Forms for Windows are contained in DLLs. Oracle Forms establishes a single DLL, F45XTB.DLL, for containing foreign functions. F45XTB.DLL is the default DLL for containing foreign functions. Multiple foreign functions can be contained in a single DLL.

Using a single DLL that contains all foreign functions can cause conflicts, especially when two programs try to access the same DLL. To alleviate dynamic library conflicts, Oracle Forms supports multiple user exit interface DLLs; foreign functions can be contained in multiple DLLs without restrictions on the name of the DLL.

FORMS45_USEREXITS parameter The FORMS45_USEREXITS parameter in the ORACLE.INI file allows you to define multiple DLLs to contain foreign functions that can be invoked from a user exit interface. The FORMS45_USEREXITS parameter includes a semicolon delimited list of user exit interface DLLs.

This is an example of defining multiple user exit interface DLLs

```
FORMS45_USEREXITS = C:\mathlib\add.dll;C:\mathlib\mult.dll;
```

A DLL loads into memory when any one of the foreign functions it contains is invoked from the user exit interface. Although the FORMS45_USEREXITS parameter can list many DLLs that each contain multiple user exit interface foreign functions, only one such DLL is in memory at any time.

Oracle Forms determines the foreign function to invoke from a user exit interface using the following criteria:

- If the FORMS45_USEREXITS parameter does not exist in the ORACLE.INI file, foreign functions must be contained in a single user exit DLL named F45XTB.DLL and located in the \ORAWIN\BIN directory.

- If the FORMS45_USEREXITS parameter exists in the ORACLE.INI file and there are multiple user exit interface DLLs define, the first occurrence of the foreign function in the list of DLLs is invoked.
- If there are multiple user exit interface DLLs that contain non–unique foreign function names, the non–unique function name that is invoked is the first occurrence of function that follows the content in the user exit interface cache memory.

NOTE: To avoid calling an unexpected foreign function, you should not use the same foreign function name more than once in any of your user exit interface DLLs. An example of a non–unique function name is when two different functions have the same name, but are contained in different user exit interface DLLs.

User Exit Interface Cache Memory Oracle Forms deals with non–unique function names by maintaining a user exit interface cache memory. Exiting a form clears the user exit interface cache memory, otherwise the user exit interface cache memory retains the last called foreign function in memory until you call another foreign function.

Invoking a foreign function that has a non–unique function name depends on what is in the user exit interface cache memory. The first occurrence of a foreign function with a non–unique function name that follows the foreign function in the user exit interface cache memory is invoked.

If the user exit interface cache memory is empty, the first occurrence of a function with the non–unique function name in the list of DLLs is invoked. Because the cache memory is not cleared until a you exit a form, subsequent calls to a non–unique function name may result in an unexpected foreign function call. For instance, you may accidentally call a function that follows a function with a unique name as opposed to calling a function that appears earlier in the sequence of user exit interface DLLs. Although in many cases, the user exit interface cache memory correctly identifies foreign functions with non–unique names, you should use unique foreign function names in the list of DLLs that are members of the FORMS45_USEREXITS variable whenever possible.

An Example of a User Exit Interface in Microsoft Windows

The following is an example of creating and invoking a foreign function from a user exit interface. This example uses the UE_SAMP.MAK project file.

1. Write a foreign function using Pro*C and Oracle precompiler statements to access the database.

   This is a precompiler foreign function in a file named UEXIT.PC. The foreign function adds an ID column to the EMP table.

   ```
   /* UEXIT.PC file */
   #ifndef UE
   #include “ue.h”
   #endif

   #ifndef _WINDLL
   #define SQLCA_STORAGE_CLASS extern
   #endif

   EXEC SQL INCLUDE sqlca.h
   void AddColumn() {
   EXEC SQL alter table EMP add ID varchar(9);
   }
   ```

2. Precompile the foreign function with the Pro*C precompiler.

   The input to the Pro*C precompiler is the file UEXIT.PC. The output from the Pro*C precompiler is the file UEXIT.C. You should also create a header file to prototype your foreign functions. In this example, a UEXIT.H file is created to declare the AddColumn function.

3. Create the IAPXTB control structure.

   Modify the file UE_XTB.C file by including the UEXIT.H file and adding the user exit name, foreign function name, and language type. Follow the example in the UE_XTB.C file. In this case, the following entry is added to the file:

   ```
   Add_ID_Column, AddColumn, XITCC
   ```

4. Modify any required foreign function integration files.

   Modify the F45XTB.DEF file by adding export statements to include the AddColumn foreign function. Follow the examples in the F45XTB.DEF file.

5. Build a DLL for use with Oracle Forms Runform.

   With the exception of UEXIT.C, the following files should already be included in the UE_SAMP.MAK project file:

   ```
   c:\orawin\forms\userexit\uez.obj
   c:\orawin\forms\userexit\osswep.obj
   c:\orawin\forms\userexit\f45xtb.def
   c:\orawin\forms\userexit\ue_xtb.c
   c:\orawin\forms\userexit\uexit.c
   ```

   The UE_SAMP.MAK project is set up to link the following files:

   ```
   LDLLCEW.LIB
   LIBW.LIB
   OLDNAMES
   c:\orawin\forms45\userexit\f45r.LIB
   c:\orawin\pro20\userexit\sql16win.lib
   c:\orawin\pro20\userexit\sqx16win.lib
   ```

   After building the UE_SAMP.MAK project, the result is a DLL named UE_SAMP.DLL. Add the UE_SAMP.DLL entry to the list of DLLs defined by the FORMS45_USEREXITS parameter in the ORACLE.INI file.

   Alternatively, you can rename UE_SAMP.DLL to F45XTB.DLL, backup the F45XTB.DLL in the c:\orawin\bin directory, and copy the new F45XTB.DLL to the c:\orawin\bin directory.

6. Invoke the foreign function from a user exit interface in Oracle Forms.

   In this case, a When–Button–Pressed Trigger calls the foreign function from a user exit interface. The following statement demonstrates how to invoke the AddColumn foreign function by specifying the user exit name Add_ID_Column in the USER_EXIT built–in subprogram:

   ```
   /* Trigger:  When–Button–Pressed */
   USER_EXIT('Add_ID_Column');
   ```

Accessing the Microsoft Windows SDK From a User Exit Interface

You can invoke Microsoft Windows SDK functions from a user exit interface. Invoking Microsoft Windows SDK functions is similar to invoking user–defined foreign functions from a user exit interface on Microsoft Windows. Instead of including the object code files of your user–defined foreign function when building a dynamic link library, you must include the source files of the Microsoft Windows SDK function.

Parameter values for Microsoft Windows SDK functions can be passed to or received from Oracle Forms like other foreign functions. For Microsoft Windows SDK functions that require a window handle parameter, you can obtain the window handle from Oracle Forms using the GET_ITEM_PROPERTY function to examine the Window_Handle property. A window handle is a unique internal character constant that is used to refer to objects. For information on the Window_Handle property, refer to the *Oracle Forms Reference Manual, Vol. 2.*

There are many reasons for accessing the Microsoft Windows SDK. For example, by obtaining a window handle from Oracle Forms, you can invoke Microsoft Windows SDK functions to externally modify the properties of objects in your Oracle Forms applications. The following is an example of calling the Microsoft Windows SDK function GetWindowRect from an Oracle Forms trigger or user–defined subprogram:

```
:block1.item_handle := get_item_property('block1.item1',
                                          Window_Handle);

USER_EXIT(GetWinRec || :block1.item_handle,'error_message');
```

CHAPTER

4

Connecting to Non–ORACLE Data Sources

Oracle Forms applications can run against non–ORACLE data sources. This chapter describes your options for developing such applications, and includes the following topics:

About Connecting to Non–ORACLE Data Sources

There are three ways to create applications that run against non–ORACLE data sources. The first is to use the Open Gateway products available from Oracle to make the connection. There are Open Gateway products available for many third–party databases. Open Gateway automatically manages the interaction between Oracle Forms and your non–ORACLE data source. For more information on Open Gateway products, contact Oracle Corporation.

Another way to connect to non–ORACLE data sources is through ODBC, using Oracle's Open Client Adapter driver. For more information, refer to Chapter 14, "Oracle Open Client Adapter for ODBC."

If no Open Gateway or Open Client Adapter drivers exist for your data source, or if your application has special requirements, you can still connect to virtually any data source by writing the appropriate set of transactional triggers in your form.

The set of transactional triggers available in Oracle Forms, together with user exits or PL/SQL calls to foreign functions, allows you to replace Oracle Forms default transaction processing with functionality appropriate to your data source. Using transactional triggers, you can develop forms that

- run against a non–ORACLE data source
- run against ORACLE *or* a non–ORACLE data source
- include some blocks that are based on ORACLE tables and other blocks that access a non–ORACLE data source
- use Open Gateway to connect to one non–ORACLE data source and use transactional triggers to connect to another

Connecting with Open Gateway

When you connect to a non–ORACLE data source with an Open Gateway product, there are four transaction processing options in Oracle Forms that you should be aware of. These options include two block properties and two form module properties:

- Key Mode block property
- Locking Mode block property
- Cursor Mode form module property
- Savepoint Mode form module property

You can set these properties to specify how Oracle Forms should interact with your non–ORACLE data source. The specific settings you will use for these properties will depend on the capabilities and limitations of the data source to which you are connecting.

Key Mode Block Property

The Key Mode block property determines how Oracle Forms uniquely identifies rows in the database. ORACLE uses unique ROWID values to identify each row. Non–ORACLE databases do not include the ROWID construct, but instead rely solely on unique primary key values to identify unique rows. If you are creating a form to run against a non–ORACLE data source, you must define primary keys, and set the Key Mode block property accordingly.

Value	*Description*
Unique (the default)	The default setting. Instructs Oracle Forms to use ROWID constructs to identify unique rows in an ORACLE database.
Updateable	Specifies that Oracle Forms should issue UPDATE statements that include primary key values. Use this setting if your database allows primary key columns to be updated and you intend for the application to update primary key values.
Non–Updateable	Specifies that Oracle Forms should not include primary key columns in any UPDATE statements. Use this setting if your database does not allow primary key values to be updated.

Note: Some Open Gateway products use pseudo ROWIDs that allow you to run your form in the default Unique_Key mode against that particular data source.

Note: When the Key Mode property is set to one of the primary key modes, you must identify the primary key items in your form by setting the Primary Key item property True for at least one item in the block.

Examples: The following examples illustrate how the Key Mode property affects transaction processing.

Consider a DEPT table with columns named DEPTNO, DNAME, and LOC. The DEPTNO column is the primary key.

The following figure shows how a row in the DEPT table would be stored in an ORACLE database. Note the ROWID pseudo–column that ORACLE uses to identify unique rows.

ROWID	*DEPTNO*	*DNAME*	*LOC*
000012C1.0001.0001	30	SALES	CHICAGO

The statement issued for an unqualified query on the preceding table from within Oracle Forms appears as follows:

```
SELECT rowid, deptno, dname, loc
FROM dept;
```

Now assume that the operator fetches the example row into the DEPT block in the form, changes the DNAME value from 'SALES' to 'CONSULTING,' and then commits the transaction. The following sections show how Oracle Forms manages the update differently, depending on the setting of the Key Mode property.

Example 1: Unique Key

When the Key Mode property is set to Unique (the default setting) Oracle Forms issues the following SQL statements to reserve the record for update, and then update the record:

```
SELECT rowid, deptno, dname, loc
FROM dept
WHERE rowid = '000012C1.0001.0001'
AND deptno = 30 AND dname = 'Sales' AND loc = 'Chicago'
FOR UPDATE OF deptno, dname, loc
NOWAIT;

UPDATE dept
SET deptno = 30, dname = 'Consulting', loc = 'Chicago'
WHERE rowid = '000012C1.0001.0001';
```

Example 2: Updateable Key

When the Key Mode property is set to Updateable, Oracle Forms issues the following statements to reserve the record for update, and then update the record:

```
SELECT deptno, dname, loc
FROM dept;

SELECT deptno, dname, loc
FROM dept
WHERE deptno = 30
FOR UPDATE OF deptno, dname, loc;
UPDATE dept
SET deptno = 30, dname = 'Consulting', loc = 'Chicago'
WHERE deptno = 30;
```

Notice that the ROWID construct is not included in these statements.

Example 3: Non–Updateable Key

When the Key Mode property is set to Non–Updateable, Oracle Forms issues the following statements to reserve the record for update, and then update the record:

```
SELECT deptno, dname, loc
FROM dept;

SELECT deptno, dname, loc
FROM dept
WHERE deptno = 30
FOR UPDATE OF dname, loc;

UPDATE dept
SET dname = ’Consulting’, loc = ’Chicago’
WHERE deptno = 30;
```

In this example, the primary key DEPTNO column is not included in the UPDATE statement issued by Oracle Forms. When you use Non–Updateable Key mode, it is usually best to set the Update Allowed item property to False for non–updateable primary key items. This setting gives immediate feedback to operators that the primary key value cannot be edited in a queried record.

Locking Mode Block Property

Specifies when Oracle Forms should attempt to obtain database locks on rows that correspond to queried records in the form.

The following table describes the allowable settings for the Locking Mode property.

Value	*Description*
IMMEDIATE (the default)	Specifies that Oracle Forms should attempt to lock the corresponding row immediately after an operator or the application modifies an item value in a queried record. With this setting, Oracle Forms locks the record as soon as the operator presses a key to enter or edit the value in a text item.
DELAYED	Specifies that Oracle Forms should wait to lock the corresponding row in the database until the transaction is about to be committed. With this setting, the record is locked only while the transaction is being posted to the database, not while the operator is editing the record.

Note: It is possible to implement an optimistic locking scheme by using the On–Lock trigger to suppress locking as necessary.

ORACLE Version 6 and the Oracle7 Server support row–level locking to maximize concurrency. Non–ORACLE databases do not always support row–level locking, but rather, support page or table–level locking. Instead of locking only the row that has been modified for update, these databases lock the entire page or table.

When a form is running against a database that supports page or table–level locking, the effect of locking on resource contention is potentially more severe because more records are affected.

Between immediate locking and delayed locking there is a potential trade–off between low concurrency and lost updates. Using immediate locking can result in low concurrency, while delaying locking can result in *lost updates*. An update can be lost when another user locks and updates the same row before the first operator finishes updating and committing a record. When this happens, Oracle Forms issues a message that the record has been modified by another user. The operator must then re–query the updated row and make the desired changes again.

For more details, refer to the description of the Locking Mode property in online Help, or in *Oracle Forms Reference Manual, Vol. 2.*

Cursor Mode Form Property

The Cursor Mode form property defines the cursor state across transactions. The cursor refers to the memory work area in which SQL statements are executed. For more information on cursors, refer to the *ORACLE RDBMS Database Administrator's Guide*.

The following table describes the values that are valid for the Cursor_Mode option:

Value	*Description*
OPEN (the default)	Specifies that cursors remain open across transactions within the data source.
CLOSE	Specifies that cursors are closed at commit time by the datasource.

Because ORACLE allows the database state to be maintained across transactions, Oracle Forms allows cursors to remain open across COMMIT operations. This reduces overhead for subsequent execution of the same SQL statement because the cursor does not need to be re–opened and the SQL statement does not always need to be re–parsed.

Some non–ORACLE databases do not allow database cursor state to be maintained across transactions. Therefore, you can set the Cursor Mode property to Close to satisfy those requirements. However, keep in mind that closing cursors at commit time and re–opening them at execute time can degrade performance in three areas:

- during the COMMIT operation
- during future execution of other SQL statements against the same records
- during execution of queries

For more details, refer to the description of the Cursor Mode property in the online Help, or in *Oracle Forms Reference Manual, Vol. 2.*

Savepoint Mode Form Property

The Savepoint Mode form property specifies whether Oracle Forms should issue savepoints during a session.

The following table describes the valid settings for the Savepoint Mode property.

Value	*Description*
True (the default)	Specifies that Oracle Forms should issue a savepoint at form startup and at the start of each Post and Commit process.
False	Specifies that Oracle Forms is to issue no savepoints, and that no rollbacks to savepoints are to be performed.

When Savepoint Mode is set to False, Oracle Forms does not allow a form that has uncommitted changes to invoke another form with the CALL_FORM procedure.

For more details, refer to the description of the Savepoint Mode property in online Help, or in *Oracle Forms Reference Manual, Vol. 2.*

Using Transactional Triggers

Included in Oracle Forms is a set of *transactional triggers* that fire in response to transaction processing events. These events represent points during application processing at which Oracle Forms needs to interact with the data source. Examples of such events include updating records, rolling back to savepoints, and committing transactions.

By default, Oracle Forms assumes that the data source is an ORACLE database, and issues the appropriate SQL statements to optimize transaction processing accordingly. However, by defining transactional triggers and user exits (3GL programs you write yourself and then link into a form at generate time), you can build a form to interact with virtually any data source, including even non–relational databases and flat files.

These next sections explain how you can use transactional triggers to create applications that run against non–ORACLE data sources. The specifics of your implementation will, of course, depend on the data source to which you are connecting.

The information in this chapter provides an overview of what is required, and points you to other sources of information in the Oracle Forms documentation set.

Transactional Trigger Set

The set of transactional triggers available for implementing non–ORACLE data source support includes the following On–event triggers:

- On–Check–Unique
- On–Close
- On–Column–Security
- On–Commit
- On–Count
- On–Delete
- On–Fetch
- On–Insert
- On–Lock
- On–Logon

- On–Logout
- On–Rollback
- On–Savepoint
- On–Select
- On–Sequence–Number
- On–Update

In addition, there are ”pre–event” triggers and ”post–event” triggers you can use to trap events that occur just prior to and immediately after the corresponding On–event:

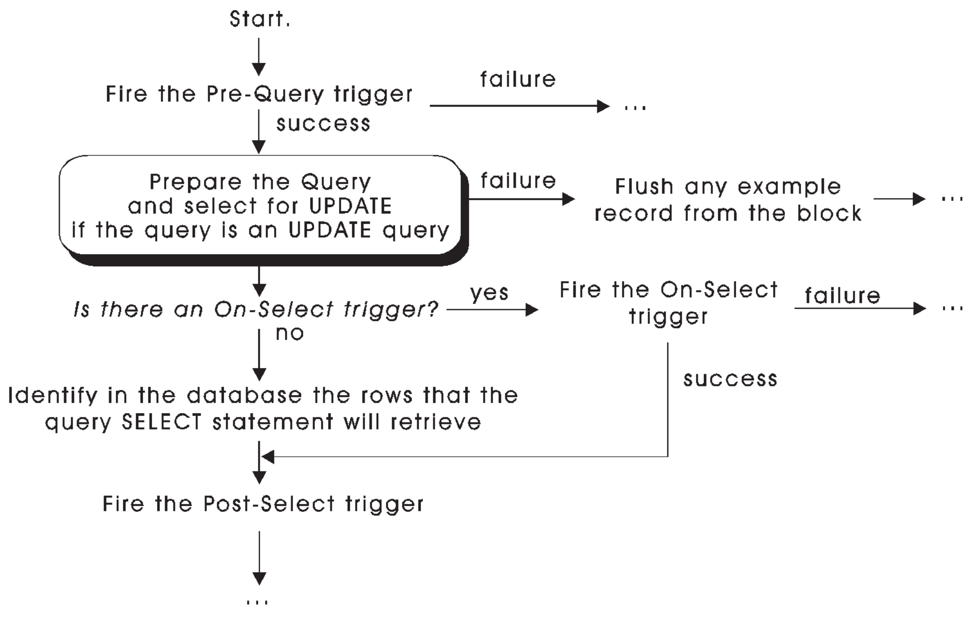
On–Event Trigger	*Corresponding Pre/Post Triggers*
On–Commit	Pre–Commit / Post–Forms–Commit / Post–Database–Commit
On–Delete	Pre–Delete / Post–Delete
On–Insert	Pre–Insert / Post–Insert
On–Logon	Pre–Logon / Pre–Logon
On–Logout	Pre–Logout / Post–Logout
On–Select	Pre–Select / Post–Select
On–Update	Pre–Update / Post–Update

The transactional triggers that apply to transaction processing in a block (On–Select, On–Fetch, etc.) can be defined at either the block level or the form level. Transactional triggers that apply to the Runform session (On–Logon, On–Savepoint, On–Rollback, etc.) can be defined at the form level only.

Replacing Default Processing

The transactional triggers whose names begin with ”On–” (On–Select, On–Update, etc.) replace the default processing that Oracle Forms would normally perform at that point in the application. Put another way, an On–event trigger bypasses the function that Oracle Forms would execute if there were no On–event trigger at that point. The following figure shows the first part of the ”Open the Query” flow chart, taken from Chapter 8 of the *Oracle Forms Reference Manual*.

Open the Query


Start.
Fire the Pre-Query trigger
failure
...
success
Prepare the Query and select for UPDATE if the query is an UPDATE query
failure
Flush any example record from the block
...
Is there an On-Select trigger?
yes
Fire the On-Select trigger
failure
...
no
success
Identify in the database the rows that the query SELECT statement will retrieve
Fire the Post-Select trigger
...

The flow chart illustrates how a query is processed differently depending on whether an On–Select trigger is defined in the form. When no On–Select trigger is present, Oracle Forms performs the default processing required to issue the SELECT statement that identifies the rows in the database that meet the current query criteria (as specified by the example record).

If, however, an On–Select trigger has been defined at the appropriate definition level, Oracle Forms bypasses the default processing for selecting records, and instead executes the code in the On–Select trigger.

In this case, the On–Select trigger is being executed *in place of* the default Oracle Forms processing.

When you define an On–event trigger in a form, you are essentially telling Oracle Forms *not* to do what it would normally have done at that point of transaction processing, because you are going to manage that function yourself by writing appropriate code in the On–trigger. That is, you are *replacing* the default functionality with the functionality you specify in the On–event trigger.

Calling User Exits

When you define transactional triggers to interact with a non–ORACLE data source, you will usually include a call to a user exit in the appropriate triggers. User exits for non–ORACLE data source support are usually written in one of the third–generation programming languages for which there is an Oracle Precompiler, including C, Ada, COBOL, FORTRAN, Pascal, and PL/I. User exits created with an Oracle Precompiler have access to form variables and item values. You call a user exit from a trigger with the USER_EXIT built–in procedure.

The code in your user exit interacts with the non–ORACLE data source. Once the user exit has performed the appropriate function (as indicated by the trigger from which it was called), it returns control to Oracle Forms for subsequent processing. For example, a user exit called from an On–Fetch trigger might be responsible for retrieving the appropriate number of records from the non–ORACLE data source. Once the records are retrieved, Oracle Forms takes over the display and management of those records in the form interface, just as it would if the records had been fetched from an ORACLE database.

Note: You can also call external functions directly from PL/SQL, without having to write user exits. For information, refer to *Oracle Forms Advanced Techniques,* Chapter 13, "PL/SQL Interface to Foreign Functions."

Augmenting and Suppressing Default Processing

When you develop applications to run against a non–ORACLE data source, you will often use On–event triggers to completely replace the default Oracle Forms processing with your own, data source–specific functionality.

Occasionally, however, you may simply want to augment the default functionality that Oracle Forms normally performs at some transaction event–point. Or you might want to suppress the default processing entirely, without doing anything in its place. This section explains how On–triggers, together with appropriate built–in subprograms, can be used to augment or suppress default transaction processing.

Built–in Subprograms for On–Event Triggers For most of the transactional On–event triggers, there is a corresponding built–in subprogram.

On–Event Trigger	*Corresponding Built–in*
On–Check–Unique	CHECK_RECORD_UNIQUENESS
On–Close	none
On–Column–Security	ENFORCE_COLUMN_SECURITY
On–Commit	COMMIT_FORM
On–Count	COUNT_QUERY
On–Delete	DELETE_RECORD
On–Fetch	FETCH_RECORDS
On–Insert	INSERT_RECORD
On–Lock	LOCK_RECORD
On–Logon	LOGON
On–Logout	LOGOUT
On–Rollback	ISSUE_ROLLBACK
On–Savepoint	ISSUE_SAVEPOINT
On–Select	SELECT_RECORDS
On–Sequence–Number	GENERATE_SEQUENCE_NUMBER
On–Update	UPDATE_RECORD

When you call one of these built–in subprograms from its corresponding transactional trigger, Oracle Forms performs the default processing that it would have done normally at that point in the transaction.

For example, if you call the INSERT_RECORD procedure from an On–Insert trigger, Oracle Forms performs the default processing for inserting a record in the database during a commit operation.

When these built–ins are issued from within their corresponding transactional triggers, they are known as *do–the–right–thing* built–ins. That is, they do what Oracle Forms would normally have done at that point if no trigger had been present. Thus, an On–Insert trigger that calls the INSERT_RECORD procedure is functionally equivalent to not having an On–Insert trigger at all. Such a trigger is explicitly telling Oracle Forms to do what it would have done by default anyway.

Augmenting Default Processing You can call do–the–right–thing built–ins from transactional triggers when you want to *augment* default processing. That is, when you want Oracle Forms to do what it would normally do, and also do something else. For example, you might create a form with three blocks, two of which are based on tables in an ORACLE database and one of which is based on data in a non–ORACLE database. At startup, the form needs to log on to ORACLE, and also to establish that the non–ORACLE data source is available. Your On–Logon trigger might look like this:

On–Logon Trigger:

```
/* Do the default logon to ORACLE */
Logon;

/* Now initialize the non–ORACLE data source */
User_Exit('other_db');
```

If you had failed to include the LOGON built–in in this trigger, Oracle Forms would never have performed its default logon processing.

A related use for do–the–right–thing built–ins is to create applications that can run against *both* an ORACLE database and a non–ORACLE data source. By including a conditional statement in the appropriate transactional triggers, you can build a form that performs default transaction processing when running against ORACLE, or executes the appropriate user exits to run against a non–ORACLE data source. The decision can be based on the current value of a form parameter or global variable that gets set at form startup.

On–Insert Trigger:

```
IF :PARAMETER.which_db = 'oracle' THEN
   Insert_Record;
ELSE
   User_Exit('do_insert');
END IF;
```

If you use any of the do–the–right–thing built–ins in your transactional triggers, be sure to check for their success or failure immediately afterward by using one of these techniques:

- Call Check_Package_Failure (which should automatically have been created in any form that contains a relation).

or

- Check the FORM_SUCCESS (BOOLEAN) function and raise the Form_Trigger_Failure exception if the do–the–right–thing built–in fails.

The following example trigger illustrates this technique:

On–Delete Trigger:

```
/*
**  Checking result of do–the–right–thing built–in
*/
IF (some–condition) THEN
   User_Exit('DELREC');
ELSE
   Delete_Record;
   Check_Package_Failure;
END IF;
```

Since a failure does not halt the currently–executing trigger, if you do not check for failure and raise the exception, the trigger will continue to execute and exit successfully. Unless Form_Trigger_Failure is raised appropriately, Runform will assume that your On–Delete trigger accomplished its goal.

Using the NULL Statement Another use for the On–event triggers is to suppress default processing completely, without replacing or augmenting it with alternative code. You can suppress default transaction processing by including the NULL statement in the appropriate On–event trigger.

An example of functionality you might want to suppress is default locking. By default, Oracle Forms takes advantage of the record locking capabilities of ORACLE by attempting to obtain locks as operators query and modify records. The On–Lock trigger fires whenever Oracle Forms requests a lock. If, however, your non–ORACLE data source does not support locking, or you want to implement a fully–optimistic locking scheme, you must suppress default locking to avoid errors.

For example, to suppress locking you could define the following On–Lock trigger:

On–Lock Trigger

```
NULL;
```

This trigger suppresses default processing by ”replacing” it with NULL, that is, by doing nothing.

Transactional Triggers Block Property

There is a Transactional Triggers block property that you can set to identify blocks in your form as non–database blocks that Oracle Forms should manage as transactional blocks.

Recall that a base table block is one that is based on a database table or view. The Base Table block property identifies the table on which such a block is based. By default, Oracle Forms automatically supports transaction processing from a base table block; that is, operators can execute queries in the block, and inserts and updates are automatically processed by the next commit.

When you create a non–ORACLE data source application, you are essentially simulating the functionality of a base table block by creating a *transactional control block*. Such a block is a control block because its base table is not specified at design time (the Base Table block property is NULL), but it is transactional because there are transactional triggers present that cause it to function as if it were a base table block.

In a non–ORACLE data source application, you will often need to define both transactional control blocks and standard control blocks. For example, you might define a standard control block to display totals and summary information calculated from queried records in a transactional block.

When you build such an application, you need a way to specify which control blocks are transactional, and which are merely standard control blocks. You can do so by setting the Transactional Triggers block property to True for any control block that you want Oracle Forms to treat as a transactional block.

Note: When you create a block to run against either ORACLE or a non–ORACLE data source (with an IF statement in each transactional trigger), you must set the Base Table property as you would normally. In this case, you do not set the Transactional Triggers property to True.

Which Transactional Triggers are Required

One of the first steps when building a non–ORACLE data source application is to decide which transactional triggers you will need to define. In most non–ORACLE data source applications, all of the transactional triggers are present, but only some of them actually call user exits to interact with the non–ORACLE data source. The remainder are used simply to suppress default processing, and so contain only NULL statements.

It is important to remember that Oracle Forms will attempt to perform default processing for any transactional event for which you do not define a corresponding On–event trigger.

Deciding which triggers should call user exits to interact with the non–ORACLE data source and which will be used only to suppress default processing requires an understanding of how Oracle Forms interacts with the data source during transaction processing. The next sections explain these processes in greater detail.

About Transaction Processing

This section provides information about specific areas of transaction processing that you need to be aware of when you define transactional triggers for non–ORACLE data source support. It covers the following topics:

- Logon and Logout Processing
- Query and Fetch Processing
- Count Query Processing
- Commit Processing
- Savepoint and Rollback Processing
- Check Column Security Processing
- Generate Sequence Number Processing
- Lock Record Processing

These sections frequently refer to specific triggers, built–in subprograms, object properties, and processing flow charts that are described elsewhere in the Oracle Forms documentation set:

- For descriptions of specific transactional triggers, refer to the online Help or Chapter 2, “Triggers,” in the *Oracle Forms Reference Manual, Vol. 1.*
- For descriptions of specific do–the–right–thing built–ins, refer to the online Help or Chapter 3, “Built–in Subprograms,” in the *Oracle Forms Reference Manual, Vol. 1.*
- For information on object properties, refer to the online Help or Chapter 5, ”Properties,” in the *Oracle Forms Reference Manual, Vol. 2.* The following properties are especially important for non–ORACLE data source support:
  - Column Security
  - Database_Value
  - Datasource
  - Key Mode
  - Locking Mode
  - Primary Key
  - Query_Hits
  - Query_Options
  - Records_to_Fetch
  - Savepoint Mode
  - Savepoint_Name
  - Transactional Triggers
  - Update_Permission
- For information on user exits, refer to Chapter 3, User Exit Interface to Foreign Functions,” in *Oracle Forms Advanced Techniques.*
- For descriptions of runtime processes, refer to the following flow charts in Chapter 8, “Processing Flowcharts,” in the *Oracle Forms Reference Manual, Vol. 2.*
  - Check Record Uniqueness
  - Close the Query
  - Count_Query
  - Execute_Query
  - Fetch Records

– Generate Sequence Number
– Lock_Record
– Lock the Row
– Logon
– Logout
– Open the Query
– Post and Commit Transactions
– Prepare the Query
– Savepoint

Logon and Logout Processing

By default, Oracle Forms attempts to log on at form startup, either to ORACLE or to an Open Gateway connection. There can be only one such default connection per Runform session. It is possible, however, to programmatically log out of one connection and log on to another during a session. It is also possible to create a form that has one or more base table blocks running against a default connection and one or more transactional control blocks running against a non–ORACLE data source.

The following transactional triggers are available for replacing and augmenting the default logon and logout processing:

Logon Triggers:

- Pre–Logon
- On–Logon
- Post–Logon

Logout Triggers:

- Pre–Logout
- On–Logout
- Post–Logout

In a non–ORACLE data source application, the On–Logon and On–Logout triggers are usually required. These triggers fire when Oracle Forms normally interacts with ORACLE, and so must either suppress or replace the default interaction. In fact, if you were to build a form that required no data source at all, such as a demo or tutorial application, you would still need to include an On–Logon trigger to suppress the default logon attempt.

Using the LOGON_SCREEN Built–in If your non–ORACLE data source does not enforce security, you need only suppress the default Oracle Forms logon in an On–Logon trigger. If, however, you plan to enforce a security scheme, you may want to capture the current operator's username, password, and connect string to use when "logging on" to the non–ORACLE data source. You can do so with the LOGON_SCREEN built–in procedure.

LOGON_SCREEN causes Oracle Forms to display the default Runform logon screen. The logon screen has fields for an operator to enter a username, password, and connect string. Once an operator accepts the logon screen, these values are known to Oracle Forms, and you can obtain them programmatically with a call to GET_APPLICATION_PROPERTY.

The following example trigger illustrates this sequence:

On–Logon Trigger:

```
/* Display the logon screen to get the operator's
username, password, and connect string */
Logon_Screen;

/* Ask Oracle Forms for the username, password, and
connect string that were entered
*/
:control.op_name := Get_Application_Property(USERNAME);
:control.op_pw   := Get_Application_Property(PASSWORD);
:control.op_con  := Get_Application_Property(CONNECT_STRING);

/* Now call a user exit that "connects" to the non–ORACLE
data source */
User_Exit('my_connect');
```

The user exit code includes an EXEC TOOLS GET statement that reads the values of the control items containing the username, password, and connect string and then uses those values to "log on" to the non–ORACLE data source.

Note: A user exit can read the values of form bind variables, including items, global variables, and form parameters. A user exit cannot read a local PL/SQL variable in a form. However, you can assign the value of a local PL/SQL variable to a NULL canvas item and then refer to that value in the user exit. Another method would be to pass parameters from the form to the user exit as part of the user exit command string.

Determining the Data Source You can use the built–in function GET_APPLICATION_PROPERTY (DATASOURCE) to determine the data source to which Oracle Forms is currently connected (ORACLE, DB2, etc.). It is important to note that this property identifies the default connection—either to ORACLE or to an Open Gateway connection. When there is no default connection, for example, when there is an On–Logon trigger that replaces the default logon with a ”logon” to a non–ORACLE data source, the return value of GET_APPLICATION_PROPERTY(DATASOURCE) is NULL.

For more information, refer to the following flow charts in Chapter 8 of the *Oracle Forms Reference Manual, Vol. 2:*

- Logon
- Logout

Count Query Processing

Count query processing refers to the sequence of events that occurs when the operator or the application initiates a count query hits operation, either with a default key or menu command, or through a call to the COUNT_QUERY built–in procedure.

Operators use the count query hits feature to find out how many records meet the current query criteria, without actually fetching those records from the data source. When the operator issues the Count Query Hits command, Oracle Forms displays message `FRM–40355: Query will retrieve <n> records.`

The first part of Count Query processing is similar to the first part of Query and Fetch processing. Both require Oracle Forms to initialize the example record, clear the block, and prepare the query. In a Count Query operation, Oracle Forms then determines how many records meet the query criteria and displays that number in a message on the message line. In Query and Fetch processing, Oracle Forms identifies the records and then fetches them into the block as needed.

During a Count Query operation, the following transactional triggers fire if they are present:

- Pre–Query
- Pre–Select
- On–Count

Of these, the On–Count trigger is required if you want to implement the count query hits feature against a non–ORACLE data source. (The Pre–Query and Pre–Select triggers are used to augment default query processing, rather than directly replace it. Note that these triggers also fire during normal Query and Fetch processing.)

On–Count Trigger To replace default count query processing, the code in the On–Count trigger must do the following:

- Call a user exit that interacts with the non–ORACLE data source to determine the number of records that match the current query criteria.
- Call SET_BLOCK_PROPERTY to set the value of the Query_Hits block property to the number of records identified by the user exit.

For example, your On–Count trigger might call a user exit that identifies the number of records that meet the query criteria and then executes the EXEC TOOLS SET statement to set the value of a global variable in the form accordingly. The global variable can then be referenced in a call to SET_BLOCK_PROPERTY:

```
User_Exit('get_count');
Set_Block_Property('block3',QUERY_HITS,:global.record_count);
```

When the On–Count trigger completes execution, Oracle Forms issues the standard query hits message, using the value of the Query_Hits block property to indicate the number of records identified by the query criteria:

```
FRM–40355: Query will retrieve <value of Query_Hits block
property> records.
```

Oracle Forms will display the query hits message even if the On–Count trigger fails to set the value of the Query_Hits block property. In such a case, the message reports 0 records.

If you want to augment, rather than replace, default Count Query processing, you can call the COUNT_QUERY built–in from the On–Count trigger.

Note: You can also call GET_BLOCK_PROPERTY to examine the current value of the Query_Hits property. Be aware, however, that the Query_Hits setting is interpreted differently depending on where you examine it:

- In an On–Count trigger, Query_Hits specifies the number of records identified by the query criteria.
- During fetch processing (outside an On–Count trigger), Query_Hits specifies the number of records that have been placed on the block's list of records so far.

For more information on Count Query processing, refer to the following flow charts in Chapter 8 of the *Oracle Forms Reference Manual, Vol. 2.*

- Check Block for Query
- Count Query
- Prepare the Query

Query and Fetch Processing

Query and fetch processing refers to the sequence of events that occurs when the operator or the application initiates a query, either with a default key or menu command, or through a call to the EXECUTE_QUERY built–in procedure.

To understand Query and Fetch processing, it is important to distinguish between the querying phase, or *selection*, and the *fetch* phase. Selection is the operation that identifies records in the data source that match the current query criteria. Fetching is the operation that actually retrieves those records from the data source and places them on the block's list of records as needed.

During Query and Fetch processing, the following transactional triggers fire if they are present:

Selection Phase:

- Pre–Query
- Pre–Select
- On–Select
- Post–Select

Fetch Phase:

- On–Fetch (fires as many times as needed to fetch all records)
- Post–Query (fires once for each record placed on the block's list of records)
- On–Close

Of these, the On–Select and On–Fetch triggers are required when you implement support for a non–ORACLE data source.

The On–Select trigger is responsible for constructing a query based on the current example record, then executing it against the non–ORACLE datasource to identify those records that match the query criteria.

The On–Fetch trigger is responsible for determining how many records are required by the block, creating the appropriate number of records on the block's waiting list (using the data that were identified by the On–Select trigger and then retrieved from the non–ORACLE data source), and signaling to the form when all of the records have been fetched.

A query remains "open" until all of the records identified by the query criteria have been fetched, or until the operator or the application aborts the query.

While the query remains open, the On–Fetch trigger continues to fire whenever the form needs more records to be placed on the block's list of records. For example, as the operator scrolls down through the block's list of records, the On–Fetch trigger will fire as many times as necessary to fetch more records to be displayed in the block.

Note: Default query and fetch processing is described in the following flow charts in the *Oracle Forms Reference Manual, Vol. 2*:

- Check Block for Query
- Open the Query
- Prepare the Query
- Fetch Records

Selection Processing The On–Select trigger fires at the point during default processing when Oracle Forms normally constructs, opens, parses, and executes a query. When you implement support for a non–ORACLE data source, the code in your On–Select trigger replaces this default functionality

By default, query and fetch processing is a two–step process, one step to identify the records in the data source, and another to fetch them into the form. This approach may or may not be appropriate for a non–ORACLE data source application. As long as the code in the On–Fetch trigger creates enough records to meet the requirements of the block, it is up to you when and how the data for those records are actually retrieved from the non–ORACLE data source.

One method would be to have the user exit called in the On–Select trigger actually retrieve some or all of the records, rather then deferring that operation until the Fetch phase. Using this technique, the code in the On–Select trigger might create a client–side record cache from which records would be subsequently read into the form by the user exit called from the On–Fetch trigger.

Or, you might use the On–Select trigger only to suppress default Selection processing, and do all of your query and fetch processing in the On–Fetch trigger. The specifics of your implementation will depend on the characteristics of the data source to which you are connecting.

For some applications, you may need to know what type of query Oracle Forms is processing. You can find out by using GET_BLOCK_PROPERTY to read the value of the Query_Options property. The Query_Options property is read–only, and is set at runtime to one of the following values:

COUNT_QUERY	Indicates that the current query operation is a count query operation. This value occurs only when Query_Options is examined from within an On–Count trigger.
FOR_UPDATE	Indicates that the FOR_UPDATE option was specified for EXECUTE_QUERY. By default, FOR_UPDATE causes Oracle Forms to attempt an immediate lock on records being selected.
VIEW	Indicates that the query is being processed against a view, rather than a table. By default, this causes Oracle Forms to issue its default SELECT statement without using row IDs. Because the use of row IDs is unique to ORACLE, this is usually not relevant when running against a non–ORACLE data source.
NULL	Indicates that no query is being processed.

Fetch Processing The On–Fetch trigger replaces Oracle Forms default fetch processing, and must perform the following actions:

- determine how many records to fetch
- create that number of records on the block's waiting list
- signal to the form when there are no more records to fetch so that the query can be closed

The following example shows an On–Fetch trigger that populates an employee block named *emp*. The *emp* block contains three text items, *empno*, *ename*, and *sal*.

On–Fetch Trigger:

```
DECLARE
   recs      NUMBER;
   emp_id    NUMBER;
   emp_name  VARCHAR2(40);
   emp_sal   NUMBER;
BEGIN
   /* Determine how many records are needed */
   recs := Get_Block_Property('emp', RECORDS_TO_FETCH);

   /* Attempt to fetch that many records from the data source */
   FOR j IN 1..recs LOOP

      /* If a row is retrieved, then create a queried record
      for it and populate the record with data values retrieved
      from the non–ORACLE data source /*
      IF fetch_row(emp_id, emp_name, emp_sal) THEN
         Create_Queried_Record;
         :emp.empno := emp_id;
         :emp.ename := emp_name;
         :emp.sal := emp_sal;
      END IF;
   END LOOP;
END;
```

Determining How Many Records to Fetch The first requirement for the On–Fetch trigger is to determine how many records are required by the block. As shown in the example, this is accomplished by examining the value of the Records_To_Fetch block property:

```
/* Determine how many records are needed */
recs := Get_Block_Property('emp', RECORDS_TO_FETCH);
```

The Records_To_Fetch block property is a runtime, read–only property whose value is set internally by Oracle Forms, based on how many records the form has requested for the block. The first time the On–Fetch trigger fires during a query, the Records_To_Fetch property is set to the array size or to the number of records displayed + 1, whichever is greater.

Note: A block's default array size is specified by the Records Fetched block property. The number of records a block can display is specified by the Records Displayed block property, and determines whether a block is a single– or multi–record block.

Creating Queried Records Once the number of records required is known, the On–Fetch trigger must create that number of records on the block's *waiting list*. The waiting list is an intermediary record buffer that contains records that have been fetched from the data source, but have not yet been placed on the block's list of active records.

Creating queried records is usually accomplished with a loop that uses the Records_To_Fetch value as its index, as shown in the example:

```
/* Attempt to fetch that many records from the data source */
FOR j IN 1..recs LOOP

   /* If a row is retrieved, then create a queried record
   for it and populate the record with data values retrieved
   from the non–ORACLE data source /*
   IF fetch_row(emp_id, emp_name, emp_sal) THEN
      Create_Queried_Record;
      :emp.empno := emp_id;
      :emp.ename := emp_name;
      :emp.sal := emp_sal;
   END IF;
END LOOP;
```

The built–in procedure CREATE_QUERIED_RECORD is called once inside the loop for each record required by the block. CREATE_QUERIED_RECORD creates a record on the block's waiting list. The new record is essentially an empty place–holder. To populate the empty record with data from the non–ORACLE data source, the data values for each column in the fetched record must be assigned to the corresponding fields in the new queried record, using standard bind variable syntax. You must do the assignment immediately after the record is created; it is not possible to create a "batch" of records and then subsequently try to populate them.

Notice that the previous example does not show how records from the non–ORACLE data source are retrieved and managed before their values are placed into queried records on the block's waiting list. Again, the specifics of your implementation will depend on the data source to which you are connecting, and on how you choose to manage these operations.

In most implementations, a call to a user exit is made from inside the loop in the On–Fetch trigger. The user exit is responsible for getting a record from the non–ORACLE data source, assigning column values from the fetched record to form variables, and then returning control to the On–Fetch trigger so that the trigger code can create a queried record on the block's waiting list and populate it with the values from the non–ORACLE data source.

As you can see, this technique requires that the user exit and Oracle Forms be tightly coupled. Specifically, you need the ability to set and get the values of form bind variables from within the user exit, and the ability to pass control between the form trigger and the user exit without losing the current context of either. To make this sort of interaction possible, Oracle has included the following commands that you can execute in user exits that you create with an Oracle Precompiler:

- EXEC TOOLS GET
- EXEC TOOLS SET
- EXEC TOOLS GET CONTEXT
- EXEC TOOLS SET CONTEXT

For information on these commands, see *Oracle Forms Advanced Techniques,* Chapter 3, “User Exit Interface to Foreign Functions.”

Signaling an End–of–Fetch As mentioned previously, a query remains ”open” until all of the records identified by the query criteria have been fetched, or until the operator or the application aborts the query. When the form requests more records to be fetched from an open query (because, for example, the operator scrolls to the end of the block’s list of records), the On–Fetch trigger will fire as necessary to meet the demand for more records.

Eventually, however, all of the records that match the query criteria will have been placed on the block’s list of records by the On–Fetch trigger. When this occurs, the On–Fetch trigger needs a way to signal to the form that no more records are available so that the On–Fetch trigger stops firing. Oracle Forms uses the following mechanism to accomplish this:

When the On–Fetch trigger fires successfully but does not create any queried records (by executing CREATE_QUERIED_RECORD), Oracle Forms assumes there are no more records to be fetched, and so closes the query and does not fire the On–Fetch trigger again.

In the previous example trigger, you saw that a queried record was created only if the user–named, BOOLEAN function called *fetch_row* returned TRUE. In an actual application, the *fetch_row* function might call a user exit to determine if another row is available in the non–ORACLE data source. If fetch row returns TRUE, that is, if there was another row to be fetched, then CREATE_QUERIED_RECORD creates a record. Otherwise, the loop executes without creating any more records.

```
FOR j IN 1..recs LOOP

   /* If a row is retrieved, then create a queried record
   for it and populate the record with data values retrieved
   from the non–ORACLE data source /*
   IF fetch_row(emp_id, emp_name, emp_sal) THEN
      Create_Queried_Record; –– only if there is another
      :emp.empno := emp_id;  –– row in the data source
      :emp.ename := emp_name;
      :emp.sal := emp_sal;
   END IF;
END LOOP;
```

Oracle Forms also looks at the number of records created by the On–Fetch trigger and sets the value of the Records_To_Fetch block property accordingly:

- The first time the On–Fetch trigger fires for a query, the value of Records_To_Fetch is set to the array size or to the number of records displayed + 1, whichever is greater.
- If the On–Fetch trigger creates this many queried records, the next time the On–Fetch trigger fires, the value of Records_To_Fetch will be the same number.
- If the On–Fetch trigger creates fewer records than specified by Records_To_Fetch, Oracle Forms fires the On–Fetch trigger again immediately, and sets Records_To_Fetch to the previous value minus the number of queried records created by the previous execution of the On–Fetch trigger.

Note: The On–Fetch trigger must never create more queried records than are required by the form (as indicated by the Records_To_Fetch property). Doing so will irretrievably disrupt form processing. (You can raise the built–in exception FORM_TRIGGER_FAILURE when fetch errors occur in the On–Fetch trigger to return to normal form processing.)

Closing the Query By default, Oracle Forms closes a query when all of the records have been fetched or the operator or the application aborts the query. In a non–ORACLE data source application, no action is required by you to explicitly close a query.

In certain cases, however, you might need to ”close,” or clean up, the connection to your non–ORACLE data source. For example, if the user exit that was called from the On–Fetch trigger was maintaining a record cache or some other context, you might want to free up those resources when they are no longer required because the query has been closed.

You can accomplish operations of this type in an On–Close trigger. Note that unlike other On–event triggers, the On–Close trigger actually augments, rather than replaces, the default close query operation. (Oracle Forms must always close a query at the appropriate time to avoid an inconsistent state in the form.)

Commit Processing

Commit Processing refers to the sequence of events that occurs when the operator or the application initiates a database commit operation, either with a key or menu command, or by executing the COMMIT_FORM built–in procedure.

Default commit processing happens in three parts:

- **Validation** When the operator or the application initiates a commit operation, Oracle Forms validates the records on each base table block's list of records.
- **Posting** During posting, Oracle Forms moves to each block in the form and writes any pending updates, inserts, and deletes to the database.
- **Committing** By default, Oracle Forms issues a savepoint at the start of a commit operation. Changes that have been posted to the database but not yet committed can be rolled back to this savepoint. To finalize the commit transaction, Oracle Forms must explicitly issue the COMMIT statement. This happens as the final step of Commit processing.

For a detailed description of default commit processing, refer to the following flow charts in Chapter 8 of the *Oracle Forms Reference Manual, Vol. 2*:

- Check Record Uniqueness
- Post and Commit Transactions
- Savepoint
- Validate the Form

During Commit processing, the following transactional triggers will fire as necessary if they are present:

- On–Savepoint (fires once at the beginning of the transaction)
- Pre–Commit (fires once at the beginning of validation)
- Pre–Delete / On–Delete / Post–Delete (fires once for each record being deleted)

- Pre–Update / On–Update / Post–Update (fires once for each record being updated)
- Pre–Insert / On–Insert / Post–Insert (fires once for each record being inserted)
- On–Check–Unique (may fire once for each record inserted or updated, depending on primary key constraints)
- Post–Forms–Commit (fires after posting, before the database commit)
- On–Commit (fires once at the end of the transaction)
- Post–Database–Commit (fires after On–Commit)

Of these, the following are always required in a non–ORACLE data source application that allows operators to modify queried records:

- On–Savepoint
- On–Delete
- On–Update
- On–Insert
- On–Commit

The On–Check–Unique trigger is also required if your application is verifying that each record has a unique primary key before updating or inserting the record in the data source. (This happens by default when the block has the Primary Key block property and one or more items in the block have the Primary Key item property set to True.)

When Oracle Forms is running against ORACLE, posting and committing are separate operations. If, however, your non–ORACLE data source does not support this functionality, changes you write to the data source are final, and do not have to be committed as a separate operation. In this case, you might define an On–Commit trigger only to suppress the default Oracle Forms COMMIT statement, and write all of your changes to the data source in user exits called by the On–Delete, On–Update, and On–Insert triggers.

Note: The On–Savepoint trigger fires at the start of Commit processing, and is discussed in the topic ”Savepoint and Rollback Processing.”

Processing Inserts, Updates, and Deletes During commit processing, the On–Delete, On–Update, and On–Insert triggers fire once for each record in the block that was changed. These triggers replace the default processing that Oracle Forms would normally perform at that point. For example, the On–Update trigger fires when Oracle Forms would normally issue the appropriate SQL UPDATE statement to update the row in the database that corresponds to the record that was modified by the form operator.

To replace default processing, you need to write a separate user exit for the On–Delete, On–Update, and On–Insert triggers. These user exits are responsible for performing the appropriate action on the corresponding row in the non–ORACLE data source. Again, the specifics of your implementation will depend on the characteristics of your data source. The following On–Delete trigger calls a user exit named *kill_row* that is responsible for deleting the appropriate row from the non–ORACLE data source:

On–Delete Trigger:

```
User_Exit('kill_row');
```

Obviously, the code in the user exit needs to know which record is being processed by Oracle Forms so that it can act on the corresponding row in the non–ORACLE data source. This is made possible by the fact that Oracle Forms processes inserts, updates, and deletes sequentially, one record at a time. Within an On–Delete, On–Update, or On–Insert trigger, the record being deleted, inserted, or updated is always the *current record*. This means that you can use standard bind variable syntax (*:block_name.item_name*) to reference the values of items in the current record from within these triggers. (Remember that a user exit can read the values of form bind variables with the EXEC TOOLS GET statement.)

Checking for Unique Primary Keys You can designate one or more items in a block as primary key items by setting the Primary Key item and block properties to True. When these properties are set, Oracle Forms enforces primary key uniqueness by

- preventing updates to primary key columns in the base table
- preventing the insertion of records that contain duplicate primary key values

Oracle Forms checks the uniqueness of primary key values just before inserting or updating the record in the database. When a record has been *inserted* in a block, Oracle Forms will always perform the uniqueness check. When a record has been *updated*, Oracle Forms performs the uniqueness check only if one or more primary key item values were modified.

Oracle Forms checks the uniqueness of a record by constructing and executing the appropriate SQL statement to select for database rows that match the record about to be inserted or updated. If a row having a duplicate primary key is found, Oracle Forms displays message `FRM–40600: Row has already been inserted.` and disallows the insert or update. When this happens, the input focus remains in the offending record, allowing the operator to correct the problem before trying to commit again.

If your non–ORACLE data source application supports primary key uniqueness, you can replace Oracle Forms default uniqueness checking by calling an appropriate user exit from an On–Check–Unique trigger. When the Primary Key item and block properties are set to True, this trigger fires just before the On–Insert trigger and, when necessary (because a primary key value was modified), just before the On–Update trigger.

On–Check–Unique Trigger The code in the On–Check–Unique trigger might perform the following tasks:

- read the values of the primary key items in the current record
- compare those values against rows in the data source
- if a duplicate row is found, display an appropriate message or alert to inform the operator, then raise the built–in exception FORM_TRIGGER_FAILURE to explicitly cause the trigger to fail and abort commit processing

The following example shows what such a trigger might look like:

On–Check–Unique Trigger::

```
DECLARE
   duplicate   BOOLEAN;
BEGIN
   User_Exit('do_check');
   IF duplicate THEN
      Message('Primary key must be unique.  Unable to commit.');
      Bell;
      RAISE Form_Trigger_Failure;
   END IF;
END;
```

In this example, the user exit *do_check* checks for duplicate primary key values in the non–ORACLE data source, then sets the value of the BOOLEAN variable *duplicate* accordingly. Raising the FORM_TRIGGER_FAILURE exception causes the trigger to fail, which in turn causes Oracle Forms to abort commit processing and roll back to the last savepoint.

Savepoint and Rollback Processing

By default, Oracle Forms issues a savepoint at form startup, and at the beginning of each Commit process. When necessary, Oracle Forms issues the appropriate SQL rollback statement to undo changes that were posted since the last savepoint was issued. If your application will run against a non–ORACLE data source that does not support savepoints, you can suppress the default savepoint and rollback statements issued by Oracle Forms by setting the Savepoint Mode form module property to False.

Note: When Savepoint Mode is False, Oracle Forms does not allow a form that has uncommitted changes to invoke another form with the CALL_FORM procedure. While this behavior is usually appropriate when running against ORACLE, it may be undesirable when running against a non–ORACLE data source, even one that does not support savepoints. To prevent this situation, you might choose to suppress savepoint processing by leaving the Savepoint Mode property to True, but including the NULL statement in the On–Savepoint and On–Rollback triggers.

When the Savepoint Mode form module property is left True (the default), Oracle Forms attempts to issue savepoints and rollbacks as necessary. In a non–ORACLE data source application, you will need to define On–Savepoint and On–Rollback triggers to replace the default processing that normally occurs during these events.

By default, Oracle Forms manages savepoint names internally. Savepoint names are in the format FM_<*number*>, where *number* is an integer value from a counter that increments each time a savepoint is issued, and decrements when a rollback occurs. For example, when Form_A calls Form_B with CALL_FORM, Oracle Forms issues savepoint FM_*n*. If the operator then initiates a commit in Form_B, Oracle Forms issues savepoint FM_*n+1*. If an error occurs during the commit, Oracle Forms attempts to roll back to savepoint *n + 1*.

When you implement custom savepoint functionality by writing On–Savepoint and On–Rollback triggers, you can capture the savepoint names that Oracle Forms would use by default by calling GET_APPLICATION_PROPERTY to examine the current value of the Savepoint_Name property:

```
my_savepoint := Get_Application_Property(SAVEPOINT_NAME);
```

The value of Savepoint_Name depends on whether you examine it from an On–Savepoint or On–Rollback trigger:

- In an On–Savepoint trigger, Savepoint_Name returns the name of the savepoint that Oracle Forms would be issuing by default, if no On–Savepoint trigger were present.
- In an On–Rollback trigger, Savepoint_Name returns the name of the savepoint to which Oracle Forms would roll back, if no On–Rollback trigger were present.

In an application that runs against both ORACLE and a non–ORACLE data source, you can use the ISSUE_SAVEPOINT built–in to issue the correct savepoint from an On–Savepoint trigger:

```
IF Get_Application_Property(DATASOURCE) = 'ORACLE' THEN
   Issue_Savepoint(Get_Application_Property(SAVEPOINT_NAME);
ELSE
   User_Exit('non_ora');
END IF;
```

Note: The value of Savepoint_Name is undefined outside an On–Savepoint or On–Rollback trigger.

Check Column Security Processing

Check Column Security processing refers to the sequence of events that occurs when Oracle Forms enforces column–level security for each block that has the Column Security block property set to True. To enforce column security, Oracle Forms does the following:

- queries the database to determine the base table columns to which the current form operator has update privileges
- for columns to which the operator does not have update privileges, Oracle Forms makes the corresponding base table items in the form non–updateable by setting the Update Allowed item property to False dynamically

By default, Oracle Forms performs these steps at form logon, processing each block in sequence.

If your non–ORACLE data source application does not require column–level security, you can suppress the default processing by making sure that the Column Security property is set to False for each block in the form, or by defining an On–Column–Security trigger that suppresses default processing.

On–Column–Security Trigger If you want to implement a security check comparable to Oracle Forms default processing, you must define an On–Column–Security trigger. The code in the On–Column–Security trigger might do the following:

- call GET_APPLICATION_PROPERTY to get the current form operator's username and password
- interact with the non–ORACLE data source to determine the columns to which the current operator has update privileges
- call SET_ITEM_PROPERTY to set the Update_Permission property to False for any items that operators should not be allowed to modify (when Update_Permission is False, operators are not allowed to update the item, and its value is not included in any UPDATE statement generated by Oracle Forms.)

The On–Column–Security trigger fires once for each block that has the Column Security property On, and can be defined at either the form level or block level. Note, however, that when you define this trigger at the form level, there is no explicit way to determine which block Oracle Forms is currently processing. (Because the On–Column–Security trigger fires at startup, before the form is instantiated, the value of SYSTEM.CURRENT_BLOCK is still undefined.) In this case, you must keep track of the current block yourself, based on the fact that blocks with the Column Security property set to True are processed sequentially, according to the sequence of the blocks in the form. (Block sequence is defined at design time by the order of blocks listed in the Object Navigator.)

When you define the On–Column–Security trigger at the block level, the current block context is, of course, readily apparent.

Generate Sequence Number Processing

Generate Sequence Number processing refers to the series of events that occurs when Oracle Forms interacts with the database to get the next value from a SEQUENCE object defined in the database. Although not strictly related to transaction processing, this operation requires database interaction, and so must be considered when you implement support for a non–ORACLE data source.

Sequences are often used to generate unique primary key values for records that will subsequently be inserted in the database. In a form, you can specify that an item's default value should be the next integer from a database sequence by setting the Default property to `:SEQUENCE.my_seq.NEXTVAL.`

When a SEQUENCE is used as a default item value, Oracle Forms queries the database to get the next value from the SEQUENCE whenever the Create Record event occurs. You can suppress or override this functionality with an On–Sequence–Number trigger.

In practice, it is unlikely that you would be referencing a SEQUENCE as a default item value in a non–ORACLE data source application. It is possible, however, that you would want to implement equivalent functionality, using values provided by some other source. In this case, it is often better to perform this operation in a When–Create–Record trigger.

Lock Record Processing

Lock Record processing refers to the sequence of events that occurs when Oracle Forms attempts to lock rows in the database that correspond to queried records in the form. By default, Oracle Forms attempts to lock a row immediately after an operator modifies an item value in a queried record; for example, as soon as the operator presses a key to enter or edit the value in a text item.

When Oracle Forms attempts to lock a row, the On–Lock trigger fires if present. You can use this trigger to replace or suppress default locking behavior.

You can also control default locking behavior by setting the Locking Mode block property, as discussed earlier in this chapter.

Accessing System Date

The default values $$DBDATE$$ or $$DBDATETIME$$ do not work when you are accessing a non–ORACLE datasource. Instead, use a When–Create–Record trigger to select the current date in a datasource–specific manner.

CHAPTER

5

Multiple–Form Applications

You can build applications by integrating multiple form, menu, and library modules. This chapter explains how to use multiple form modules in a single application. It includes the following topics:

About Multiple–Form Applications

A *multiple–form application* is one that is designed to open more than one form during a single Runform session. Every invocation of Runform begins the same way—by starting a single form module. Once the first form is loaded into memory and begins execution, it can programmatically invoke any number of additional forms. Those forms can, in turn, invoke still other forms. Modular application development can provide advantages at both design time and during deployment.

When one form programmatically invokes another, Oracle Forms looks for the new form in the appropriate directory and then loads it into memory. When you deliver a multiple–form application to end users, all of the .FMX, .MMX, and .PLL (form, menu, and library) files that will be called during the session must reside in the working directory or search path defined for your system.

Invoking Forms

There are three ways that one form can programmatically invoke another form:

- Execute the OPEN_FORM procedure to open an independent form.
- Execute the NEW_FORM procedure to replace the current form with a different form.
- Execute the CALL_FORM procedure to call a modal form.

When one form invokes another form by executing OPEN_FORM, the first form remains displayed, and operators can navigate between the forms as desired. An opened form can share the same database session as the form from which it was invoked, or it can create a separate session of its own. For most GUI applications, using OPEN_FORM is the preferred way to implement multiple–form functionality.

When one form invokes another form by executing NEW_FORM, Oracle Forms exits the first form and releases its memory before loading the new form. Calling NEW_FORM completely replaces the first form with the second. If there are changes pending in the first form, the operator will be prompted to save them before the new form is loaded.

When one form invokes another form by executing CALL_FORM, the called form is modal with respect to the calling form. That is, any windows that belong to the calling form are disabled, and operators cannot navigate to them until they first exit the called form.

Both OPEN_FORM and CALL_FORM allow you to leave the calling form displayed. Using this technique, forms can be integrated so tightly that operators are not aware they are invoking separate forms.

Any locks obtained by a form are maintained across both OPEN_FORM (in the same session), CALL_FORM and NEW_FORM procedure calls. Thus, a called form automatically has the same locks as its calling form.

CALL_FORM is an unrestricted procedure, OPEN_FORM and NEW_FORM are restricted. Therefore, CALL_FORM is valid in Enter Query mode, while OPEN_FORM and NEW_FORM are not.

Multiple–Form Applications and the Root Window

When you create multiple–form applications with OPEN_FORM or CALL_FORM, you will usually want to avoid defining a root window in any forms that will be displayed at the same time.

At runtime, only one root window can be displayed at a time, even in a multiple–form application. This means that if the first canvas–view displayed by Form B is a content view assigned to the root window, it will display in Form A's root window, rather than in a separate root window of its own. As a result, Form B's content view hides any Form A views already displayed in the root window. For most multiple–form applications, this is not the desired behavior, and using a root window should be avoided.

If circumstances require that a root window be used, you can prevent the second form from hiding the first by making sure that the target canvas–view displayed by the second form is one of the following:

- a content canvas–view or stacked canvas–view assigned to a window other than the root window
- a stacked canvas–view assigned to the root window

When the target canvas–view is assigned to a different window, Oracle Forms displays the second form in a separate window, and the first form's root window display remains unchanged.

When the target canvas–view is a stacked canvas–view assigned to the root window, Oracle Forms displays it in the first form's root window. However, if the stacked canvas–view is smaller than the content canvas–view (usually the case), the first form's content view will remain at least partially visible.

Invoking Independent Forms with OPEN_FORM

One form can programmatically open another by executing the OPEN_FORM built–in, as shown here:

```
Open_Form(’stocks’);
```

By default, the opened form is activated immediately and focus is set to the first navigable item in the form. You can specify this default behavior explicitly by including the optional ACTIVATE parameter in the call to OPEN_FORM:

```
Open_Form(’stocks’,ACTIVATE);
```

If you do not want the opened form to receive focus, call OPEN_FORM with the NO_ACTIVATE parameter, as shown here:

```
Open_Form(’stocks’,NO_ACTIVATE);
```

When you open a form with ACTIVATE specified (the default), focus is set to the opened form immediately, and any trigger statements that follow the call to OPEN_FORM are ignored and never execute.

When you open a form with NO_ACTIVATE specified, any trigger statements that follow the call to OPEN_FORM execute after the opened form has been loaded into memory and its initial start–up triggers (if any) have been fired.

Whether you open a form with ACTIVATE or NO_ACTIVATE, any triggers that would normally fire at form start–up will execute in the form being opened. This could potentially include Pre–Form, When–New–Form–Instance, When–New–Block–Instance, When–New–Record–Instance, and When–New–Item–Instance.

You can close an independent form with the CLOSE_FORM procedure, as shown here:

```
Close_Form(’stocks’);
```

Navigation Between Independent Forms

Navigation between forms in a multiple–form application can occur when the operator navigates with the mouse, or when a form calls one of the following navigational procedures:

- GO_FORM
- NEXT_FORM
- PREVIOUS_FORM

The GO_FORM procedure takes one parameter that specifies the name of the target form:

```
Go_Form('schedule');
```

NEXT_FORM and PREVIOUS_FORM navigate to the next or previous form in the sequence defined by the order the forms were opened at runtime:

```
Next_Form;
Previous_Form;
```

Many triggers fire in response to navigational events, including Pre– and Post– triggers (Pre–Block, Post–Record, etc.) and When–New–Instance triggers (When–New–Block–Instance, When–New–Item–Instance, etc.). When you build an application with multiple forms, it is important to understand how trigger processing is affected by navigation.

In a multiple–form application, each open form has one item that is the *current item* for that form. If you initiate navigation to an open form programmatically with GO_FORM, NEXT_FORM, PREVIOUS_FORM, or EXIT_FORM, the target item is always the current item in the target form.

For example, when Form A opens and activates Form B, focus is set to the current item in Form B. If Form B subsequently calls EXIT_FORM or PREVIOUS_FORM, the focus returns to the current item in Form A—in this case, the item that was current when Form A opened Form B.

Keep in mind the following points about inter–form navigation:

- When navigating between independent forms, no validation occurs in the starting form. It is possible to navigate out of a field that is currently invalid provided that the target field is in a different form. Upon returning to the starting form and attempting to navigate within that form, normal validation is enforced.
- When navigating between independent forms, no triggers fire. The only exceptions are When–Window–Deactivated, which fires in the form that initiates navigation, and When–Window–Activated, which fires in the target form. The Pre–, Post–, and When–New–Instance triggers do not fire when navigating between forms.

When the operator navigates from Form A to Form B by clicking with the mouse, the target item can be either the current item in Form B or an item other than the current item in Form B.

If the operator clicks on the *current item* in Form B, no triggers fire. If the operator clicks on an item *other than the current item*, only triggers that would normally fire when navigating from the current item in Form B to the target item fire, and validation occurs as required. Thus, form processing proceeds exactly as it would if the operator were navigating from the current item in Form B to the target item, without regard to any external forms.

Opening Forms in Different Database Sessions

At runtime, Oracle Forms automatically establishes and manages a single connection to ORACLE. By default, one user session is created for this connection. However, the multiple sessioning feature of ORACLE allows a single client to establish multiple sessions within a single connection. All of ORACLE's transaction management and read–consistency features are implemented at the session level, so creating multiple sessions allows a single user to have multiple, independent transactions.

Record locking and read consistency behavior for two forms in different sessions is the same as it would be for two independent clients with separate connections. When two independent forms access the same table, it is possible for one form to obtain a lock that would prevent the other form from accessing records in the table.

In a multiple–form application, you have the option to create an independent session whenever one form opens another form with the OPEN_FORM procedure. By default, Oracle Forms does not issue a savepoint when you open a form, and the opened form shares the same session as the form from which it was opened.

The following examples are equivalent, and open a form without creating a new session:

```
Open_Form('stocks'); –– default; NO_SESSION is implicit
Open_Form('stocks',ACTIVATE,NO_SESSION) –– explicit; for clarity
```

To open a form in its own, independent session, call OPEN_FORM with the SESSION parameter, as shown here:

```
Open_Form('stocks',ACTIVATE,SESSION);
```

This statement opens the STOCKS form in its own session, and sets focus to it immediately.

When a COMMIT is initiated in any form, Oracle Forms does validation and commit processing for each open form that shares the same session as that form. Forms are processed in the sequence defined by the order in which they were opened, starting with the form that initiated the

COMMIT. If an error occurs during commit processing, the input focus is returned to the form that initiated the COMMIT.

When you call OPEN_FORM with the NO_SESSION parameter (the default), Oracle Forms does not issue a savepoint for the form as it does when you use CALL_FORM. If Form A opens Form B, both forms share the same session. If Form B executes CLEAR_FORM (which does an implicit ROLLBACK), all of the changes that were made in both Form A and Form B will be rolled back.

Creating independent sessions is usually appropriate for forms that access different tables, and that manage transactions that are logically independent. For example, an order entry form might have a menu option to invoke an appointment calendar form. If the appointment calendar is opened in the same session as the order entry form, the operator will have to commit any changes made to the order entry and calendar forms at the same time. If, however, the calendar form is opened in a separate session, the operator can enter a new appointment and save it to the database independently, without having to commit in the order entry form.

Note: Oracle Forms Runform must be running with the Session option turned On when you execute OPEN_FORM with the *session_mode* parameter set to SESSION. If the Session option is Off, Oracle Forms issues an error and does not open the indicated form. You can set session On for all Runform invocations by setting the FORMS45_SESSION environment variable to TRUE. When you set the FORMS45_SESSION variable, all Runform invocations inherit its setting, unless you override the environment variable by setting the Session option from the Runform command line.

Opening Multiple Instances of the Same Form

It is common in multiple–form applications to allow operators to open multiple instances of the same form. This technique is useful for applications that allow operators to perform similar functions on different record sets, particularly when it is useful for each form to be managed within a separate transaction session.

When you programmatically open the same form *n* times, you have *n* instances of that form displayed on the screen and stored in memory. When you need to refer to a specific instance of a form, for example, when calling GO_FORM or CLOSE_FORM, you must use the internally stored ID of the form instance, rather than the name of the form (which is common to all instances of the same form).

Once a form has been opened, you can obtain its internal ID with the built–in FIND_FORM function:

```
DECLARE
    form_id   FormModule;
BEGIN
    Open_Form('stocks',NO_ACTIVATE);
    form_id := Find_Form('stocks');
    :GLOBAL.formid := To_Char(form_id.id);
END;
```

In this example, a variable of type FormModule is declared. (FormModule is one of the native Oracle Forms types.) Once the STOCKS form is opened, its internal ID is assigned to the variable by calling the FIND_FORM function. Then, the *instance identifier* is converted to CHAR and assigned to a global variable with the syntax:

```
:GLOBAL.formid := To_Char(form_id.id);
```

Once you have stored the instance identifier in a global variable, you can retrieve it at any time for operations on the specific form instance. The following example passes the instance identifier to the built–in procedure GO_FORM:

```
DECLARE
    form_id   FormModule;
BEGIN
    form_id.id := To_Number(:GLOBAL.formid);
    Go_Form(form_id);
END;
```

The GO_FORM procedure takes a parameter of type FormModule, so the first step is to declare a variable of that type to hold the instance ID currently held in the global variable. Notice also that the instance identifier must be cast to NUMBER to be a valid FormModule value.

By combining the use of global variables with the built–in COPY procedure, you can store the instance identifier for several different forms, as shown here:

```
DECLARE
    form_id FormModule;
    temp    CHAR(10);
BEGIN
    –– Initialize a counter, open the STOCKS form, and
    –– store its internal ID in form_id
    ––
    Default_Value('1','GLOBAL.counter');
    Open_Form('stocks',NO_ACTIVATE);
    form_id := Find_Form('stocks');
    ––
    –– The Copy() procedure reads only global variables
    –– and item values, so put the instance identifier in
    –– in a a temporary global variable
    ––
    :GLOBAL.temp_var := To_Char(form_id.id);
    ––
    –– Now assign the ID to the next occurrence of the global
    –– variable :GLOBAL.form<x>, where x is the counter value
    ––
    Copy(Name_In('GLOBAL.temp_var'),
                     'GLOBAL.form'||:GLOBAL.counter);
    :GLOBAL.counter := To_Number(:GLOBAL.counter)+1;
END;
```

This example creates a set of global variables (:GLOBAL.form1, :GLOBAL.form2 ... GLOBAL.form*n*) which contain the instance identifiers for each open form. You can then use the appropriate variable for programmatic operations on a specific form.

Replacing the Current Form with NEW_FORM

You can replace the current form in an application by executing the NEW_FORM built–in procedure. The following specification shows the formal parameters for NEW_FORM:

```
New_Form(formmodule_name   CHAR
      [,rollback_mode      NUMBER,
       query_mode          NUMBER,
       parameterlist_name  CHAR]);
```

If you leave the optional parameters unspecified, Oracle Forms runs the new form using the defaults for those parameters. Hence,

```
New_Form('form_B');
```

is logically equivalent to

```
New_Form('form_B',TO_SAVEPOINT,NO_QUERY_ONLY);
```

If the calling form was itself a called form invoked with the CALL_FORM procedure, the new form assumes the parent form's position in the call stack. Further, Oracle Forms runs the new form with the same CALL_FORM parameters (HIDE or NO_HIDE, DO_REPLACE or NO_REPLACE, and QUERY_ONLY or NO_QUERY_ONLY) as the calling form.

Calling Modal Forms with CALL_FORM

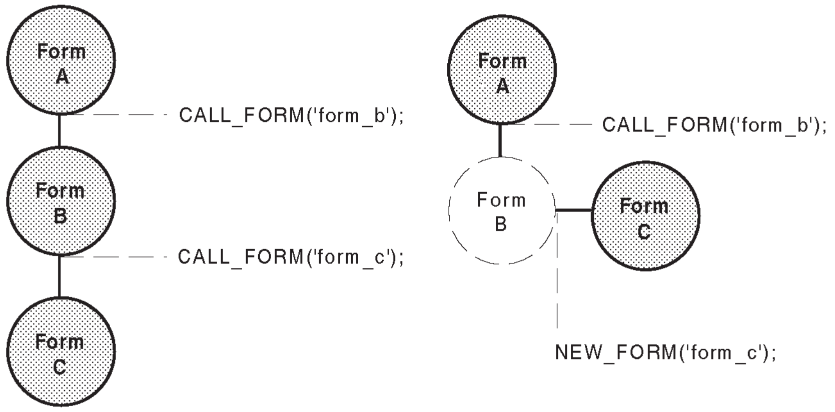
One form can programmatically invoke, or *call,* another form by executing the CALL_FORM built–in procedure. For example, Form A can invoke Form B with the following procedure call:

```
Call_Form('form_B');
```

CALL_FORM loads the indicated form while leaving the calling form loaded. For example, when Form A calls Form B, Form B becomes the active form in the session, but Form A remains in memory. If the operator exits Form B, Form A again becomes the active form. Form B, in turn, can execute the CALL_FORM procedure to invoke Form C. When successive forms are loaded via the CALL_FORM procedure, the resulting module hierarchy is known as the *call form stack.*

When a form calls another form with CALL_FORM, the called form is modal with respect to the calling form. Windows that belong to the calling form are disabled, and operators cannot navigate to them until the operator exits the called form.

In contrast to the CALL_FORM procedure, NEW_FORM exits the current form and releases its associated memory before running the new form at the current position in the call form stack.



The following specification shows the formal parameters and default values for the built–in procedure CALL_FORM:

```
Call_Form(formmodule_name  CHAR
       [,display           NUMBER := HIDE,
        switch_menu        NUMBER := NO_REPLACE,
        query_mode         NUMBER := NO_QUERY_ONLY,
        parameterlist_name  CHAR]);
```

The only required parameter is *formmodule_name*. If you leave the optional parameters unspecified, Oracle Forms runs the called form using the default values for those parameters. Hence,

```
Call_Form('form_B');
```

is logically equivalent to

```
Call_Form('form_B',HIDE,NO_REPLACE,NO_QUERY_ONLY);
```

When you execute CALL_FORM, you can specify whether the calling form should remain displayed by passing an appropriate constant for the *display* parameter. Valid constants are HIDE (the default) and NO_HIDE.

The following procedure call invokes Form B and leaves Form A (the calling form) displayed:

```
Call_Form('form_B',NO_HIDE);
```

Only the called form is active, and operators cannot navigate to items in the calling form until they exit the called form.

Exiting from a Called Form

By default, Oracle Forms exits a form when any of the the following operations occurs:

- EXIT_FORM procedure
- NEW_FORM procedure
- Exit command on the default Action menu
- [Exit/Cancel] key

When a called form exits, control returns to the calling form, and processing resumes where it left off. This means that the trigger, menu item command, or user–named routine that executed the CALL_FORM procedure resumes processing at the statement immediately following the CALL_FORM procedure call.

Allowing Operators to Quit from a Called Form

It is often desirable to allow operators to quit an application from a called form, rather than requiring them to explicitly exit each form in the call stack. You can accomplish this by using a global variable that indicates whether the operator wants to quit the entire session or return to the calling form. (Global variables are visible across called forms.)

For example, the called form might have a button labeled QUIT that allows operators to quit the entire application from a called form. The When–Button–Pressed trigger for each button sets the value of a global variable to indicate that the operator wants to either quit the session or return to the calling form.

When–Button–Pressed Trigger on QUIT button in Called Form B:

```
:GLOBAL.quit_check := ’quit’;
Exit_Form;
```

Then, in the calling form, read the value of the global variable immediately after the CALL_FORM procedure statement:

Trigger Text in Calling Form A:

```
Call_Form(’form_B’);
/*
** The following statements execute immediately after
**  returning from the called form.
*/
IF :GLOBAL.quit_check = ’quit’ THEN
   Exit_Form;
END IF;
```

Calling a Form in Query–Only Mode

When you call a form by executing the built–in procedures CALL_FORM or NEW_FORM, you can specify whether the called form should run in normal mode or query–only mode. A form in query–only mode can query the database but cannot perform inserts, updates, or deletes.

You specify the query mode for a called form by supplying the appropriate constant for the *query_mode* parameter in the CALL_FORM or NEW_FORM argument list. Valid constants are NO_QUERY_ONLY (normal mode) and QUERY_ONLY.

For example,

```
Call_Form(’form_B’,NO_HIDE,NO_REPLACE,QUERY_ONLY);
```

or

```
New_Form(’form_B’,TO_SAVEPOINT,QUERY_ONLY);
```

Note: Oracle Forms runs any form called from a form in query–only mode as a QUERY_ONLY form, even if the CALL_FORM or NEW_FORM syntax specifies that the called form is to run in NO_QUERY_ONLY (normal) mode.

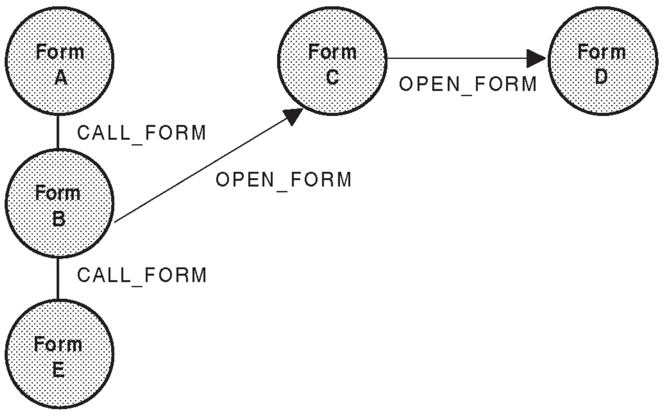
Using CALL_FORM with OPEN_FORM

When you invoke a modal form by executing CALL_FORM, the calling form is disabled until the operator exits the called form and returns to the calling form. When a calling form is disabled, its windows are grayed out, and operators are unable to set focus to items in the form. A called form can in its turn call another form. The result is three forms loaded in memory, only one of which is active and available to the operator. When successive forms are loaded via the CALL_FORM procedure this way, the resulting module hierarchy is known as the *call form stack*.

When you invoke multiple forms with OPEN_FORM and CALL_FORM in the same application, there are certain restrictions you should be aware of:

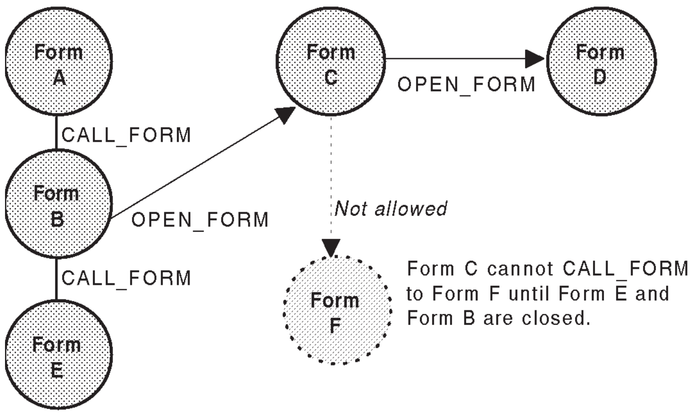
- **Navigation:** Any attempt to navigate programmatically to a disabled form in a call form stack is disallowed.
- **Calling Forms:** An open form cannot execute the CALL_FORM procedure if a chain of called forms has been initiated by another open form.
- **Clear All/Rollbacks:** When a form is invoked with CALL_FORM, Oracle Forms issues a savepoint. Any subsequent ROLLBACK, in any active form, will roll back only changes that were made since the last savepoint was issued; that is, since the execution of CALL_FORM.

Restrictions on Navigation Consider the following example. Form A executes CALL_FORM to call Form B. Form B then executes OPEN_FORM to open Form C. Form C then opens Form D. Form B then calls Form E.



Restrictions on Calling Forms At this point in the example, there are three active, navigable forms (E, C, and D), and two disabled, non–navigable forms (A and B). Any attempt to navigate programmatically to Form A or Form B will raise an error.

Together, Form A, Form B, and Form E represent the current call form stack in the application. If Form C attempted to call a form with CALL_FORM, an error would occur because of the restriction that an open form cannot issue a CALL_FORM when there is an existing stack of called forms that was initiated by another form (Form A, in this example).



Before Form C can successfully execute CALL_FORM, the operator or the application would have to exit Form E and return to Form B, then exit Form B to close out the call form stack.

If Form D exits, focus returns to the form from which Form D was opened, in this case, Form C. If Form C then exits, focus returns to Form E, even though Form B originally opened Form C. This is because Form B is currently disabled as a result of having issued a CALL_FORM to invoke Form E.

Restrictions on Clear All/Rollback Oracle Forms issues a savepoint whenever a form executes CALL_FORM to invoke a modal form. When there are multiple open forms running in the same session, each transaction event, *in any form*, is part of the same continuum. This means that if the operator selects the Clear All command on the default menu, or any form executes CLEAR_FORM (both of which cause an implicit ROLLBACK), only changes that have been made since the called form was invoked (and the savepoint was issued) will be rolled back. Any changes that happened before that time, in any form, will remain in effect.

Managing Commit Processing When Using CALL_FORM

In a multiple–form application, both called forms and calling forms can issue DML commands to lock records, commit changes, and roll back posted changes. This section describes concepts and techniques for managing commit processing across called forms.

Post vs. Commit

If your application requires database transactions to span called forms, you need to understand the difference between posting and committing.

During a default commit operation, Oracle Forms issues the SQL statements necessary to update, delete, or insert records that have been marked in the form as changed, deleted, or inserted. Oracle Forms then issues the COMMIT statement to commit these transactions in the database.

Posting consists of writing updates, deletions, and insertions in the form to the database, but not committing these transactions to the database. You can explicitly cause Oracle Forms to post without committing by executing the POST built–in procedure.

When an application executes the POST procedure, Oracle Forms does all of the default validation and commit processing, but does not issue the COMMIT statement to finalize these transactions.

Because posted transactions have been written to the database, Oracle Forms does not have to maintain the status of the affected records across called forms. More importantly, because these transactions have not been committed, they can be rolled back programmatically. You can take advantage of this functionality to manage transactions across called forms.

What is Post–Only Mode?

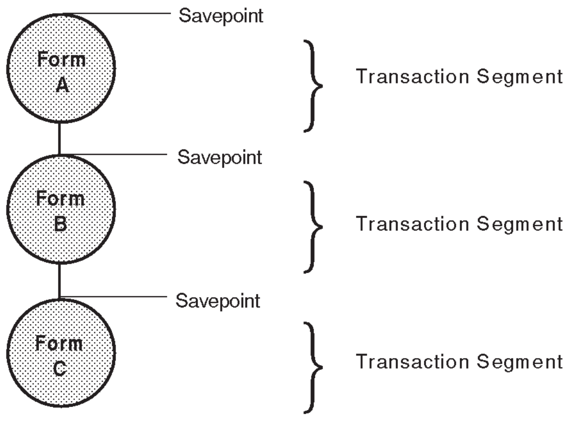
When a calling form has pending updates or deletes that have not been explicitly posted, Oracle Forms runs the called form in *post–only mode*. Further, any form called from a form running in post–only mode will also be in post–only mode.

In post–only mode, no commits or full rollbacks are allowed. Any attempt to commit returns the message ″`FRM-40403: A calling form has unposted changes.  Commit not allowed.`″ If an operator makes changes to records in the called form and then issues the default Exit command, Oracle Forms displays the alert ″`Do you want to Post the changes you have made?`″ . This functionality exists so that locks obtained by a calling form are maintained until control is returned to that form.

Savepoints and Rollbacks

When all of the forms in an application have the Savepoint Mode property On (the default), Oracle Forms issues a savepoint each time a form is loaded into memory at form startup, or via NEW_FORM or CALL_FORM (no savepoint is issued when a form is invoked with OPEN_FORM). When an application invokes multiple forms, these savepoints separate database transactions into segments that correspond to specific form modules.

Because Oracle Forms issues a savepoint for each called form, your application can post and roll back changes in individual forms without affecting changes in other forms that were active during the session.



Rolling Back Changes

The built–in procedures EXIT_FORM and NEW_FORM are similar in that both result in Oracle Forms leaving the current form; the difference between them is that while EXIT_FORM merely leaves the current form, NEW_FORM names another form to be loaded in its place. In both cases, Oracle Forms must either preserve or roll back changes that were made in the current form before exiting. In addition to EXIT_FORM and NEW_FORM, a CLEAR_FORM operation also either preserves or rolls back changes.

You can control whether changes in a form are preserved or rolled back when the following operations occur:

- a form executes the EXIT_FORM built–in procedure
- the operator selects the Exit item on the default Action menu (this is equivalent to calling EXIT_FORM with default parameters)
- a form executes the NEW_FORM built–in procedure
- a form executes the CLEAR_FORM built–in procedure
- the operator selects the Action, Clear All command from the default menu (this is equivalent to calling CLEAR_FORM with default parameters)

Whether changes are preserved or rolled back depends on the rollback mode specified for these operations.

When you call the EXIT_FORM, NEW_FORM, or CLEAR_FORM procedures, the rollback mode is determined by the constant you supply for the *rollback_mode* parameter: TO_SAVEPOINT, NO_ROLLBACK, or FULL_ROLLBACK.

Rollback Mode Parameters

The TO_SAVEPOINT, NO_ROLLBACK, and FULL_ROLLBACK parameters are predefined numeric constants, and should be entered without single quotes, as shown in the following examples:

```
Clear_Form(ASK_COMMIT,TO_SAVEPOINT);
New_Form('my_form',NO_ROLLBACK);
Exit_Form(ASK_COMMIT,FULL_ROLLBACK);
```

Rollback Parameter	*Description*
TO_SAVEPOINT	The default rollback mode. Oracle Forms rolls back uncommitted changes (including posted changes) to the last savepoint. If the form is a called form, any changes that were posted in the calling form are preserved. But changes in the current form or locks acquired are lost.
NO_ROLLBACK	Oracle Forms does not issue a rollback, and posted changes are preserved. (Unposted changes in the current form are lost.) When calling a new form or exiting a called form, any locks that were obtained by the current form remain in effect.
FULL_ROLLBACK	Oracle Forms rolls back all uncommitted changes pending in the current session (including posted changes). This includes changes made in the current form, posted changes made in forms that called the current form, and posted changes made in forms that were called by the current form.

Note that full rollbacks are not allowed when a form is in post–only mode, and calling a procedure with the FULL_ROLLBACK parameter returns an error such as ”`FRM–40739: Clear_Form with FULL_ROLLBACK not allowed in post–only form.`”

Default Rollback Mode

The default rollback mode for the CLEAR_FORM, NEW_FORM, and EXIT_FORM procedures is TO_SAVEPOINT. Hence, the statements

```
Clear_Form;
New_Form(’form_name’);
Exit_Form
```

are logically equivalent to

```
Clear_Form(ASK_COMMIT,TO_SAVEPOINT);
New_Form(’form_name’,TO_SAVEPOINT);
Exit_Form(ASK_COMMIT,TO_SAVEPOINT);
```

Note: Oracle Forms interprets a ROLLBACK statement in a PL/SQL block as a CLEAR_FORM built–in procedure with no parameters.

Modifying the CLEAR ALL and EXIT Commands

The default Oracle Forms menu provides a Clear All command on the Action menu that allows operators to roll back changes in a form. By default, the Clear All menu item is functionally equivalent to executing the CLEAR_FORM procedure with default parameters:

```
Clear_Form(ASK_COMMIT,TO_SAVEPOINT);
```

Similarly, the Exit menu item on the Action menu is functionally equivalent to calling the EXIT_FORM procedure with default parameters:

```
Exit_Form(ASK_COMMIT,TO_SAVEPOINT);
```

Because these menu items map directly to key commands, you can change their functionality by writing an appropriate Key trigger. For example, you might want to change the default behavior of the Clear All menu item to roll back all changes in the session, including changes posted in calling forms. To do so, you could write the following Key–CLRFRM (”clear–form”) trigger to cause a full rollback:

Key–CLRFRM Trigger:

```
Clear_Form(NO_COMMIT,FULL_ROLLBACK);
```

Similarly, you can change the rollback mode for the Exit menu item to NO_ROLLBACK, so that changes posted in the current form are not rolled back on exit. To do so, write the following Key–Exit trigger:

Key–EXIT Trigger:

```
Exit_Form(ASK_COMMIT,NO_ROLLBACK);
```

Using Posting and Rollback Mode to Manage Transactions

Although an application can include more than one form module, it can process only one database transaction at a time. A commit operation in a form commits updates, inserts, and deletes pending for the entire application session, rather than for any individual form in the application. This means that a commit issued by Form B can commit changes that were posted by Form A. Similarly, a rollback command can roll back changes that were posted in forms other than the current form.

When you build an application with multiple forms, you can use posting and rollback mode to control commit processing across called forms.

Example 1: Committing from a Called Form

Recall that Oracle Forms runs a called form in post–only mode when there are unposted changes in the calling form. Although post–only mode is useful in many applications, externalizing it to end–users can add complexity to your application's interface. For some applications, it may be better to prevent forms from running in post–only mode altogether. To do so, design the application so that changes are always explicitly posted before another form is called.

To post changes in a form, execute the POST built–in procedure just prior to calling a form with the CALL_FORM or NEW_FORM procedure. For example, you can include the POST procedure in the menu item command or When–Button–Pressed trigger that calls the next form:

```
Post;
IF (System.Form_Status <> 'QUERY') THEN
   Call_Form('form_B',NO_HIDE);
END IF;
```

When changes are posted in the calling form, the called form does not run in post–only mode, and operators can issue a commit from the called form. The following steps illustrate this sequence, and show how posting allows a called form to commit changes that were made in the calling form:

1. Update or delete records in Form A.
2. Post in Form A.
3. Call Form B.
4. Insert, update, or delete records in Form B.
5. Commit in Form B. (This commits the changes made in Form B and the posted changes in Form A.)
6. Exit Form B.

Checking for Changed Records

When there are no changes to be posted, executing the POST built–in procedure causes error "`FRM–40405: No changes to post.`" To avoid this error, check the system variable SYSTEM.FORM_STATUS to verify that at least one record in the form has been validated as CHANGED before attempting to post, as shown here:

```
ENTER;
IF System.Form_Status = 'CHANGED' THEN
   Post;
END IF;
Call_Form('form_B',NO_HIDE);
```

Example 2: Committing from a Calling Form

In this example, posting is used to allow a *calling form* to commit changes that were posted in a *called form*.

The called form is allowed to run in post–only mode, and any changes made in the called form are posted before returning to the calling form.

The following steps illustrate this sequence:

1. Update or delete records in Form A.
2. Call Form B. (Form B is in post–only mode because changes were made in Form A that were not explicitly posted.)
3. Insert, update, or delete records in Form B.
4. Post in Form B.
5. Exit Form B with NO_ROLLBACK parameter.
6. Commit in Form A. (This commits changes made in Form A and changes posted in Form B.)

When Form B exits and returns control to Form A (step 5), the rollback mode is set to NO_ROLLBACK to preserve changes that were posted in Form B.

Because the default rollback mode when exiting a form is TO_SAVEPOINT, you must explicitly override this functionality to avoid a rollback. For example, if the operator selects the Exit item on the default Action menu, changes made in the called form will be lost. To override the default Exit command, you might write the following trigger:

Key–EXIT Trigger:

```
IF System.Form_Status = 'CHANGED' THEN
    Post;
END IF;
Exit_Form(NO_COMMIT,NO_ROLLBACK);
```

This same technique can be used when you leave the current form by executing the NEW_FORM procedure rather than by calling EXIT_FORM.

```
IF System.Form_Status = 'CHANGED' THEN
    Post;
END IF;
New_Form(NO_COMMIT,NO_ROLLBACK);
```

In this case, changes posted by the called form can then be committed by the new form that takes its place.

Getting Information About the Call Form Stack

You can use the built–in function GET_APPLICATION_PROPERTY to get information about the call form stack in a multiple–form application, including the following:

- the name of the current form (CURRENT_FORM)
- the name of the form that called the current form. (CALLING_FORM).
- the username and password of the current operator (USERNAME, PASSWORD)

For example, to determine which form called the current form (with CALL_FORM), you could write the following code:

```
DECLARE
    parent_form CHAR(20) :=Get_Application_Property(CALLING_FORM);
BEGIN
    IF parent_form = ’form_A’ THEN
      Post;
      Exit_Form(NO_COMMIT,NO_ROLLBACK);
    ELSE
      Commit_Form;
      Exit_Form;
    END IF;
END;
```

If the current form is not a called form,

```
Get_Application_Property(CALLING_FORM);
```

returns NULL.

Suppressing Post and Commit Transaction Messages

When you build multiple–form applications, you might want to suppress messages regarding transaction posting and committing. To do so, set the SYSTEM.MESSAGE_LEVEL system variable to 5 just before a post or commit, then reset it to the desired value.

Passing Parameters to Forms

When you invoke a form with the procedures OPEN_FORM, CALL_FORM, or NEW_FORM, you can pass values for form parameters from the calling form to the called form.

To pass parameter values from one form to another, each parameter and its value must be in a *parameter list*. Parameter lists are internal, three–column data structures that contain the key (name), the type (Text_Parameter or Data_Parameter) and the value of each parameter on the list.

The parameters whose values are being passed must have been defined in the called form at design time. That is, the called form must be expecting a value for each of the parameters included in the parameter list it receives from the calling form.

You can define parameters in a form in the Object Navigator at design time and also programmatically at runtime. The properties of a parameter include Name, Datatype, Length, and Default Value. Parameter lists must be created programmatically with the built–in routines CREATE_PARAMETER_LIST and ADD_PARAMETER. See Chapter 16, ”Defining Form Parameters” for more information on form parameters and parameter lists.

Parameter values are not visible across multiple forms. Thus, even if there is a parameter named *p1* defined in both Form A and Form B, each form has a separate context, and setting the value of *p1* in Form B has no effect on the value of *p1* in Form A. For this reason, parameters are useful in multiple–form applications primarily as inputs to a form when it is first invoked.

If your application requires variables whose values are visible across called forms, you should use global variables. Global variables are visible across called forms, and remain active until they are explicitly deleted with the ERASE built–in procedure, or until the session ends. For information on global variables, see Chapter 18, ”Using PL / SQL in Oracle Forms,” in the Oracle Forms Developer’s Guide.

Passing a Parameter List

A parameter list is passed from one form to another as the last argument to the OPEN_FORM, CALL_FORM, or NEW_FORM built–in procedures:

```
DECLARE
    the_list PARAMLIST;
BEGIN
    the_list := Create_Parameter_List('form_a_params');
    Add_Parameter(the_list,'p1',TEXT_PARAMETER,'BICYCLE');
    Open_Form('form_B',ACTIVATE,NO_SESSION,the_list);
END;
```

In this example, a parameter list named *form_a_params* is created and its object ID is assigned to a variable named *the_list*. A text parameter named *p1* is then added to the list with ADD_PARAMETER, and its value is set to "BICYCLE." Finally, the parameter list is passed to Form B as the last argument to the OPEN_FORM procedure.

For this example to work, a form parameter named *p1* must have been declared in Form B at design time, and its datatype and length must be compatible with the value being passed.

The Default Parameter List

Each form includes a built–in parameter list named Default. The Default parameter list contains all of the form parameters that were defined in the form at design time. For example, if you define parameters *p1*, *p2*, and *p3* in Form A at design time, they are automatically included in the Default parameter list for Form A.

Like any other parameter list, the Default parameter list can be passed to a called form by including it in the argument list of the OPEN_FORM, CALL_FORM, or NEW_FORM built–in procedures.

```
DECLARE
    the_list PARAMLIST:= Get_Parameter_List('default');
BEGIN
    Open_Form('form_B',ACTIVATE, NO_SESSION,'default');
END;
```

Parameter Validation

When you pass the Default parameter list to a form, remember that each parameter in the list must have been defined in that form at design time. Oracle Forms validates the individual parameters against the parameters defined in that form as follows:

- Each parameter must have been defined in the called form at design time. If you pass a parameter list that includes an undefined parameter, the OPEN_FORM, CALL_FORM or NEW_FORM fails, and Oracle Forms issues error `FRM–47023: No such parameter named <parameter name> exists in <form module name>.`
- The datatype and length of the parameter defined in the called form must be compatible with the parameter value being passed. (Values in a parameter list are untyped strings, whereas form parameters are declared as type CHAR, NUMBER, or DATE.) If the parameter is incompatible with the value, the OPEN_FORM, CALL_FORM, or NEW_FORM call fails, and Oracle Forms issues error `FRM–47024: Parameter <parameter name> type does not match definition in <name of called form>.`

You can modify the Default parameter list by adding and deleting parameters with the built–in procedures ADD_PARAMETER and DELETE_PARAMETER.

To avoid a validation error, you can remove unwanted parameters from the list before passing it to a form. For example, if Form B requires only parameters *p1* and *p2*, you could remove *p3* from the list before passing the list to Form B.

```
DECLARE
    the_list PARAMLIST:= Get_Parameter_List('default');
BEGIN
    Delete_Parameter(the_list, 'p3');
    Open_Form('form_B',ACTIVATE, NO_SESSION,'default');
END;
```

Initial Values of Parameters in a Called Form

The following table shows how the initial values of parameters in the called form are determined by the parameter list being passed by the calling form with OPEN_FORM, CALL_FORM, or NEW_FORM. It assumes that there are corresponding parameters in Form A (the calling form) and Form B (the called form).

If Form A passes...	*The values of the corresponding parameters in Form B are set to...*
No parameter list	The default parameter values that were specified in Form B at design time. (If no default value was specified in the Parameter property sheet, parameter values default to NULL.)
Default parameter list	The current values of the corresponding parameters in Form A.
Programmatically created parameter list	The values specified for the parameters when they were put in the parameter list with ADD_PARAMETER.

Integrating Form and Menu Modules

Just as multiple form modules can be combined into a single application, so too can multiple menu modules be used as needed. You can design an application so that several form modules share the same menu module, or have each form in the application run its own menu. How menus are loaded in multiple–form applications depends on whether you are using OPEN_FORM, NEW_FORM, or CALL_FORM.

OPEN_FORM Menu Functionality

When you create multiple–form applications with OPEN_FORM, each form has its own menu. When the operator navigates between forms, the menu for the current form becomes the current menu for the application. Note that on MS Windows, the MDI display style dictates that the menu for the current form should be displayed on the MDI application window. This means that as operators navigate between forms, only the menu for the current form will be displayed.

NEW_FORM Menu Functionality

When you create multiple–form applications with the NEW_FORM built–in procedure, Oracle Forms always loads the new form's menu module, even if it is the same module used by the calling form.

The only exception to this rule is if the form that executes the NEW_FORM procedure is itself a called form that was invoked with the NO_REPLACE parameter. In this case, the new form inherits the NO_REPLACE restriction, and runs under the calling form's menu.

CALL_FORM Menu Functionality

When you create multiple–form applications with the CALL_FORM built–in procedure, the *switch_menu* parameter determines whether Oracle Forms loads a different menu or retains the current menu. Valid constants for the *switch_menu* parameter include DO_REPLACE and NO_REPLACE.

The NO_REPLACE parameter (the default) causes Oracle Forms to keep the current menu module. DO_REPLACE causes Oracle Forms to replace the current menu module with the menu that was attached to the called form at design time (either the default Oracle Forms menu or a custom menu module).

For example,

```
Call_Form('form_B',HIDE,DO_REPLACE);
```

runs Form B with the menu that was attached to Form B at design time, rather than keeping the current menu.

Creating a Master Menu

In multiple–form applications, it is common to create a front–end form that exists only to display the application's main menu. Items on the main menu can then invoke individual form modules that perform specific application functions. To create a master menu that invokes other forms, first create a form module with no blocks or items, then attach your master menu module to that form.

Invoking Forms from a Master Menu with OPEN_FORM When items on a master menu invoke forms with OPEN_FORM, each independent form has its own menu. You cannot specify that an independent form use the current menu.

Invoking Forms from a Master Menu with NEW_FORM When items on the master menu invoke forms with the NEW_FORM built–in procedure, the menu module attached to each form is loaded when that form is invoked. When all of the forms in an application share the same menu, reloading the same menu each time a new form is invoked (and potentially querying the database each time) may be unacceptable.

To avoid this problem, create a start–up form that exists only to call the front–end form (and its master menu) by executing CALL_FORM with the NO_REPLACE parameter.

With this design, Oracle Forms does not reload the menu each time a different form is invoked from the main menu. Because the front–end form is itself a called form, any form modules invoked with the NEW_FORM procedure automatically inherit the NO_REPLACE restriction, and run under the master menu.

Invoking Forms from a Master Menu with CALL_FORM When items on the master menu invoke forms with the CALL_FORM built–in procedure, you can specify whether called forms should inherit the master menu with the *switch_menu* parameter (DO_REPLACE or NO_REPLACE).

Using the REPLACE_MENU Built–in Procedure

You can use the REPLACE_MENU built–in procedure to dynamically replace the current form's menu with a different menu:

```
Replace_Menu(menu_module_name     CHAR,
           [,menu_type            NUMBER := PULL_DOWN
            starting_menu         NUMBER,
            group_name            CHAR
            use_file              BOOLEAN := NULL];
```

For example, to replace the current menu with a menu called *my_new_menu* you could write the following procedure call:

```
Replace_Menu('my_new_menu');
```

CHAPTER

6

Responding to Mouse Events

Oracle Forms allows you to initiate actions based on mouse events. This chapter describes the following topics:

About Mouse Events, Triggers, and System Variables

There are several mouse events that can occur at runtime. For example, when an operator clicks the mouse on an item, the item clicked can signal whether it received a single or a double–click.

You can execute a command or initiate an action whenever one of the following mouse events occurs:

- operator presses the mouse down
- operator presses the mouse down and releases the mouse button
- operator clicks the mouse
- operator double–clicks the mouse
- operator moves the mouse into a canvas or item
- operator moves the mouse out of a canvas or item
- operator moves the mouse

Mouse Event Triggers

Creating a mouse event trigger to respond to a mouse event allows you to initiate an action whenever the specified mouse event occurs.

Mouse Trigger	*Description*
When–Mouse–Down	Initiates an action when the operator presses down the mouse button within an item or canvas–view.
When–Mouse–Up	Initiates an action when the operator presses down and releases the mouse button within an item or canvas–view.
When–Mouse–Click	Initiates an action when the operator clicks the mouse within an item or canvas–view. Clicking the mouse consists of the mouse down and up events.
When–Mouse–DoubleClick	Initiates an action when the operator double–clicks the mouse within an item or canvas–view. Double–clicking the mouse consists of the mouse down, mouse up, mouse click, mouse down, and mouse up events.
When–Mouse–Enter	Initiates an action when the operator moves the mouse into an item or canvas–view.

Mouse Trigger	*Description*
When–Mouse–Leave	Initiates an action when the operator moves the mouse out of an item or canvas–view.
When–Mouse–Move	Initiates an action when the operator moves the mouse within an item or canvas–view.

Mouse Event System Variables

When any of the mouse events occur, Oracle Forms initializes the appropriate mouse event system variable (see below) and executes the corresponding mouse trigger.

Note: Mouse system variables are set immediately *before* a mouse trigger is fired. To guarantee that your mouse system variables are updated, you should only use the mouse system variables within mouse triggers.

Mouse Event System Variable	*Value*
SYSTEM.MOUSE_BUTTON_ PRESSED	The number of the button (1–2) that was pressed. **Note:** Mouse button support is limited to buttons 1 and 2 (left and middle) on a three button mouse. Mouse button 3, or the right mouse button, is reserved for future pop-up support.
SYSTEM.MOUSE_BUTTON_ SHIFT_STATE	The shift modifier pressed during the mouse click.
SYSTEM.MOUSE_ITEM	The name of the item the mouse is currently in.
SYSTEM.MOUSE_CANVAS	The name of the canvas which the mouse is currently in.
SYSTEM.MOUSE_X_POS	The current mouse X–coordinate on the canvas according to the coordinate system except when used within a When–Mouse–Enter trigger. When using a When–Mouse–Enter trigger, SYSTEM.MOUSE_X_POS represents the X–coordinate relative to the item entered, not the item position on the canvas.
SYSTEM.MOUSE_Y_POS	The current mouse Y–coordinate on the canvas according to the coordinate system except when used within a When–Mouse–Enter trigger. When using a When–Mouse–Enter trigger, SYSTEM.MOUSE_Y_POS represents the Y–coordinate relative to the item entered, not the item position on the canvas.

SYSTEM.MOUSE_RECORD	The record number of the record the mouse is in.
SYSTEM.MOUSE_RECORD_ OFFSET	The offset from the first visible record that the mouse is in.

Note: Oracle Forms does not perform mouse navigation internally to fire the mouse triggers. As a result, mouse move, enter, and leave can fire on the non–current item, or on an item that does not contain a record (for example, the fifth record in a multi–record block which is in Enter Query mode).

Performing Actions Based on the Mouse Button Pressed

You can initiate an action based on the button the operator presses, whether the button pressed is button 1 or 2, a single click or a double–click; a single click combined with a shift modifier, and so on.

Note: Mouse button support is limited to buttons 1 and 2 (left and middle) on a three button mouse. Mouse button 3, or the right mouse button, is reserved for future popup support.

Single or Double–click To perform an action when the operator clicks or double–clicks the mouse, use either a When–Mouse–Click or a When–Mouse–DoubleClick trigger.

Button Number Pressed To initiate an action based on the mouse button pressed, use a When–Mouse–Click trigger in conjunction with the SYSTEM.MOUSE_BUTTON_PRESSED system variable.

The following example demonstrates how to determine which mouse button is pressed when the operator clicks the mouse.

```
/*
**  Trigger:  When–Mouse–Click
*/
DECLARE
   the_button_pressed  VARCHAR(1);
BEGIN
   the_button_pressed := :System.Mouse_Button_Pressed;
   IF the_button_pressed = ’1’ THEN
     Show_Window(’options_window’);
   END IF;
END;
```

Click Combined with Modifier Key To perform an action based on the modifier key pressed when the operator clicks the mouse, use the When–Mouse–Click trigger in conjunction with the SYSTEM.BUTTON_SHIFT_STATE system variable.

Performing Actions Based on Mouse Location

You can initiate an action based on the mouse's current location, whether the mouse is in an item, record, coordinate area, or canvas–view.

Mouse in Item To determine which item the mouse is in, use a When–Mouse–Move trigger in conjunction with the SYSTEM.MOUSE_ITEM system variable.

Example:

```
/*
**  Trigger:  When–Mouse–Move
*/
DECLARE
   mouse_location   varchar(50);
BEGIN
   mouse_location := :System.Mouse_Item;
END;
```

Mouse Coordinates To initiate an action based on the current X and Y mouse coordinates, use a When–Mouse–Move trigger in conjunction with the SYSTEM.MOUSE_X_POS and SYSTEM.MOUSE_Y_POS system variables.

In the following example, the SYSTEM.MOUSE_X_POS and SYSTEM.MOUSE_Y_POS system variables are used within a When–Mouse–Click trigger to dynamically reposition items.

```
DECLARE
   item_to_move       VARCHAR(50);
   the_button_pressed VARCHAR(50);
   target_x_position  VARCHAR(3);
   target_y_position  VARCHAR(3);
BEGIN
/*
**  Get the name of the item that was clicked.
*/
   item_to_move := :System.Mouse_Item;
   the_button_pressed := :System.Mouse_Button_Pressed;
```

```
/*
**  If the mouse was clicked on an area of a canvas that is
**  not directly on top of another item, move the item to
**  the new mouse location.
*/
   IF item_to_move IS NOT NULL AND the_button_pressed =
   ’2’ THEN
     target_x_position := :System.Mouse_X_Pos);
     target_y_position := :System.Mouse_Y_Pos);
     Set_Item_Property(item_to_move,position,
        target_x_position,target_y_position);
     target_x_position := NULL;
     target_y_position := NULL;
     item_to_move := NULL;
   END IF;
END;
```

Mouse in Record To perform an action based on the record number the mouse is in, use When–Mouse–Move or a When–Mouse–Click trigger in conjunction with the SYSTEM.MOUSE_RECORD system variable.

Mouse in Canvas To perform an action based on the canvas the mouse is in, use a When–Mouse–Move or a When–Mouse–Click trigger in conjunction with the SYSTEM.MOUSE_CANVAS system variable.

In the following example, the SYSTEM.MOUSE_CANVAS system variable is used within a When–Mouse_Move trigger to determine which canvas the mouse is in.

```
DECLARE
   canvas_name  VARCHAR(50);
BEGIN
   canvas_name := :System.Mouse_Canvas;
END;
```

To determine which window the mouse is in, add the following line to the example above:

```
window_name := :Get_View_Property(canvas_name,window_name);
```

CHAPTER

7 Using Timers

This chapter explains how to create and manipulate timers. It includes the following sections:

Creating Timers

A timer is an “internal time clock” that you programmatically create to perform an action each time the timer expires.

Timer duration can be between 1 and 2,147,483,647 millisecond (1 second=1000 milliseconds). The maximum duration of a timer is approximately 24.85 days.

When you work with timers you perform these steps:

1. Using the CREATE_TIMER built–in subprogram, create the desired number of repeating or non–repeating timers.
2. Create a When–Timer–Expired trigger that performs the desired action whenever your timer expires.

You create a timer by using the CREATE_TIMER built–in subprogram.

```
CREATE_TIMER(timer_name, milliseconds, iterate);
```

where:

timer_name	Specifies the timer name of up to 30 alphanumeric characters. The name must begin with an alphabetic character. The datatype of the name is CHAR.
milliseconds	Specifies the duration of the timer in milliseconds. The range of values allowed for this parameter is 1 to 2147483648 milliseconds. Values > 2147483648 will be rounded down to 2147483648. Note that only positive numbers are allowed. The datatype of the parameter is NUMBER. See Restrictions below for more information.
iterate	Specifies whether the timer should repeat or not upon expiration. Takes the following constants as arguments: **REPEAT** Indicates that the timer should repeat upon expiration. Default. **NO_REPEAT** Indicates that the timer should not repeat upon expiration, but is to be used once only, until explicitly called again.

Note: When–Timer–Expired is a form–level trigger. It fires **any** time a timer expires. If your application contains several timers, your When–Timer–Expired trigger should contain code that will handle the different timers accordingly.

Example:

```
/*  Create a repeating timer that expires every hour */
DECLARE
     hour_timer TIMER;
     one_hour     NUMBER(7):=3600000;
BEGIN
     hour_timer:= CREATE_TIMER('alarm',one_hour,REPEAT);
END;
```

Timer Usage Rules

When using timers, consider these usage rules:

- A When–Timer–Expired trigger cannot fire during transactions, trigger processing, navigation, etc. Thus, a When–Timer–Expired trigger only fires while Oracle Forms is waiting to accept user input.

  As a result, a timer may not expire exactly on the millisecond, but it will fire after the specified number of milliseconds.
- By default, a timer repeats on expiration unless you specify NO_REPEAT.
- When a timer is created, Oracle Forms puts it on a queue.

  A When–Timer–Expired trigger will only fire once for each timer that is on the queue.
- A repeating timer will not repeat while it is on the queue. It will begin repeating once it has been serviced off of the queue by a When–Timer–Expired trigger. Thus, only one instance of a timer may be placed on the expired timer queue at a time.
- If the operator exits an application prior to timer expiration, any timer on the queue will not be executed by the When–Timer–Expired trigger.

Responding to Multiple Timers

When working with multiple timers, remember that the When–Timer–Expired is a form–level trigger. It fires **any** time a timer expires. If your application contains several timers, your When–Timer–Expired trigger should contain code that will handle the different timers accordingly.

Note: To retrieve the timer name of the most recently executed timer, initiate a call to GET_APPLICATION_PROPERTY from within a When–Timer–Expired trigger. Otherwise, the results of the built–in are undefined.

GET_APPLICATION_PROPERTY(timer_name) returns the name of the most recently expired timer. Oracle Forms returns NULL in response to this constant if there is no timer**.**

Example:

```
/*  Create a When–Timer–Expired trigger that can handle multiple
**  application timers.
*/
DECLARE
     expired_timer  CHAR(20);
BEGIN
     expired_timer:=GET_APPLICATION_PROPERTY(TIMER_NAME);
     IF expired_timer='T1'
        THEN /* handle timer T1 */;
     ELSIF expired_timer='T2'
        THEN /* handle timer T2 */;
     ELSE /* handle all other timers */;
     END IF;
END;
```

Modifying Timers Programmatically

You can use the following built–in subprograms when working with timers:

- CREATE_TIMER
- DELETE_TIMER
- FIND_TIMER
- GET_APPLICATION_PROPERTY
- SET_TIMER

Note: You cannot change a timer's name programmatically.

You can use the SET_TIMER built–in subprogram to modify timer intervals and repeat parameters.

For example:

```
/*
** Builtin:  FIND_TIMER
**
** Example:  If the timer exists, reset it. Otherwise create
**           it.
*/
PROCEDURE Reset_Timer_Interval( Timer_Name VARCHAR2,
Timer_Intv NUMBER ) IS
  tm_id       Timer;
  tm_interval NUMBER;
BEGIN
  /*
  ** User gives the interval in seconds, the timer routines
  ** expect milliseconds
  */
  tm_interval := 1000 * Timer_Intv;
  /* Lookup the timer by name */
  tm_id := Find_Timer(Timer_Name);
  /* If timer does not exist, create it */
  IF Id_Null(tm_id) THEN
    tm_id := Create_Timer(Timer_Name,tm_interval,NO_REPEAT);
  /*
  ** Otherwise, just restart the timer with the new interval
  */
  ELSE
    Set_Timer(tm_id,tm_interval,NO_REPEAT);
  END IF;
END;
```

Deleting a Timer

You can delete repeating and non–repeating timers by using the DELETE_TIMER built–in subprogram.

You can use the FIND_TIMER and DELETE_TIMER built–in subprograms to find and delete any timer that is created during the current session.

Example:

```
/*
** Builtin:  DELETE_TIMER
** Example:  Remove a timer after first checking to see if it
**           exists
*/
PROCEDURE Cancel_Timer (tm_name VARCHAR2) IS
     tm_id  TIMER;
BEGIN
  tm_id:=Find_Timer(tm_name);
  IF NOT Id_Null(tm_id) THEN
    Delete_Timer(tm_id);
  ELSE
    Message(’Timer ’||’ has already been cancelled.’);
  END IF;
END;
```

CHAPTER

8

Integrating with Other Oracle Tools

You can integrate other Oracle tools with Oracle Forms to build complete applications. This chapter explains how to integrate with Oracle Reports, Oracle Graphics, and Oracle Book, and includes the following topics:

About Integration with Other Oracle Tools

Oracle Forms supports integration with other Oracle tools, including Oracle Reports, Oracle Graphics, and Oracle Book. Integrated applications improve end–user productivity by providing seamless access to multiple special–purpose tools. For example, an inventory control system can include data entry capability, integrated reporting functions, and full–color charts and graphs, all within a single application.

The primary way to integrate Oracle Forms with other tools is to write an appropriate form trigger to invoke another tool in response to some event. When Oracle Forms invokes another product, it can

- Pass command line parameters to the product.
- Pass text parameters to the product.
- Pass record groups to the product that can be used to satisfy named queries defined in the called product, thus eliminating or reducing the need for the called product to query records from the database itself.

When applicable, you can specify that the called product run in batch or runtime mode, and you can pass command line parameters to specify runtime options for the called product. Called products can either run in the background or be displayed alongside the form. For instance, when a form invokes Oracle Reports, the report can be displayed on the screen where it can be viewed by the operator, or it can be sent directly to a printer.

Creating chart items in forms makes possible an additional level of integration between Oracle Forms and Oracle Graphics. A chart item is a special type of form item that is associated with a specific Oracle Graphics display (chart, graph, or other graphical display). Chart items can be updated dynamically when required by the application. When you use chart items, Oracle Forms can pass data to Oracle Graphics for use in constructing the display, or Oracle Graphics itself can query the data required. Once Oracle Graphics creates the display, Oracle Graphics passes the display to Oracle Forms to be displayed in the form interface.

Another option for integration between Oracle Forms and Oracle Graphics is OLE2 container support. Oracle Graphics is an OLE2 server application that can be linked or embedded in a custom OLE item in a form. Note, however, that OLE container functionality is supported only on MS Windows, whereas chart item integration, which relies on the Oracle Tools integration system, is fully portable, and also allows for bi–directional data passing.

Calling Other Products from Oracle Forms

You can invoke other products from Oracle Forms with the RUN_PRODUCT built–in procedure. The syntax for RUN_PRODUCT is shown here:

```
RUN_PRODUCT(product, document, commmode, execmode, location,
            list, display);
```

For example, to invoke Oracle Reports, you could make the following call:

```
Run_Product(REPORTS,'stats',ASYNCHRONOUS,BATCH,FILESYSTEM);
```

By default, when you invoke Oracle Reports or Oracle Graphics with RUN_PRODUCT, the called product logs on to ORACLE using the current form operator's USERID.

Oracle Forms uses the parameters you pass to RUN_PRODUCT to construct a valid command line invocation of the called product. RUN_PRODUCT takes the following parameters:

Product A numeric constant that specifies the Oracle tool to be invoked: FORMS, REPORTS, GRAPHICS, or BOOK.

Document Specifies the document or module to be opened by the called product.

Commmode Specifies the communication mode to be used when running the called product. Valid numeric constants for this parameter are SYNCHRONOUS and ASYNCHRONOUS.

- SYNCHRONOUS specifies that control returns to Oracle Forms only after the called product has been exited. The operator cannot work in the form while the called product is running. Synchronous is required when passing a record group to a called product as a DATA_PARAMETER; for example, when invoking Oracle Graphics to return an Oracle Graphics display that will appear in a form chart item.
- ASYNCHRONOUS specifies that control returns to the calling application immediately, even if the called application has not completed its display. Do not use ASYNCHRONOUS when passing a record group to a called product as a DATA_PARAMETER; for example, when invoking Oracle Graphics to return an Oracle Graphics display that will appear in a form chart item.

Execmode Specifies the execution mode to be used when running the called product, either BATCH or RUNTIME. When you run Oracle Reports and Oracle Graphics, execmode can be either BATCH or

RUNTIME. When you run Oracle Forms, always set execmode to RUNTIME.

Location Specifies the location of the document or module you want the called product to execute, either the file system or the database.

List Specifies the name or ID of a parameter list to be passed to the called product.

Display Specifies the name of the Oracle Forms chart item that will contain the display generated by Oracle Graphics.

For more information, refer to the description of RUN_PRODUCT in the *Oracle Forms Reference Manual, Vol. 1.*

Suppressing the Logon in Oracle Graphics

By default, when Oracle Forms invokes Oracle Graphics, Oracle Graphics logs on to ORACLE with the same USERID as the current form operator. In some cases, it may not be necessary for Oracle Graphics to log on to ORACLE, for example, when a form queries data and then passes it to Oracle Graphics by way of a DATA_PARAMETER. When a form provides the data for all of the queries in a display, there is no need for Oracle Graphics to query the database, and the needless logon should be avoided.

You can prevent Oracle Graphics from logging on by passing a text parameter with *key* set to LOGON and *value* set to NO. To do so, use the ADD_PARAMETER built–in to add the LOGON parameter to the parameter list being passed to Oracle Graphics, as shown here:

```
DECLARE
    list_id    ParamList;
BEGIN
    list_id := Create_Parameter_List('input_params');
    Add_Parameter(list_id,'PRINT',TEXT_PARAMETER,'YES');
    Add_Parameter(list_id,'COPIES',TEXT_PARAMETER,'1');

    –– Now add the LOGON parameter to prevent logon
    Add_Parameter(list_id,'LOGON',TEXT_PARAMETER, 'NO');
    Run_Product(GRAPHICS,'daily_sums',SYNCHRONOUS, BATCH,
                       FILESYSTEM, list_id);
END;
```

Note: A separate logon is always required if the Oracle Graphics display executes Data Manipulation Language (DML) commands against the database.

Invoking Oracle Book from Oracle Forms

You can integrate Oracle Forms with Oracle Book to provide online documentation and context–sensitive help in your form applications. To invoke Oracle Book, use the RUN_PRODUCT or HOST built–ins to execute a valid Oracle Book logon.

Oracle Book has hypertext capabilities that allow operators to quickly navigate through documents by clicking on in–text *links* to jump to corresponding *targets*. You can take advantage of these targets in Oracle Book documents to implement context–sensitive help systems for your form applications. For example, a form can invoke an Oracle Book document and navigate to a specific topic, based on current application context.

There is an Oracle Book command line parameter called TARGET that you can set to specify the name of a hypertext target when you invoke Oracle Book. When you invoke Oracle Book with HOST, you can pass a value for the TARGET parameter as part of the command line argument. When you invoke Oracle Book with RUN_PRODUCT, you can pass the name of the target as a parameter in a parameter list.

Passing Parameters to Called Products

You can pass parameters to products you invoke from Oracle Forms. A parameter you pass to a called product can be either a *text parameter* or a *data parameter*.

The value of a text parameter is a CHAR string that can represent any of the following:

- a command line parameter to be used by the called product at startup; for example, a value for the RUNREP command line parameter DESTYPE, which indicates the display destination for the report
- a user–defined parameter defined in Oracle Reports or Oracle Graphics; for example, a value for a department number parameter required by a report
- a bind or lexical reference defined in Oracle Reports or Oracle Graphics; for example, a value for the bind reference :SALARY, or a value for the lexical reference &MINTOTAL

The value of a data parameter is always the name of a record group defined in the current form. When Oracle Forms passes a data parameter to Oracle Reports or Oracle Graphics, the data in the specified record group can substitute for a query that Oracle Reports or Oracle Graphics would ordinarily execute to run the report or display.

Note: Passing data parameters is not supported when invoking Oracle Forms from Oracle Forms.

Creating Parameter Lists

To pass a parameter to a called product you must first programmatically create a *parameter list.* A parameter list is simply a list of parameter names (called *keys*) and their values. To pass one or more parameters to a called product, your form must perform the following steps in a trigger or user–named subprogram:

- execute the CREATE_PARAMETER_LIST built–in function to programmatically create a parameter list
- execute the ADD_PARAMETER built–in procedure to add one or more parameters to the parameter list, specifying the key, type, and value for each parameter being added
- execute the RUN_PRODUCT built–in procedure and include the ID or name of the parameter list to be passed to the called product

For more information on parameters and parameter lists, see Chapter 16, Defining Form Parameters."

The following example illustrates these steps by creating a parameter list to be passed to Oracle Reports when running a report called *Daily_Sums*.

```
/* Declare an appropriately typed variable to store
** the parameter list ID
*/
DECLARE
    list_id   ParamList;
BEGIN

/* Create a parameter list named ”input_params” */
list_id := Create_Parameter_List(’input_params’);

/* Add two parameters to the list that pass values
** for RUNREP command line parameters; for each parameter
** specify its key, type (text or data), and value
*/
Add_Parameter(list_id,’DESTYPE’,TEXT_PARAMETER,’PRINTER’);
Add_Parameter(list_id,’DESNAME’,TEXT_PARAMETER,’PS_3’);
```

```
/* Now run the report, referencing the parameter list ID
** in the last argument to the RUN_PRODUCT procedure */
Run_Product(REPORTS,'daily_sums',SYNCHRONOUS,RUNTIME,
            FILESYSTEM,list_id);
END;
```

Parameter Attributes When you add call ADD_PARAMETER to add a parameter to a parameter list, you specify three attributes:

key	The name of the parameter.
paramtype	The parameter type, either DATA_PARAMETER or TEXT_PARAMETER.
value	The parameter value. For a TEXT_PARAMETER, the value is any CHAR value. For a DATA_PARAMETER, the value is always the CHAR name of a record group defined in the current form.

Parameter Associations When you execute the RUN_PRODUCT procedure and pass a parameter list to another product, Oracle Forms uses the parameters in the list to construct a valid command line invocation for the called product. Any text parameters in the list are interpreted as either pre–defined command line parameters, or as user–defined bind or lexical parameters. Similarly, any data parameters are understood to map directly to named queries in Oracle Reports or Oracle Graphics.

When you pass a parameter to a called product, the parameter key (that is, the name of the parameter) must be the same as the name of the corresponding parameter or query in the called product:

- When passing a value for a command line option, the text parameter key must be the same as the command line keyword ('DESTYPE' or 'COPIES').
- When passing a value for a bind or lexical reference, the text parameter key must be the same as the name of the bind or lexical reference.
- When passing data, the data parameter key must have the same name as the query defined in the called product.

There can be any number of text and / or data parameters in a parameter list. For example, a form can pass data parameters for any number of named queries defined in a single report or display.

Note: When you use RUN_PRODUCT to invoke Oracle Reports, DATA_PARAMETERs can be passed only to master queries. Passing DATA_PARAMETERs to child queries is not supported.

Note: Passing DATA_PARAMETERs is not supported when invoking Oracle Forms from Oracle Forms with RUN_PRODUCT.

Using Chart Items to Embed Oracle Graphics Displays

You can create *chart items* in a form that contain displays generated by Oracle Graphics. The data used by Oracle Graphics to construct the display can be derived from a query in the form and then passed to Oracle Graphics, or Oracle Graphics itself can query the required data when it creates the display.

Integration between Oracle Forms and Oracle Graphics is based on the Oracle Tools integration system, making possible full automation of any chart item container. Applications built with this system are fully portable across platforms. When an Oracle Graphics display is embedded in a form, the form has complete control of the display, including the ability to pass mouse events to the display, as well as bi–directional parameter and data passing between Oracle Forms and Oracle Graphics.

These are the main steps to embed an Oracle Graphics display in a form:

1. In the Oracle Graphics Designer, create the display that you want to embed in the form.
2. In the form, create a chart item on the desired canvas.
3. Write the trigger that will start the Oracle Graphics batch executable and open the display. Opening the display starts Oracle Graphics and populates the chart item in the form.
4. (Optional)Automate the display as desired:
   - Pass text parameters or data parameters from the form to Oracle Graphics that Oracle Graphics can use to update the display.
   - Pass text parameters or data parameters from Oracle Graphics to Oracle Forms.
   - Write mouse event triggers to pass mouse events from Oracle Forms to Oracle Graphics. Oracle Graphics can then update the display in the chart item, just as it would in Oracle Graphics runtime.
   - Control the display from Oracle Forms by calling Oracle Graphics procedures from within form triggers.

You can use the services in the special OG package to create dynamic Oracle Graphics displays. The OG package is defined in the OG.PLL library, which is delivered as part of your Oracle Forms installation. To call subprograms in the OG package, you need to attach the OG.PLL library to your form. OG contains the following subprograms:

- OG.OPEN
- OG.CLOSE
- OG.GETCHARPARAM
- OG.GETNUMPARAM
- OG.INTERPRET
- OG.MOUSEDOWN
- OG.MOUSEUP
- OG.REFRESH

Creating a Chart Item

Chart items are special items that display a chart or other layout display generated by Oracle Graphics. When you create a chart item, you must also create a display in Oracle Graphics, and then write the triggers to initialize and populate the chart item at runtime.

You can create a chart item in the Layout Editor or in the Object Navigator. Because chart items do not store database values, they are always control items. Many of the properties that apply to other item types do not apply to chart items.

You can populate a chart item with an Oracle Graphics display by calling either RUN_PRODUCT or the OG.OPEN procedure available in the OG.PLL library. Each of these procedures takes an argument that specifies the name of the chart item that Oracle Graphics should populate with the indicated display.

```
Run_Product(GRAPHICS,'og_display',SYNCHRONOUS,BATCH,FILESYSTEM,
            NULL,'emp.my_chart');
```

When you create a chart item, consider the size of the display in Oracle Graphics and the size of the chart item in the form. When you display a chart item with RUN_PRODUCT, the display is scaled to fit the dimensions of the chart item. If the chart item is smaller than the actual display size, some distortion may occur. When you populate a chart item with OG.OPEN, you can specify whether the display should be scaled to fit, or should retain its actual dimensions and be clipped if necessary.

Fonts, colors, and patterns used to create the display in Oracle Graphics are matched as closely as possible to the attributes available on the runtime system.

You can create chart items in both single– and multi–record blocks. When you use a chart in a multi–record block, you must execute a separate RUN_PRODUCT, OG.OPEN, or OG.REFRESH call for each record, that is, for each instance of the chart item.

For example, you might create a multi–record base table block with a chart item that displays a graph based on the data in each record. You can build such a form using a block–level Post–Query trigger that executes a RUN_PRODUCT call for each record retrieved by the query. (Post–Query fires once for each record retrieved by a query.)

Passing Parameters and Data to Oracle Graphics

A form that contains an embedded chart item can pass parameters to Oracle Graphics for use in constructing or updating the display associated with the chart item. For example, when an operator enters a department number of 10 in a DEPT.DEPTNO field, the form can then pass that value to Oracle Graphics to use in constructing a bar chart showing expenses incurred by Department 10 in the last quarter.

You can also pass data parameters to Oracle Graphics. A data parameter is a pointer to a record group defined in the current form. Passing data parameters is appropriate when you want Oracle Graphics to use the results of a query executed in Oracle Forms, without re–executing the same query.

You can pass a parameter list that includes text and / or data parameters from Oracle Forms to Oracle Graphics whenever your application executes any of the following procedures:

- RUN_PRODUCT
- OG.OPEN
- OG.INTERPRET
- OG.MOUSEDOWN
- OG.MOUSEUP
- OG.REFRESH

Each of these procedures takes a parameter list ID as its final argument. Thus, any time you call these procedures in a form you have the option to send the Oracle Graphics display text or data parameters.

The following example shows two ways to pass parameters when populating a chart item with an Oracle Graphics display. This example demonstrates passing parameters with RUN_PRODUCT.

First, a display called *sal_chart* is created in Oracle Graphics that compares the salaries paid to managers and line employees in a given department. A query was defined in Oracle Graphics that accepts a parameter *dept_num* that defines the department number to be used in the WHERE clause of the query.

In the form, the designer created a single–record block based on the DEPT table, then added a chart item to the block called *chart_item*.

A When–New–Form–Instance trigger was defined that executes the following call at form startup:

```
Og.Open('sal_chart','dept.chart_item');
```

The OPEN procedure starts Oracle Graphics in batch mode and associates the display *sal_chart* with the chart item *dept.chart_item*. When the form operator queries a department record into the form, the chart item displays the pie chart showing the salary breakdown for employees in that department.

There are two ways to implement this functionality:

- Pass a text parameter.

  The form passes Oracle Graphics a parameter with a value for the *dept_num* parameter that was defined in the display; Oracle Graphics then uses that value to execute the query, builds the display, and passes it to the form chart item.

- Pass a data parameter.

  The form programmatically creates and populates a query record group, then passes the resulting record set to Oracle Graphics as a data parameter. The data queried by the form satisfies the named query defined in the sal_chart display. Oracle Graphics uses the data to build the display (without querying the database), then populates the form chart item.

Passing a Text Parameter The following sample code shows the text of a block–level Post–Query trigger that might be used to pass a department number to Oracle Graphics for use in constructing the *sal_chart* display.

```
PROCEDURE pass_param IS
    pl_id  ParamList;
BEGIN

/* Create a parameter list for data passing */
pl_id := Create_Parameter_List('my_param_list');

/* Add a text parameter to the list to supply a value for the
** 'dept_num' parameter that Oracle Graphics is expecting
*/
Add_Parameter(pl, 'dept_num', TEXT_PARAMETER,
           TO_CHAR(:dept.deptno));

/* Call Oracle Graphics to populate the chart item */
Og.Refresh('sal_chart','dept.chart_item',pl);

/* Get rid of the parameter list */
Destroy_Parameter_List(pl);

END;
```

Passing a Data Parameter The following sample code shows the text of a block–level Post–Query trigger that might be used to pass data records to Oracle Graphics. In this example, Oracle Forms issues a query and stores the resulting data in a record group. A data parameter is then passed to Oracle Forms that references the record group, and Oracle Graphics uses the data to construct the *sal_chart* display.

```
PROCEDURE pass_data IS
val   CHAR(20) := 'chart_data'; –– Name of Record Group in form
pl    ParamList;
rg    RecordGroup;
qry   VARCHAR(2000);
stat  NUMBER;
BEGIN

/* Prepare a query in a string */
qry := 'SELECT empno, sal FROM emp WHERE deptno='||
To_Char(:dept.deptno);

/* Try to get the ID of the 'chart_data' record group */
rg := Find_Group('chart_data');

/* If it doesn't exist, create the group based on the
** query in the 'qry' string
```

```
*/
IF Id_Null(rg) THEN
rg := Create_Group_From_Query(’chart_data’,qry);
END IF;

/* Make sure the ’chart_data’ record group is empty */
Delete_Group_Row(rg, ALL_ROWS);

/* Populate the ’chart_data’ record group with the query
** in the ’qry’ string
*/
stat := Populate_Group_With_Query(rg,qry);

/* Create a parameter list for data passing */
pl := Create_Parameter_List(’foo’);

/* Add a data parameter to the parameter list to
** specify the relationship between the named query
** ’query0’ in the Oracle Graphics display and the named
** record group in the form, ’chart_data’.
*/
Add_Parameter(pl,’query0’, DATA_PARAMETER,val);

/* Invoke Oracle Graphics to create the chart  */
Og.Refresh(’sal_chart’,’dept.chart_item’,pl);

/*
** Get rid of the parameter list
*/
Destroy_Parameter_List(pl);
END;
```

Creating a Chart Item that Responds to Mouse Events

You can automate a chart item by allowing operators to update and manipulate the embedded Oracle Graphics display by clicking on it with the mouse. For example, operators could be allowed to click on different parts of a graphical factory floor layout to see status information about a specific process. Or, they might drill down on a map to locate the region they wanted to see, with each mouse click updating the display to show a more granular view.

To create a chart item that operators can manipulate with the mouse, you must first create a display in Oracle Graphics that responds to the appropriate mouse events, just as you would if you were constructing it to run stand–alone in Oracle Graphics runtime.

Once you build the display, you can create the chart item in Oracle Forms, and then do the following:

- Write a trigger that calls the OG.OPEN packaged procedure to open the display and prepare it to receive mouse events. (When you use OG.OPEN, you do not need to call RUN_PRODUCT to initialize Oracle Graphics.)

  ```
  OG.Open('map','blk3.chart_item');
  ```

  If you want the display to be visible at form startup, you would typically call OG.OPEN in a When–New–Form–Instance or Pre–Form trigger. The display would then be able to receive mouse events immediately. After that, whenever you wanted Oracle Graphics to update the chart item, you could call OG.REFRESH, passing new parameters or data as needed.

  If you wanted to defer populating the chart item until some event occurred, such as the operator executing a query in the form, you might wait to call OG.OPEN until the appropriate event trigger fired, such as Post–Query or When–Button–Pressed.

- Write the appropriate mouse event trigger to respond to mouse events in the chart item.

  Oracle Forms includes a set of mouse event triggers that fire in response to mouse events, including mouse events that happen in a chart item. To notify Oracle Graphics that a mouse event has occurred in a chart item, you need to write an Oracle Forms mouse trigger that calls either the OG.MOUSEDOWN or OG.MOUSEUP packaged procedures. These procedures notify Oracle Graphics that a mouse event occurred, and pass the X,Y coordinates of the click on the display.

The OG Package

The OG package provides a set of PL / SQL subprograms that you can use when you are embedding Oracle Graphics displays in Oracle Forms chart items. The services in the OG package are based on the portable Oracle Tools integration mechanism, and allow you to create dynamic chart displays in a form. The OG package is defined in the OG.PLL library, which is delivered as part of your Oracle Forms installation. To call subprograms defined in the OG package, you need to attach the OG.PLL library to your form. For information on attaching libraries, refer to the *Oracle Forms Developer's Guide*, Chapter 20, "Working with Libraries."

OG.CLOSE

Syntax: `OG.CLOSE(`*`display,item`*`);`

Built–in Type: procedure

Description: Closes the indicated Oracle Graphics display.

Parameters:

display	The CHAR name of the display.
item	The CHAR name of the chart item with which the display is associated.

Example:

```
OG.Close('totals.ogd','blk3.chart_item');
```

OG.GETCHARPARAM

Syntax: `OG.GETCHARPARAM(`*`display,item,param`*`);`

Built–in Type: Function

Returns: VARCHAR2

Description: Returns the current value for the indicated Oracle Graphics CHAR parameter.

Parameters:

display	The CHAR name of the display.
item	The CHAR name of the chart item with which the display is associated.
param	The name of the parameter whose value you want to examine.

Example:

```
:dept.dept_name := OG.GetCharParam('my_disp.ogd',
               'blk2.chart_item,'deptname');
```

OG.GETNUMPARAM

Syntax: `OG.GETNUMPARAM(`*`display,item,param`*`);`

Built–in Type: Function

Returns: NUMBER

Description: Returns the current value for the indicated Oracle Graphics NUMBER parameter.

Parameters:

display	The CHAR name of the display.
item	The CHAR name of the chart item with which the display is associated.
param	The name of the parameter whose value you want to examine.

Example:

```
:dept.deptno := OG.GetCharParam('my_disp.ogd',
           'blk2.chart_item,'deptno');
```

OG.INTERPRET

Syntax:

```
OG.INTERPRET(display,item,pls_string);
OG.INTERPRET(display,item,pls_string,refresh);
OG.INTERPRET(display,item,pls_string,refresh,plist);
```

Description: Instructs Oracle Graphics to execute the indicated PL/SQL statement for the indicated display. The PL/SQL statement can include calls to Oracle Graphics built–in and user–named subprograms, as well as anonymous blocks of PL/SQL code.

Built–in Type: procedure

Parameters:

display	The CHAR name of the display.
item	The CHAR name of the chart item with which the display is associated.
pls_string	The PL/SQL statements to execute (type CHAR). Include the trailing semicolon as required. If you are issuing more than one statement at a time, include the appropriate DECLARE, BEGIN, and END keywords.
refresh	Optional BOOLEAN parameter that specifies whether the chart item display should be updated

after the PL / SQL statement executes in Oracle Graphics. Set to TRUE or FALSE. (Default=TRUE.)

plist	Optional; specifies the ID of a parameter list to be passed to Oracle Graphics. Use a variable of type PARAMLIST. (Default=TOOLS.null_parameter_ list.)

Example:

```
OG.Interpret('shop.ogd','control.chart2','do_update;',
               plist => p1);
```

OG.MOUSEDOWN

Syntax:

```
OG.MOUSEDOWN(display,item);
OG.MOUSEDOWN(display,item,x,y);
OG.MOUSEDOWN(display,item,x,y,refresh);
OG.MOUSEDOWN(display,item,x,y,refresh,clickcount);
OG.MOUSEDOWN(display,item,x,y,refresh,clickcount,button);
OG.MOUSEDOWN(display,item,x,y,refresh,clickcount,button,
             constrained);
OG.MOUSEDOWN(display,item,x,y,refresh,clickcount,button,
             constrained,plist);
```

Built–in Type: procedure

Description: Passes a mouse down event from Oracle Forms to the Oracle Graphics display associated with the indicated chart item. MOUSEDOWN is typically called from a When–Mouse–Down or When–Mouse–Click trigger attached to the chart item in Oracle Forms. When a mouse down event is passed from the form to the display, Oracle Graphics responds just as it would if the mouse down had occurred in Oracle Graphics runtime.

By default, OG.MOUSEDOWN passes Oracle Graphics the X,Y display coordinates of the mouse down event. Specifying these parameters explicitly is necessary only if you want to override the actual coordinates.

Parameters:

display	The CHAR name of the display.
item	The CHAR name of the chart item with which the display is associated.
x	Optional; specifies the X NUMBER coordinate at which Oracle Graphics should interpret the mouse click to have occurred. (Default=The actual X coordinate of the mouse down event.)

y	Optional; specifies the Y NUMBER coordinate at which Oracle Graphics should interpret the mouse click to have occurred. (Default=The actual Y coordinate of the mouse down event.)
refresh	Optional; specifies whether Oracle Graphics should update the chart item display. Set to TRUE or FALSE. (Default=TRUE)
clickcount	Optional; specifies a NUMBER counter for mouse down events that occur as part of a sequence of mouse events. For example, a double–click event includes a mouse down (count 1), a mouse up, a second mouse down (count 2), and a second mouse up. (Default=The actual clickcount.)
button	Optional; specifies a NUMBER corresponding to the mouse button that Oracle Graphics should interpret as having been pressed for the mouse event. (Default=The actual button that was pressed.)
constrained	Optional; specifies whether the mouse click should be interpreted as having occurred while the SHIFT key was pressed. Set to TRUE or FALSE. (Default=FALSE)
plist	Optional; specifies the ID of a parameter list to be passed to Oracle Graphics. Use a variable of type PARAMLIST. (Default=TOOLS.null_parameter_ list.)

OG.MOUSEUP

Syntax:

```
OG.MOUSEUP(display,item);
OG.MOUSEUP(display,item,x,y);
OG.MOUSEUP(display,item,x,y,refresh);
OG.MOUSEUP(display,item,x,y,refresh,button);
OG.MOUSEUP(display,item,x,y,refresh,button,constrained);
OG.MOUSEUP(display,item,x,y,refresh,button,constrained,plist);
```

Built–in Type: procedure

Description: Passes a mouse up event from Oracle Forms to the Oracle Graphics display associated with the indicated chart item. MOUSEUP is typically called from a When–Mouse–Up or When–Mouse–Click trigger attached to the chart item in Oracle Forms. When a mouse up event is passed from the form to the display, Oracle Graphics responds just as it would if the mouse up had occurred in Oracle Graphics runtime.

By default, OG.MOUSEUP passes Oracle Graphics the X,Y display coordinates of the mouse up event. Specifying these parameters explicitly is necessary only if you want to override the actual coordinates.

Parameters:		
	display	The CHAR name of the display.
	item	The CHAR name of the chart item with which the display is associated.
	x	Optional; specifies the X NUMBER coordinate at which Oracle Graphics should interpret the mouse click to have occurred. (Default=The actual X coordinate of the mouse up event.)
	y	Optional; specifies the Y NUMBER coordinate at which Oracle Graphics should interpret the mouse click to have occurred. (Default=The actual Y coordinate of the mouse up event.)
	refresh	Optional; specifies whether Oracle Graphics should update the chart item display. Set to TRUE or FALSE. (Default=TRUE)
	button	Optional; specifies a NUMBER corresponding to the mouse button that Oracle Graphics should interpret as having been pressed for the mouse event. (Default=The actual button that was pressed.)
	constrained	Optional; specifies whether the mouse click should be interpreted as having occurred while the SHIFT key was pressed. Set to TRUE or FALSE. (Default=FALSE)
	plist	Optional; specifies the ID of a parameter list to be passed to Oracle Graphics. Use a variable of type PARAMLIST. (Default=TOOLS.null_parameter_list.)

OG.OPEN

Syntax:

```
OG.OPEN(display,item);
OG.OPEN(display,item,clip);
OG.OPEN(display,item,clip,refresh);
OG.OPEN(display,item,clip,refresh,plist);
```

Built–in Type: procedure

Description: Activates the indicated Oracle Graphics display associated with the indicated chart item. The OPEN procedure includes a call to the built–in RUN_PRODUCT, but also allows you to specify additional parameters to control how Oracle Graphics should be activated. OPEN is typically called in a When–New–Form–Instance trigger to initialize a graphics display at form startup.

Parameters:

display	The CHAR name of the display.
item	The CHAR name of the chart item with which the display is associated.
clip	Optional BOOLEAN parameter that specifies whether Oracle Graphics should scale the chart display to fit the dimensions of the chart item, or use the default display size, cropping the display as needed to fit the chart item. Set to TRUE or FALSE. (Default=TRUE)
refresh	Optional BOOLEAN parameter that specifies whether the chart item display should be updated.
plist	Optional; specifies the ID of a parameter list to be passed to Oracle Graphics. Use a variable of type PARAMLIST. (Default=TOOLS.null_parameter_ list.)

Example:

```
OG.Open('shop.ogd','control.chart2');
```

OG.REFRESH

Syntax:

```
OG.REFRESH(display,item);
OG.REFRESH(display,item,plist);
```

Built–in Type: procedure

Description: Causes Oracle Graphics to update the bitmap display for the indicated chart item. Use OG.REFRESH to pass new parameters or data to Oracle Graphics to use to update a display for a chart item. To call OG.REFRESH successfully, Oracle Graphics must already have been initialized through a call to OG.OPEN.

Parameters:

display	The CHAR name of the display.
item	The CHAR name of the chart item with which the display is associated.
plist	Optional; specifies the ID of a parameter list to be passed to Oracle Graphics. Use a variable of type PARAMLIST. (Default=TOOLS.null_parameter_ list.)

Calling Oracle Forms from 3GL Programs

The IFZCAL function allows you to call an Oracle Forms application from any C language program that is linked to Oracle Forms. The syntax of the IFZCAL function allows you to specify a standard Oracle Forms command line. This feature means you can run a form from a program with the same options that you can use from the command line.

The prototype and constants for the IFZCAL command are located in the ifzcal.h file on the medium that contains your Oracle Forms executables.

Syntax: The following syntax describes the prototype for the IFZCAL() function:

```
int ifzcal(char *command_line, int command_length)
```

where *command_line* specifies the Oracle Forms Runform command line, and *command_length* specifies the number of characters in *command_line*.

The following IFZCAL statement runs the ORDER_ENTRY form in debug mode with the SCOTT username:

```
ifzcal(”f40run order_entry scott/tiger debug=YES”,40);
```

Note: If the operator is not logged into ORACLE and the command line does not invoke the Oracle Forms Login screen, Oracle Forms uses the userid provided in the IFZCAL call. If the command line does not provide a userid, Oracle Forms uses any existing OPS$ login. If the command line provides neither a userid nor an OPS$ login, the Logon screen displays.

If the operator is not logged on to ORACLE and the command line invokes the the Oracle Forms Logon screen (logon_screen=YES), Oracle Forms displays the Logon screen. If the command line provides an OPS$ login, Oracle Forms uses that logon and ignores any operator entry. If the command line does provide an OPS$ login, Oracle Forms uses the userid provided by the operator.

If the operator is logged into ORACLE and the command line does not invoke the Logon screen, Oracle Forms uses the existing connection even if a userid is specified on the command line. If the command line invokes the Logon screen in this case, Oracle Forms ignores any entries by the operator.

Restrictions: Do not use IFZCAL in a user exit. Doing so can ruin data structures. To invoke additional forms when Runform is already running, use OPEN_FORM or CALL_FORM.

CHAPTER

9

Designing for Portability

Oracle Forms applications run on multiple platforms, including both GUI and character–mode. This chapter describes areas to consider when developing portable applications, and includes the following topics:

About Portability

Portability is the quest for uniformity in a diverse environment. Ideally, portability means:

- You can develop an application on one platform, such as a UNIX workstation, and
- You can run the application transparently (without changes) on multiple other platforms, such as MS Windows and Macintosh.

In reality, the different ways that various platforms and windowing systems implement the basics that provide their distinctive ”look and feel” mean that the distinguishing characteristics of one user interface may appear differently on another user interface. For example, the placement and presentation of the Help menu is different on MS Windows, Macintosh, and Motif.

Portability includes:

- Platform portability: the ability to develop and deploy on different platforms.

  Example: Develop on UNIX, deploy on MS Windows and Macintosh.

- Device portability (including variations in screen size and resolution, as well as monochrome / color differences): the ability to develop and deploy on the same platform, but different devices.

  Example: Develop on MS Windows, deploy on both MS Windows large screen, high resolution and MS Windows small screen, low resolution.

- UI portability: the ability to deploy the same application on GUI and character–mode devices.

  Example: Develop on MS Windows, deploy on MS Windows and a VT220.

Planning

A major goal in planning for portability is to anticipate the kinds of user interfaces on which your system will be deployed over time.

Although this information can be difficult to obtain in advance, you may find it worthwhile to consider these questions:

- On which platform do you currently develop applications?
- Are you planning to migrate to a different development platform?
- What platform do your forms operators currently use?
- Are there plans to migrate to another range of deployment interfaces?

For example, a typical development environment might include:

- Forms developers using Motif on large color monitors.
- Sales staff using small–screen MS Windows laptops in the field.
- Forms operators using MS Windows on large–screen monochrome monitors.

To help you plan for developing cross–platform applications, obtain the platform–specific Installation Guide for each platform you're using.

The Porting Process

The general process for creating portable applications includes these stages:

1. Create standards for your application to ensure a similar ”look and feel” both within the application and across platforms.
2. Create the application on the base platform.
3. Copy, re–generate, and run on the target platform, testing for any areas where adjustment is needed for optimum appearance.

   When you port an application, you will be able to run it directly on the target platform. The Toolkit level that Oracle Forms is built on will render all widgets with the native look–and–feel of the target platform.

   However, because of platform–to–platform differences, you may decide to make some minor changes to your original source code so that it meets your requirements on both the original and the target platform.
4. In the Designer, fine–tune the base–platform application.

Setting Standards

One common approach to designing cross–platform applications is to start by creating and then refining standards, based on prototyping efforts and usability testing. Because each company's requirements are unique, the best standards for your company will also be unique.

Consider developing three types of standards:

- Coding standards, including naming standards for all objects and files.
- GUI standards, including standards for screen appearance, such as color, spacing and layout, as well as standards for specific GUI objects, such as buttons and check boxes.
- Usage standards, including look–and–feel standards that ensure that various parts of the application react to user input in the same way.

For guidance in setting UI standards, consult the following platform–specific references:

- MS Windows: *The Windows Interface: An Application Design Guide*
- Motif: *Motif Style Guide*
- Macintosh: *Macintosh Human Interface Guidelines*

Template Forms

In addition to creating a standards document, you may also want to embody the standards in a template form. Then, instead of starting a new form, developers open the template form and save it under a new name. The template form provides a starting point so applications can be developed according to the standards.

Template forms include anything that may be used across all forms in an application:

- object groups
- property classes
- toolbars
- libraries that include shareable pre–written, pre–tested subroutines, such as calculations, validations, and utility routines

Before distributing any templates, be sure to test them on all platforms.

Choosing a Form Coordinate System

The form module property, Coordinate System, controls both the size of objects and their position on the screen. (That is, width and height of objects, as well as X and Y coordinates, are interpreted according to the Coordinate System.) You can set the Coordinate System to Character or Real. Real coordinate units may be set to inches, centimeters, pixels, or points.

If you require GUI to character–mode portability and want to optimize for GUI, set Coordinate System to Real—either inches, centimeters, or points. These coordinate units are based on measurable distances rather than number of physical dots, and therefore are more portable than pixels, which change depending on screen resolution. The Real setting provides maximum flexibility for proportional fonts, but may require some fine–tuning to avoid overlapping fields on the character–mode side.

If you want to optimize for character–mode, set Coordinate System to Character. This setting provides less flexibility for the proportional fonts used on GUIs, but lets you line up character cell boundaries exactly.

For this type of application...	*Set Coordinate System to...*
GUI only	Real: inches, centimeters, or points
Character–mode only	Character
Mixed character–mode and GUI:	
– Optimize for GUI	Real
– Optimize for character–mode	Character

Using Colors

Because your users spend a significant amount of each day using the screens you design, you may want to consult with a human factors engineer for help with choosing colors that are both easy to use and aesthetically pleasing. Research shows that black text on various light–colored backgrounds provides the most contrast for users.

You may want to choose three separate light background colors to signal three categories of information, such as the main window, popup windows, and small objects such as buttons and LOVs.

To ensure cross–platform consistency, you will want to test the colors you choose to be sure they are effective on all platforms.

Color coding of fields is only useful if the user is trained in the meaning of the colors, and uses a monitor that can render them accurately. To avoid problems with monochrome and character mode terminals:

- Test each color and color combination to make sure that all widgets and labels are visible on both color and monochrome monitors.
- Use color redundantly: Color should never be the only cue to a specific meaning. For example, in an accounting application, negative amounts might be shown in red, but they should also be preceded by a minus sign.

Limiting the use of color is essential for portability. Keep in mind the characteristics of the monitors used to run an application. Code to the lowest common denominator for size, resolution, depth, and color.

- A 4–bit VGA monitor will limit your selection of color.
- MS Windows GUI objects can use only one of 16 colors, so even though Oracle Forms provides a 256–color palette, if you want the background of items like checkboxes to be the same color as the canvas, choose the canvas color from the MS Windows 16–color palette.
- If your application must run on both color and monochrome monitors, use a monochromatic color scheme, with a color palette such as black, white, light grey, and dark blue.

Another approach is to start with one master template containing all the application libraries, any standard referenced objects, and the standard color palette. By using this template as a starting point (use Save As to give each individual form a new name), each form will include the standard color palette.

Choosing Fonts

When you develop portable applications, you will want to pay particular attention to your choice of fonts, for both text items and boilerplate items.

Check first for availability of the font you plan to use. Is it available on all the platforms you're using? Then test your font choices on each platform. Even fonts with the same name can have a different appearance on different platforms. For a polished appearance, you'll want to choose the font that looks best on each platform. Use font aliasing to specify font mapping when porting an application from one platform to another.

In the past, many portable applications used Courier, a non–proportional font. However, using Courier has several disadvantages:

- Courier is not available on all platforms (only Geneva and Chicago are required on Macintosh).
- GUI users may expect proportional fonts and reject non–proportional fonts.

Recommendation: To meet user expectations for a proportional font, use:

- MS Windows: MS Sans Serif
- Motif: Helvetica
- Macintosh: Geneva and Chicago

Tip: Remember that using named Visual Attributes to set defaults such as font and color so that they'll change from platform to platform works only for Oracle Forms objects. Boilerplate text, however, remains static. To specify cross–platform font substitution, use font aliasing. (Use named Visual Attributes to change colors dynamically at runtime.)

Font Aliasing

Font aliasing lets you use the *uifont.ali* file to specify what cross–platform font substitution you prefer. Each platform will have a separate *uifont.ali* file to define font substitution.

Example 1:

You develop on MS Windows using the MS Sans Serif font.
To run on UNIX, add this line to your font alias file :

```
MS Sans Serif=Helvetica
```

Example 2:

You develop on UNIX using the Helvetica font.
To run on MS Windows, add this line to your font alias file :

```
Helvetica=MS Sans Serif
```

Example 3:

You develop on UNIX using the Helvetica font.
To run on Macintosh, add this line to your font alias file :

```
Helvetica=Geneva
```

For more information on font aliasing, refer to your platform–specific Installation Guide.

Using Icons

Icons can be used to identify windows, menu items, and buttons. While icons are an essential part of GUI design, they are not inherently portable, so if you are developing cross–platform applications, you will want to limit the number of icons.

- Icon files are not portable, so you will need a complete set of icon files for each platform.
- Image size for all bitmaps must be designed to accommodate the lowest resolution on the platform. For example, a PC with VGA resolution will allow fewer pixels for rendering than a PC with SVGA resolution, so developing in VGA and porting to SVGA will work better than developing in SVGA and porting to VGA.
- Icons will need to be tested on a platform–by–platform basis.

Setting Window Size

Developing portable applications requires limiting the size of windows. For example, you may have to keep window size small enough to accommodate laptop users.

- Applications that must run in character mode should be designed to support a grid of 80 x 22 cells, plus 2 lines for the console (message and status area).
- To avoid scrolling, applications that must run on VGA and SVGA monitors must fit within their respective window size constraints.

Form Functionality

When you’re building portable applications, you may want to use the GET_APPLICATION_PROPERTY built–in to return the following values:

- DISPLAY_HEIGHT
- DISPLAY_WIDTH
- OPERATING_SYSTEM
- USER_INTERFACE

For example, you could use Get_Application_Property (OPERATING_SYSTEM) to obtain the name of the platform this application runs on. Then you could use When–New–Form–Instance to set properties appropriate for the current platform before the form is used.

Recommendations

To implement portable form functionality, consider the following recommendations:

- To call Oracle Reports, Oracle Graphics, and Oracle Book, use RUN_PRODUCT. (Use the HOST built–in only for calling other external applications.)
- Avoid user exits, which must be re–written or at least re–compiled for each platform.
- Isolate platform–specific functionality:
  – MS Windows: DDE, OLE, and VBX.
  – Macintosh: AppleEvents.
- Be aware of differences in Macintosh functionality:
  – Only text items are navigable, other items (such as buttons) are not navigable.
  – Text item beveling does not apply (raised and lowered beveling looks the same).

Character–Mode Considerations

Porting to character mode platforms requires attention to both font and functionality issues:

- Work within limitations caused by differences between proportional and non–proportional fonts.
- Use widget mnemonics to substitute for using the mouse to press buttons, check boxes, and radio buttons (for example, use Alt–P to press a ”Print” button).

For more information, see the *Oracle Forms Reference Manual, Vol. 2,* App. A, ”Compatibility with Prior Versions,” under ”Migration Strategies.”

Running in Character Mode

Running a character–mode application in Oracle Forms 4.5 does not require any special conversion. To run an application in character mode, run the .FMX file from the GUI version using the character–mode executable.

Example:

```
f45run custform.fmx scott/tiger
```

Text Issues in Character Mode

If you are designing on a GUI platform and porting to a character–mode platform, you will need a strategy to account for the difference between proportional and non–proportional fonts. To account for the difference, first count the characters in a prompt, then compute the difference in length based on the size of a character cell grid, thus making sure you have that number of cells available to display the prompt.

To approximate the screen layout in character mode, use a monospaced font, such as Courier, on your GUI platform at design time. (If you use a proportional font, you’ll find that the spaces between labels and fields are too large in the bitmapped version, or the labels and fields overlap in the character mode version.)

Aligning Boilerplate Text in Character Mode

When you are creating a single application to run in both GUI mode and character mode, boilerplate text takes up more space in the monospaced font used in character mode than in the proportional font used in the GUI.

Example 1: Single record block, labels to the left of text items

When boilerplate text is used for text item labels, most applications call for the boilerplate text to be right aligned, ending just before the text item.

To implement this behavior in a portable manner:

1. Multiple–select all boilerplate text in the window.
2. Select Format–>Drawing Options–>Text.

   Oracle Forms displays the Text Drawing Options dialog.
3. Select Horizontal Origin: At Right.

   Oracle Forms sets a snap point at the right.
4. Select Format–>Alignment–>Right.

   Oracle Forms starts at the snap point and allows the text to expand to the left.

Example 2: Multi–record block, labels above text items

In a multi–record block, most applications call for labels *above* text items. To implement this behavior, follow the same steps as above, except select Horizontal Origin: At *Left* and Format–>Alignment–>*Left.*

Properties Restricted to Character Mode Applications

The following properties are limited to character mode applications, or have special restrictions related to character mode applications:

- Help
- Hint (Menu Item)
- Identification
- Size
- Visual Attribute Type: Character Mode Logical Attribute

CHAPTER

10 Object Linking and Embedding (OLE)

Oracle Forms provides support for Object Linking and Embedding (OLE). This chapter includes the following topics:

About OLE

Oracle Forms supports Object Linking and Embedding(OLE) on the Microsoft Windows and Macintosh platforms. OLE provides you with the capability to integrate objects from many application programs into a single compound document. Compound documents enable you to use the features from multiple application programs.

An *OLE server application* creates OLE objects that are embedded or linked in an *OLE container application*; OLE containers store and display OLE objects. Applications can be OLE server applications, OLE container applications, or both. Oracle Forms is an OLE container application, and Oracle Graphics and Microsoft Word are examples of OLE server applications.

An OLE object is *embedded* or *linked* in an OLE container, an item in Oracle Forms. In Oracle Forms, embedded objects become part of the form module, and linked objects are references from a form module to a linked source file.

You can modify OLE objects by activating them. Activating an OLE object provides you access to features from the OLE server application that originated the OLE object. Embedded objects can be activated with *in–place activation* or *external activation.* Linked objects can only be activated with external activation.

OLE server applications can create many object classes. The object classes that an OLE server can create are installed in a *registration database.* If a registration database does not already exist, one is created during installation of an OLE server application. The registration database contains the object classes that are valid for embedding and linking into an OLE container in a form module. For instance, Microsoft Word classes include MS Word 6.0 Document, MS Word 6.0 Picture, and MS WordArt 2.0.

About OLE Objects

OLE objects are created from OLE server applications. Microsoft Word is an OLE server application that creates Word document OLE objects. Another example of an OLE object is a spreadsheet that is created from Microsoft Excel. OLE objects are embedded or linked in OLE container applications such as Oracle Forms.

More specifically, Oracle Forms defines an item called OLE container. An OLE object is embedded or linked into an OLE container in a form module.

Embedded Objects

An embedded OLE object is created by an OLE server application and is embedded in an Oracle Forms form module. An embedded object is stored as part of a form module or as a LONG RAW column in a database. Embedded objects can be queried as OLE container items.

An example of object embedding is to insert a spreadsheet in an OLE container of a form module. The spreadsheet is stored as part of the form module or as a LONG RAW column in a database; there is no separate source file containing the spreadsheet.

Linked Objects

A linked OLE is created by an OLE server application. A linked object is stored in a separate source file that is created from an OLE server application. An image representation of the linked object and the information about the location of the linked object's source file is stored in a form module or as a LONG RAW column in a database. The content of the linked object is not stored as part of a form module or as a LONG RAW column in a database; it is retained in a separate file known as the linked source file.

An example of object linking is to link a word processor document in a form module. An image of the document appears in the OLE container of the form module and the location of the document is stored as part of the form module or as a LONG RAW column in a database.

About OLE Servers and OLE Containers

Applications that support OLE can be OLE servers, OLE containers, or both. An OLE server application creates OLE objects that are embedded or linked in OLE containers. Examples of OLE servers are Microsoft Word and Microsoft Excel.

Unlike OLE server applications, OLE container applications do not create documents for embedding and linking. Instead, OLE container applications provide a place to store, display, and manipulate objects that are created by OLE server applications. Oracle Forms is an example of an OLE container application.

Many of the options available for manipulating an OLE object in an OLE container application are determined by the OLE server application. For instance, options from the OLE popup menu, also known as OLE verbs, are exposed by the OLE server application. The information contained in the registration database, such as object classes available for linking and embedding, also depends on the OLE server application.

About the Registration Database

The registration database stores a set of classes that categorize OLE objects. The information in the registration database determines the object classes that are available for embedding and linking in OLE containers. OLE server applications export a set of classes that become members of the registration database. Each computer has a single registration database. If the registration database does not already exist when an OLE server application is installed, one is created.

A single OLE server application can add many OLE classes to the registration database. The process of adding classes to the registration database is transparent and occurs during the installation of an OLE server application. For instance, when Microsoft Excel is installed, several classes are added to the registration database; some of the classes that are installed in the registration database include Excel Application, Excel Application 5, Excel Chart, Excel Sheet, ExcelMacrosheet, and ExcelWorkSheet.

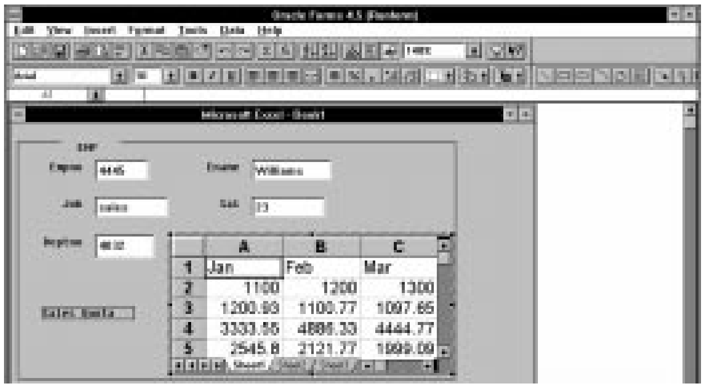
About OLE Object Activation Styles

You can activate an OLE object from an Oracle Forms OLE container with in–place activation or external activation. Activating an OLE object allows you to have access to features from the OLE server application that originated the OLE object. In–place activation and external activation are possible if the OLE server application that originated the OLE object is accessible by your computer.

Both in–place activation and external activation also depend on the OLE activation property settings of the OLE container. The OLE activation property settings are specified in the Oracle Forms Designer. If the OLE server application is accessible, the activation property settings of the OLE container determine whether in–place activation or external activation occurs when an embedded OLE object is activated. Linked objects can only be activated with external activation; in–place activation does not apply to linked objects, even if the in–place activation property is set to True. Activation of an OLE object is possible in the Oracle Forms Designer and during Oracle Forms runtime.

In–place Activation

In–place activation occurs when an OLE container and its embedded object remain in place when activated; Oracle Forms remains surrounding the OLE container. In–place activation is available for embedded objects, but it is not available for linked objects.



	A	B	C
1	Jan	Feb	Mar
2	1100	1200	1300
3	1200.93	1100.77	1097.65
4	3333.55	4886.33	4444.77
5	2545.8	2121.77	1999.09

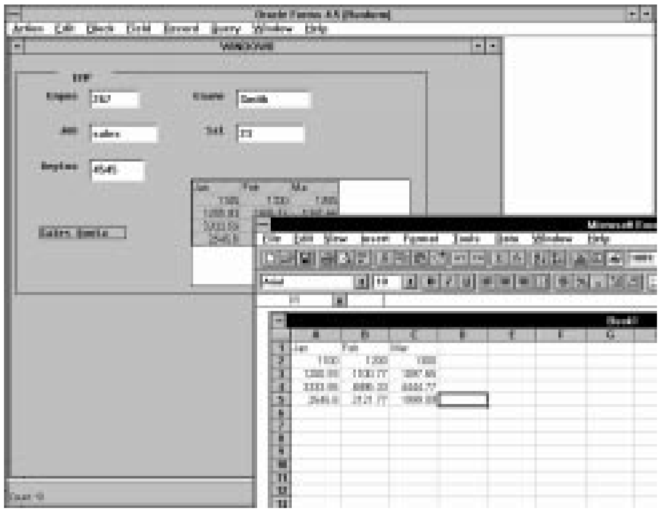
In–place activation of a spreadsheet containing sales data

When an object is activated, the object appears inside a hatched border. To deactivate in–place activation, click anywhere outside the hatched border.

During in–place activation, some menu options of the OLE server application replace Oracle Forms menu options. If the window containing the OLE container has a toolbar, in–place activation also replaces the Oracle Forms toolbar with the OLE server application toolbar. Replacing menu options and toolbars provide access to features that are available from the OLE server application. Oracle Forms menu options and toolbars reappear when you deactivate in–place activation.

External Activation

External activation occurs when an OLE object appears in a separate window that is opened by an object's originating OLE server application. Because an OLE server application starts up with its own windows, Oracle Forms menu options and toolbars remain intact during external activation. External activation is available for both embedded and linked objects.



External activation of a spreadsheet containing sales data

When an OLE object is activated, the object's originating OLE server application is launched, and the OLE object appears in a separate OLE server application window. The separate window has the menu options and toolbars of the OLE server application. To deactivate external activation, you must explicitly exit the OLE server application.

When the contents of a linked source file is modified with external activation, a linked object can be updated manually or automatically. Manual updates require an explicit instruction for an object to reflect changes from a linked source file. Automatic updates occur as soon as you modify a linked source file.

About OLE Automation

OLE automation allows an OLE server application to expose a set of commands and functions that can be invoked from an OLE container application. OLE automation provides a way for an OLE container application to use the features of an OLE server application to manipulate an OLE object from the OLE container environment.

In Oracle Forms, you can use PL/SQL to access any command or function that is exposed by an OLE server. The OLE2 package provides a PL/SQL Application Programming Interface for creating, manipulating, and accessing the commands and functions. For more information about the OLE2 package, refer to the *Oracle Procedure Builder Developer's Guide*. The OLE2 package documentation is also available in Oracle Forms online Help.

When to Embed or Link OLE Objects

Whether an OLE object is embedded or linked depends how often an OLE object is updated, how an OLE object is shared, how the source document is accessed, and if there are storage constraints.

You should use object embedding when:

- The object that you are embedding does not have to be the most current version of data because updating is performed from within a form module instead of a standalone document.
- The object does not need to be included in more than one document, so that changes are only necessary in the object embedded in the form module and nowhere else.
- The source document cannot be accessed if the object is linked. In this case, only a single form module needs to be maintained and not a form module in addition to the source document.
- The form module size is not a concern, because with embedding, the size of the form module increases by approximately the size of the embedded object.

You should use object linking when:

- The object that you are linking has to be the most current version of data because updating is performed from outside a form module on a standalone document.
- The object needs to be included in more than one document, so that changes affect multiple form modules and other documents.
- The source document can always be accessed if the object is linked.
- The form module size is a concern: Linking, unlike embedding, does not increase the size of the form module by the size of the object.

OLE in Oracle Forms

Oracle Forms is an OLE container application that offers the following:

- Embedding and linking of OLE server objects into Oracle Forms OLE containers.
- In–place activation of embedded contents in Oracle Forms OLE containers.
- Programmatic access to OLE objects, properties, and methods through OLE automation support from PL/SQL.
- Seamless storage of OLE objects in a database in LONG RAW columns.

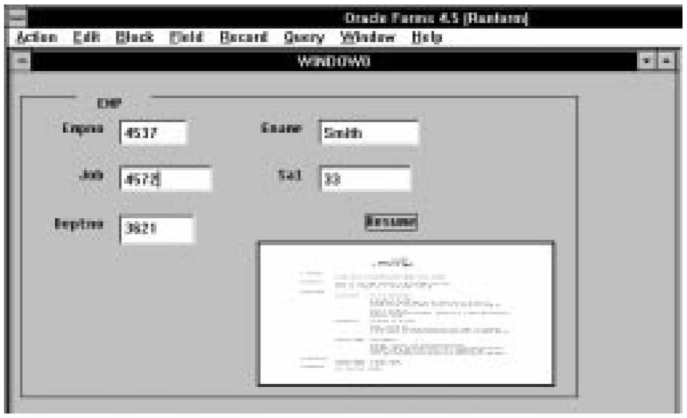
An OLE container is an item in Oracle Forms. An OLE object is linked or embedded into an OLE container. OLE objects can be base table items or control items.

Oracle Forms supports OLE in–place activation. In–place activation allows you to access menus and toolbars from OLE server applications to edit embedded OLE objects while you are in Oracle Forms.

Oracle Forms also supports OLE automation. Using PL/SQL, you can invoke commands and functions that are exposed by OLE servers supporting OLE automation.

From Oracle Forms, you can save OLE objects to a database, as well as query OLE objects from a database. OLE objects are saved as LONG RAW columns in the database. When linked objects are saved, only the image and the link information are retained in the database. The contents of a linked object remains in a linked source file. Saving an embedded object retains all the contents of an embedded object in the database.

An example of using OLE in Oracle Forms is an application that integrates Microsoft Word documents with a form module. The integration of a form module and Microsoft Word document provides you with access to features from both Oracle Forms and Microsoft Word. You can format the Microsoft Word document with any of the text processing features from Microsoft Word, and you can use all of the Oracle Forms features for displaying and manipulating the data from the database.


Oracle Forms 4.5 (Runform)
Action Edit Block Field Record Query Window Help
WINDOW0
EMP
Resume

Empno 4537
Ename Smith
Job 4578
Sal 33
Deptno 3621

Microsoft Word document embedded in a form module

The characteristics of an OLE object in Oracle Forms depends on the properties of the OLE container. OLE container properties determine how OLE objects are activated, displayed, and manipulated.

Using OLE in the Oracle Forms Designer

The Oracle Forms Designer is where the properties are set for OLE containers. Although OLE objects are active in both the Designer and at runtime, the OLE container property values determine OLE functionality at both design time and runtime.

From the Designer, you determine the criteria for OLE object activation, representation, and operation by setting the OLE container properties. For example, when the Show OLE Popup Menu property is set to True, the OLE popup menu is an option from the Layout Editor of the Designer and at runtime. The OLE popup menu appears when the mouse cursor is in the OLE container and the right mouse button is pressed. When the Show OLE Popup Menu property is set to False, the

OLE popup menu is not an option at runtime. The OLE popup menu can offer an array of operations that you can use with OLE objects.

From the Designer, you can restrict the object classes for embedding and linking in an OLE container at runtime. For example, if you specify that the object class of an OLE container is restricted to the ExcelWorkSheet class, only a Microsoft Excel worksheet can be an OLE object in the OLE container. If no object class restriction is specified, any OLE object that is categorized as an object class in the registration database is a potential OLE object in the OLE container.

When an OLE container is created, the properties associated with the OLE container are initially set to default values. The OLE container properties are divided into the following three groups of properties:

- Activation Properties
  - OLE Activation Style
  - OLE Do In Out
  - OLE In–place Activation
- OLE Popup Menu Properties
  - OLE Popup Menu Items
  - Show OLE Popup Menu
- OLE Tenant Properties
  - OLE Class
  - OLE Resize Style
  - OLE Tenant Aspect
  - OLE Tenant Types
  - Show OLE Tenant Type

Activation properties specify whether editing an OLE object is performed with in–place activation or external activation. OLE popup menu properties determine if an OLE popup menu is available and what options appear on the OLE popup menu. OLE Tenant Properties specify conditions that are required for an OLE object in an OLE container.

You can manually insert OLE objects in an OLE container from the Oracle Forms Designer. The OLE object that you insert into an OLE container adheres to the OLE container property settings. Inserting an OLE object into an OLE container from the Oracle Forms Designer initializes the OLE container. The initialization of an OLE container determines the initial content of the OLE container at runtime. At runtime, any form query from a LONG RAW column of the database to populate the OLE container will overwrite the content that is used for initialization purposes.

Using OLE in Oracle Forms at Runtime

The ability to display and manipulate an object in Oracle Forms at runtime is determined by the OLE container properties that are defined in the Oracle Forms Designer. Runtime attributes include activation style, OLE popup menu options, and OLE tenant properties.

The OLE Tenant Aspect property determines how an OLE object is viewed. For instance, when the OLE Tenant Aspect property is set to Icon, any OLE object that you insert in the OLE container appears as an icon.

At runtime, you can manually insert an OLE object into an OLE container, or you can query the database for an OLE object to populate the OLE container. When you manually insert an OLE object at runtime, the OLE object appears in the OLE container until the next record query. The OLE object that you insert into an OLE container adheres to the OLE container property settings. For any subsequent record queries, the OLE container appears in a state as defined in the Designer or populated with an OLE object from the database.

You can also query the database for an OLE object to populate an OLE container. Each record query populates an OLE container with an OLE object from the database. A query to the database for an OLE object can only be made for an OLE container item. An OLE object's data type in the database is LONG RAW. When queried, an OLE object is displayed according to the property settings of the OLE container. Similarly, the operations permitted on the OLE object depend on the property settings of the OLE container. OLE container properties are not saved as part of an OLE object.

Creating an OLE Container in Oracle Forms

You can create an OLE container in a form module from the Oracle Forms Designer. An OLE container is an item in Oracle Forms. When you create an OLE container, the properties that are associated with the OLE container are set to default values. You can change the properties to suit your needs.

The following steps show how to create an OLE container in the Oracle Forms Designer.

To create an OLE container in the Layout Editor:

1. In the Layout Editor tool palette, select the OLE tool.
2. Position and size the OLE container on the canvas.
3. Set the OLE container properties.

   Activation Properties:

   OLE Activation Style

   OLE Do In Out

   OLE In–place Activation

   Popup Menu Properties:

   OLE Popup Menu Items

   Show OLE Popup Menu

   OLE Tenant Properties:

   OLE Class

   OLE Resize Style

   OLE Tenant Aspects

   OLE Tenant Types

   Show OLE Tenant Type

After you create an OLE container, you can insert an OLE object from the Designer or at runtime. If you want to manually insert an OLE object, make sure you set the Show OLE Popup Menu property to True and the Insert Object OLE popup menu option to Display and Enable.

Linking and Embedding OLE Objects

An OLE object that you manually insert into an OLE container from the Oracle Forms Designer initializes the OLE container for runtime. If you manually insert an OLE object at runtime, the OLE object appears in the OLE container until the next record query. For any subsequent record queries, the OLE container appears in a state as defined in the Designer or populated with an OLE object from the database.

Embedding Objects

Access to the Insert Object option on the OLE popup menu is necessary for manual insertion of an OLE object into an OLE container. You can create a new OLE object to insert into an OLE container, or you can embed an OLE object from an existing file into an OLE container.

To embed a new OLE object:

1. Create an OLE container with the OLE Tenant Types property set to Any or Embedded.

   Make sure the Show OLE Popup Menu property is True, and the Insert Object item of the OLE Popup Menu Items is set to Display and Enable.

2. Move the mouse cursor on the OLE container and press the right mouse button to access the OLE popup menu. From the OLE popup menu, choose Insert Object. The Insert Object dialog appears.
3. From the Insert Object dialog, select Create New, choose an object type from the Object Type list, and click OK. The OLE server application starts up.
4. Create the object in the OLE server application.
5. Exit the OLE server application.

To embed an object from an existing file:

1. Create an OLE container with the OLE Tenant Types property set to Any or Embedded.

   Make sure the Show OLE Popup Menu property is True, and the Insert Object item of the OLE Popup Menu Items is set to Display and Enable.
2. Move the mouse cursor on the OLE container and press the right mouse button to access the OLE popup menu. From the OLE popup menu, choose Insert Object. The Insert Object dialog appears.
3. From the Insert Object dialog, select Create from File, and specify the file name of the object to embed.

   You can use the Browse option to help locate the file. If you use the Browse option, click OK from the Browse dialog after selecting a file.
4. After you determine the file name, click OK on the Insert Object dialog.

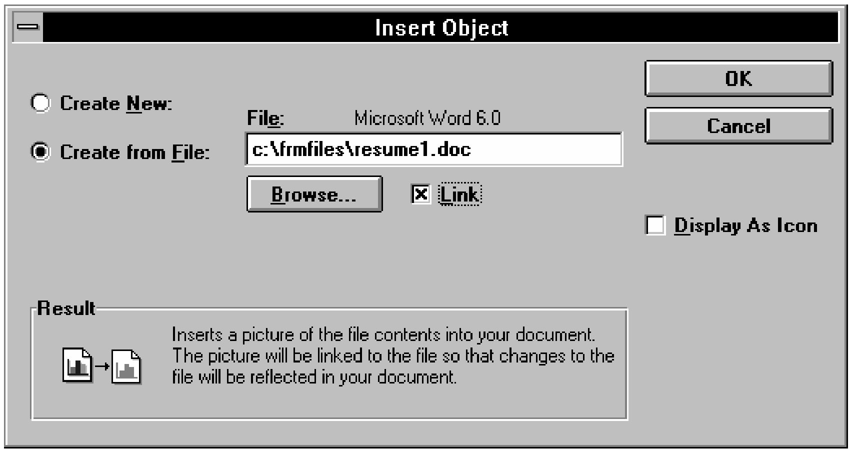
Linking Objects

Because linked objects contain information about the location of an OLE object, you can only link OLE objects from existing files. You cannot create an object when you are trying to link, because the location of new objects is not yet established.

To link an object:

1. Create an OLE container with the OLE Tenant Types property set to Any or Linked.

   Make sure the Show OLE Popup Menu property is True, and the Insert Object item of the OLE Popup Menu Items is set to Display and Enable.
2. Move the mouse cursor on the OLE container and press the right mouse button to access the OLE popup menu. From the OLE popup menu, choose Insert Object. The Insert Object dialog appears.



Create from File: [✓]

File: c:\frmfiles\resume1.doc

Link [✓]

Display As Icon [ ]

Result: Inserts a picture of the file contents into your document. The picture will be linked to the file so that changes to the file will be reflected in your document.

3. From the Insert Object dialog, select Create from File, and specify the file name of the object to link.

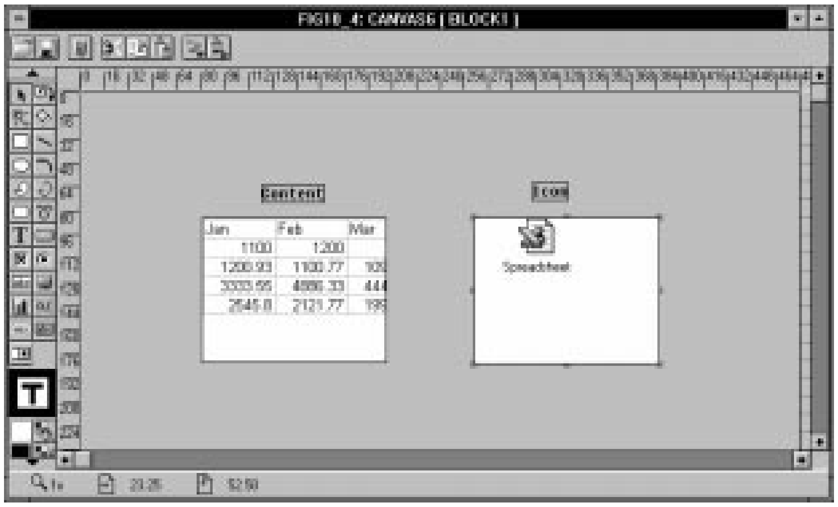
   You can use the Browse option to help locate the file. If you use the Browse option, click OK from the Browse dialog after selecting a file.

4. Check Link on the Insert Object dialog. After you determine the file name, click OK on the Insert Object dialog.

Displaying OLE Objects

You can display an OLE object as any of the following:

- the content of the OLE object as it appears in the OLE server
- an icon of the OLE server application that originated the OLE object
- a thumbnail preview of the OLE object that appears in a reduced view



Jan	Feb	Mar
1100	1200	
1200.93	1100.77	109
3333.55	4886.33	444
2545.8	2121.77	199

You specify the display option with the OLE Tenant Aspect property. You can also choose Display As Icon on the Insert Object dialog of the OLE popup menu. When you select Display As Icon, an icon appears in the OLE container to represent the embedded or linked object. Display As Icon overrides the value of the OLE Tenant Aspect property.

To display an object as an icon using Display As Icon:

1. Create an OLE container.
2. From the Insert Object dialog, determine the object to embed or link. Before closing the Insert Object dialog, select Display As Icon.

   When you select Display As Icon, the current icon that represents the OLE server application that originated the OLE object for insertion is displayed in the Insert Object dialog. Icon selections can be changed by selecting Change Icon.

   The Change Icon dialog shows the current icon selection and the default icon selection. You can change both the current and default icon selections by selecting the desired icon. Different icons are stored in different files. After selecting an icon, click OK from the Change Icon dialog.
3. From the Insert Object dialog, click OK.

Editing OLE Objects

You can edit an embedded or linked object by activating the OLE object. Activating an OLE object causes the the OLE server application that originated the OLE object to start up.

When the OLE server application is not available on your computer for editing an embedded or linked object, editing can be performed by permanently or temporarily converting the OLE object to another format. For more information on OLE object conversion, see the section, "Converting OLE Objects."

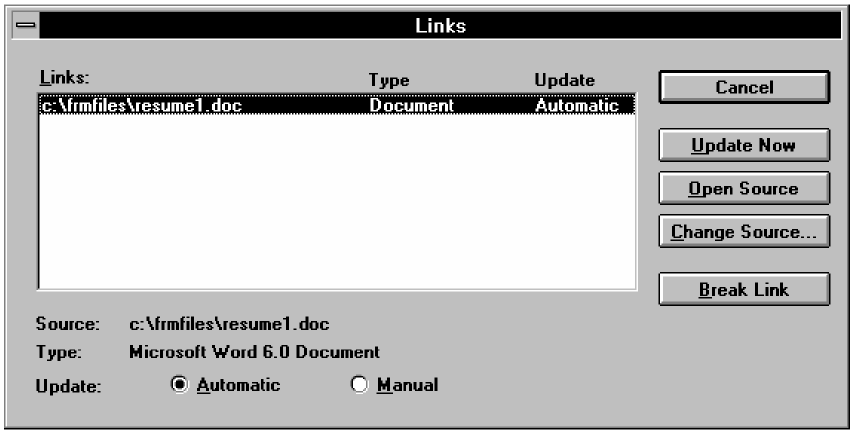
Editing Embedded Objects

When an embedded object is modified, changes are immediately reflected in the form module, because embedded objects are stored as part of the form module.

Editing Linked Objects

When a source file of a linked object is modified, the linked object is updated automatically or manually. Automatic updates reflect modifications to an OLE object immediately. Manual updates require that you select Update Now from the Links dialog after the modification of a document.

Automatic or manual updating is set in the Links dialog. Through the Links dialog, you can open a linked source file for editing, change links of an OLE object to link to other sources, or break links completely. Broken links leave only a fixed image of an OLE object in an OLE container; no information about the location of the linked source file is retained.



Links:	Type	Update
c:\frmfiles\resume1.doc	Document	Automatic

Source: c:\frmfiles\resume1.doc
Type: Microsoft Word 6.0 Document
Update: [✓ Automatic] Manual

To update a linked object automatically:

1. Move the mouse cursor on the OLE container.

   Make sure the Show OLE Popup Menu property is True, and the Links item of the OLE Popup Menu Items is set to Display and Enable.

2. With the mouse cursor on the OLE container, use the right mouse button to access the OLE popup menu. From the OLE popup menu, choose Links. The Links dialog appears.

3. The Links dialog shows the current update option for the selected linked source file. Select the Automatic radio button that appears next to the Update label.

4. Click Close to save your changes and exit the Links dialog.

To open a linked source file from the Links dialog:

1. Move the mouse cursor on the OLE container.

   Make sure the Show OLE Popup Menu property is True, and the Links item of the OLE Popup Menu Items is set to Display and Enable.

2. With the mouse cursor on the OLE container, use the right mouse button to access the OLE popup menu. From the OLE popup menu, choose Links. The Links dialog appears.

3. The Links dialog shows the current links to source files. Select a link to a source file, and select Open Source. The OLE server application that originated the linked source file starts up. Edit the linked source file.

4. Exit the OLE server application to save your changes and return to Oracle Forms.

To change an existing link from one linked source file to another:

1. Move the mouse cursor on the OLE container.

   Make sure the Show OLE Popup Menu property is True, and the Links item of the OLE Popup Menu Items is set to Display and Enable.

2. With the mouse cursor on the OLE container, use the right mouse button to access the OLE popup menu. From the OLE popup menu, choose Links. The Links dialog appears.

3. The Links dialog shows the current links to source files. Select a link to a source file, and select Change Source. The Change Source dialog appears. Select another linked source file. Click OK on the Change Source dialog to establish a new link.

4. Select Close on the Links dialog to return to Oracle Forms.

To break a link to a linked source file:

1. Move the mouse cursor on the OLE container.

   Make sure the Show OLE Popup Menu property is True, and the Links item of the OLE Popup Menu Items is set to Display and Enable.

2. With the mouse cursor on the OLE container, use the right mouse button to access the OLE popup menu. From the OLE popup menu, choose Links. The Links dialog appears.

3. The Links dialog shows the current links to source files. Choose a link to a source file, and select Break Link. Notice the link is removed from the Links dialog.

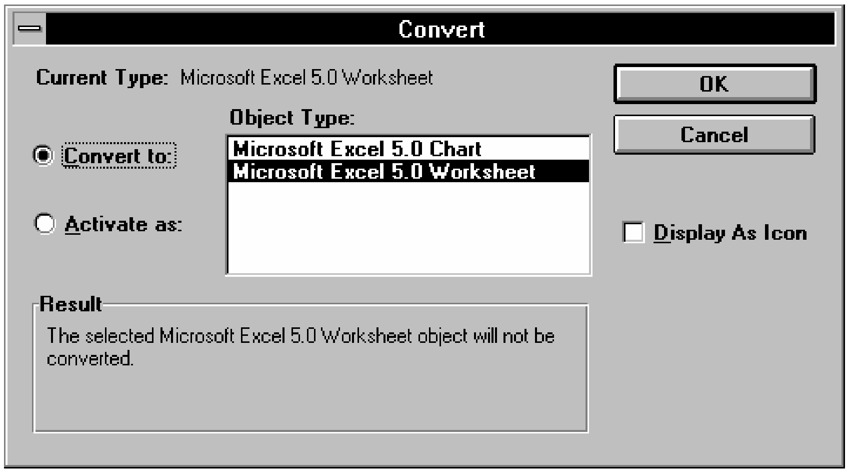
4. Select Close from the Links dialog to return to Oracle Forms.

Converting OLE Objects

You can convert an OLE object from one format to another. OLE object conversion is used for editing OLE objects when the OLE server application that originated an OLE object is not available.

The convert option is available on the Object submenu of the OLE popup menu. You can permanently convert the object to a new format, or you can temporarily convert the object to a new format for editing purposes.

Display As Icon in the Convert dialog allows you to display the converted object as an icon. For more information on displaying an object as an icon, see the section, "Displaying OLE Objects."



Current Type: Microsoft Excel 5.0 Worksheet

(● Convert to:) Microsoft Excel 5.0 Worksheet

☐ Display As Icon

The selected Microsoft Excel 5.0 Worksheet object will not be converted.

Converting Embedded Objects

You can convert embedded objects from one format to another.

To convert an embedded object:

1. Move the mouse cursor on the OLE container.

   Make sure the Show OLE Popup Menu property is True, and the Object item of the OLE Popup Menu Items is set to Display and Enable.

2. With the mouse cursor on the OLE container, use the right mouse button to access the OLE popup menu. From the OLE popup menu, choose Convert from the Object submenu. The Convert dialog appears.

3. The Convert dialog shows the current object type and the conversion possibilities. Select a conversion object type.
4. Select either Convert To or Activate As. Convert To permanently alters the format of the object to the selected object type. Activate As is a temporary conversion that provides a method for editing or viewing an object when an OLE server application cannot be found.
5. Click OK on the Convert dialog to save changes.

Converting Linked Objects

Because linked object source files are not part of a form module, you cannot convert linked objects from one format to another in Oracle Forms. You can only convert linked objects from the source level.

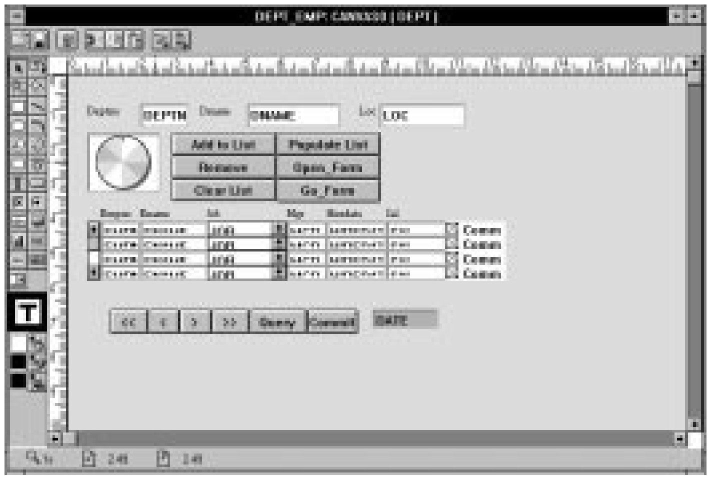
CHAPTER

11 VBX Controls

Oracle Forms for Microsoft Windows provides support for VBX controls. This chapter includes the following topics:

About VBX Controls

VBX controls provide a simple method of building and enhancing user interfaces. The controls can be used to obtain user input and display program output. VBX controls were originally developed as extensions for the Microsoft Visual Basic environment, and include such items as sliders, grids, and knobs.



The knob is a VBX control

VBX controls are a special type of dynamic link library distributed on files with the .VBX extension. A single VBX file can contain many different VBX controls, and each VBX control is defined by a set of properties, events, methods, and error messages. Refer to the documentation that accompanies your VBX control for more information on properties, events, methods, and error messages.

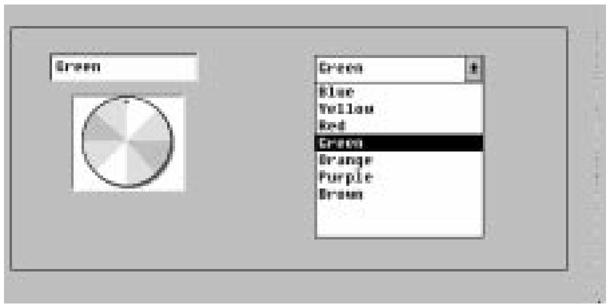
Although you can develop your own VBX controls for use in Oracle Forms, developing VBX controls requires a significant amount of effort. The procedure for developing VBX controls is documented in the Microsoft Visual Basic Professional Edition Manuals. However, it is recommended that you use VBX controls developed by third party vendors.

VBX Controls in Oracle Forms

A VBX control is an item in Oracle Forms. Like other Oracle Forms items, VBX controls serve as a way to represent and manipulate data that displays on a form. VBX controls can be base table items or control items.

VBX Control as an Oracle Forms Item

You can interchange a VBX control with other Oracle Forms items without affecting your intended use for the item. A text item in Oracle Forms displays data from the database on a form. A VBX control can accomplish the same task. For example, a text item displaying the number 10 can be depicted by a VBX control that is a knob. Both items can reflect changes in the data from the database. When the number 10 changes to the number 5, the number 5 appears in the text item on the form, and the knob control redirects its position to represent the number 5.



Both the knob VBX control and the poplist can be used to make selections

A VBX control, like some Oracle Forms items, can also be used to input data. An item on a form can require the entry of a number in the range from 1 to 10. Using a poplist, you can display the choices on a list for selection. Similarly, a VBX control that is a knob can be used to provide choices by turning the knob. Instead of the choices appearing in the poplist, the choices are depicted by the position of the knob.

VBX controls can be used in place of Oracle Forms items when any of the following conditions are met:

- A VBX control is the best representation for your data in Oracle Forms.
- A VBX control is the best method of data input into Oracle Forms.
- You want a simple method of enhancing the user interface to build professional Oracle Forms applications.

The Value of a VBX Control

Like other items in Oracle Forms, a VBX control can store values such as the number 10 or the character string 'BLUE'. The value of a VBX control is derived from the item property, VBX Control Value Property. Oracle Forms uses the setting of the VBX Control Value Property for database querying and setting. Use standard PL/SQL bind variable syntax to set the value of a VBX control.

The value of the VBX Control Value Property can be any one of the control's scalar–valued properties. Many scalar–valued properties can exist for any single VBX control, but not all the properties make sense as the VBX Control Value Property for a control's intended use.

Oracle Forms sets the VBX Control Value Property after a VBX control name is specified. In many cases, a VBX control has a default property that denotes the value of the control. If there exists a default property that denotes the value of the control, the default property is set to the VBX Control Value Property.

For example, the knob VBX control has a property called *Value Current*. *Value Current* is the default property that denotes the value of the knob control. After selecting to use the knob control, the VBX Control Value Property is set to the knob control property, *Value Current*. Other properties of the knob control, such as *Value Initial*, *Value Maximum*, and *Value Minimum*, are valid values to assign to the VBX Control Value Property. However, these VBX control properties may not provide meaningful data as the VBX Control Value Property.

As previously illustrated, the VBX control that is a knob can be used to represent a number in a range of values. If you expect the knob control to represent the current value, *Value Current* is the appropriate setting for the VBX Control Value Property. In this scenario, if the VBX Control Value Property is set to the knob control property *Value Minimum*, the knob always represents the minimum value in the range. Although the *Value Minimum* knob control property is a valid setting for the VBX Control Value Property, it is not meaningful in this particular instance.

You can set the VBX Control Value Property from the Oracle Forms Designer. In addition, you can programmatically get and set the VBX Control Value Property by using the VBX.GET_VALUE_PROPERTY and VBX.SET_VALUE_PROPERTY built–in subprograms from the VBX package in Oracle Forms. For more information on the VBX built–in subprograms, refer to the *Oracle Forms Reference Manual, Vol. 1*.

VBX Controls in the Oracle Forms Designer

Use the Oracle Forms Designer to create a VBX control for use with Oracle Forms. From the Designer, you can see both the Oracle Forms VBX control properties and VBX control properties.

Oracle Forms VBX Control Properties

Oracle Forms VBX control properties are listed in the Functional section of the Properties window. Oracle Forms VBX control properties include VBX Control File, VBX Control Name, and VBX Control Value Property. For more information on the Oracle Forms VBX control properties, refer to the *Oracle Forms Reference Manual, Vol. 2*.

VBX Control Properties

VBX control properties are the properties that are part of a VBX control definition. VBX control properties vary with each VBX control, although some properties are common to several VBX controls. For information about the properties of a particular VBX control, refer to the documentation for the specific VBX control.

In the Oracle Forms Designer, most of the VBX control properties appear in the Miscellaneous section of the Properties window. Where possible, Oracle Forms maps VBX control properties to equivalent Oracle Forms item properties (many of which are visual attributes). For example, the VBX control property BackColor is mapped to the Oracle Forms Background Color property.

The VBX control properties that are mapped to Oracle Forms properties do not appear in the Miscellaneous section of the Properties window. Instead, the mapped VBX properties appear in the Properties window as Oracle Forms properties.

This is a list of VBX properties that are mapped to Oracle Forms Properties:

VBX Property	*Oracle Forms Property*
BackColor	Background Color
Enabled	Enabled
FontBold	Font Weight
FontItalic	Font Style
FontName	Font Name
FontSize	Font Size
FontStrikeThru	Font Style
FontUnderline	Font Style
ForeColor	Foreground Color
Height	Height
Left	X Position (in Oracle Forms coordinates)
TabIndex*	
TabStop*	
Top	Y Position (in Oracle Forms coordinates)
Visible	Displayed
Width	Width

*These properties do not have a one–to–one mapping with Oracle Forms properties. The properties are handled internally by Oracle Forms and do not appear in the Properties window.

VBX Controls in Oracle Forms at Runtime

You can interact with VBX controls in Oracle Forms in many ways. Because VBX controls are defined with a set of properties, events, and methods, you can decide the behavior of a VBX control in an Oracle Forms application. You can interact with a VBX control in Oracle Forms by doing any of the following:

- responding to VBX control events
- firing VBX control events
- getting VBX control properties
- setting VBX control properties
- invoking VBX control methods

Responding to VBX Control Events

You can define the course of action that takes place following any VBX control event. For instance, you can increase a value in a field on a form when the spin button VBX control spins upward. Similarly, you can decrease a value in a field on a form when the spin button spins downward. To perform the appropriate action, you must be able to detect whether a spin button spins upward or downward. You can determine whether the spin button VBX control spins upward or downward by responding to the events raised by the spin button VBX control. In this case, the spin button VBX control raises a SpinUp event when a button spins upward and a SpinDown event when a button spins downward. Events vary among VBX controls. For information about the events of a particular VBX control, refer to the documentation for the specific VBX control.

To capture a VBX control event:

1. Create a VBX control.
2. Create a When–Custom–Item–Event trigger attached to the VBX control item.
3. In the When–Custom–Item–Event trigger, read the value of the system variable SYSTEM.CUSTOM_ITEM_EVENT to determine the name of the event that fires the trigger.

   The When–Custom–Item–Event trigger fires any time a VBX control event is raised. For instance, the When–Custom–Item–Event trigger fires when a knob control is turned or a spin button control is pressed. The system variable SYSTEM.CUSTOM_ITEM_EVENT stores the VBX control event that triggered the custom item event. When the spin button control is pressed, either a SpinDown or SpinUp event is raised. VBX control events are case sensitive.

Example:

```
/*
** TRIGGER:        When–Custom–Item–Event
** ITEM:           SpinButton VBX Control
** EXAMPLE:        Capture events and determine course of action
*/
IF :System.Custom_Item_Event = ’SpinDown’ THEN
    :QTY := :QTY – 1;
ELSE
    IF :System.Custom_Item_Event = ’SpinUp’ THEN
      :QTY := :QTY + 1;
    END IF;
END IF:
```

Firing VBX Control Events

VBX controls raise events in response to runtime interaction. You can also raise VBX events explicitly in PL/SQL to cause a particular course of action. For example, you can explicitly raise the VBX ClickIn event of the knob VBX control.

The event raised by a VBX control can be obtained from the SYSTEM.CUSTOM_ITEM_EVENT system variable. Some VBX control events have event parameters. You can obtain VBX control event parameters from the system variable SYSTEM.CUSTOM_ITEM_EVENT_PARAMETER. For information on system variables, refer to the *Oracle Forms Reference Manual, Vol. 1*.

To fire a VBX control event:

1. Create a trigger or a user–defined subprogram.
2. Call the VBX.FIRE_EVENT built–in subprogram and specify the VBX control item that should raise the event, and the VBX control event to raise.

   If the VBX control event you want to raise requires event parameters, you must also specify a parameter list containing the required event parameters. If the VBX control event to raise does not require event parameters, the value of the parameter list is NULL.

Syntax:

```
VBX.FIRE_EVENT(item_id, event_name, paramlist_id);
VBX.FIRE_EVENT(item_id, event_name, paramlist_name);
VBX.FIRE_EVENT(item_name, event_name, paramlist_id);
VBX.FIRE_EVENT(item_name, event_name, paramlist_name);
```

Example 1:

```
/*
** Built–in:      VBX.FIRE_EVENT
** Example:       Raises AddList event of combo box VBX
**                control and adds an entry to the combo box.
**                The AddList event requires an event parameter
**                –– a value to add to in the combo box.
*/
DECLARE
    ItemName      VARCHAR2(40) := ’COMBOBOX’;
    PL_NAME       VARCHAR2(20) := ’EventParam’;
    PL_ID         PARAMLIST;
BEGIN
    PL_ID := Create_Parameter_List(PL_NAME);
    Add_Parameter(PL_NAME,’ENTRY’,TEXT_PARAMETER,’BLUE’);
    VBX.FIRE_EVENT(ItemName,’AddList’,PL_NAME);
END;
```

Example 2:

```
/*
** Built–in:       VBX.FIRE_EVENT
** Example:        Raises SpinDown event of spin button control.
**                 The SpinDown event does not require an event
**                 parameter.
*/
DECLARE
   ItemName        VARCHAR2(40) := ’SPINBUTTON’;
BEGIN
   VBX.FIRE_EVENT(ItemName,’SpinDown’,NULL);
END;
```

Getting VBX Control Properties

VBX control properties describe the attributes of a VBX control. VBX properties can be used for many purposes. For example, you can specify a course of action based on the value of a VBX control property, or you can set other Oracle Forms properties based on the value of a VBX control property.

You can programmatically obtain the current setting of a VBX control property using the VBX.GET_PROPERTY built–in subprogram. For information about the properties of a particular VBX control, refer to the documentation for the specific VBX control. For information on the VBX.GET_PROPERTY built–in subprogram, refer to the *Oracle Forms Reference Manual, Vol. 1*.

Example 1:

```
/*
** Built–in:       VBX.GET_PROPERTY
** Example:        Uses the VBX.GET_PROPERTY built–in to obtain
**                 the CURRTAB property of the VBX item named
**                 TABCONTROL.  The property value of CURRTAB is
**                 returned to the TabNumber variable and is used
**                 as input in the user–defined Goto_Tab_Page
**                 subprogram.
** Trigger:        When–Custom–Item–Event
*/
DECLARE
   TabEvent        VARCHAR2(80);
   TabNumber       CHAR;
BEGIN
   TabEvent := :system.custom_item_event;
   IF (UPPER(TabEvent) = ’CLICK’) then
     TabNumber := VBX.Get_Property(’TABCONTROL’,’CurrTab’);
     Goto_Tab_Page(TO_NUMBER(TabNumber));
   END IF;
END;
```

Some VBX control properties are arrays of properties. Like scalar–valued properties, you can programmatically obtain the current setting of a VBX control property using the VBX.GET_PROPERTY built–in subprogram.

Example 2:

```
/*
** Built–in:       VBX.GET_PROPERTY
** Example:        Uses the VBX.GET_PROPERTY built–in to obtain an
**                 indentation value from the Indent property
**                 array of the VBX control named OUTLINE.
**                 The Indent property value is returned to the
**                 IndentVal variable and is used as input to the
**                 user–defined SetIndent subprogram.
** Trigger:        When–Custom–Item–Event
*/
DECLARE
    ClickEvent     VARCHAR2(80);
    IndentVal      CHAR;
BEGIN
    ClickEvent := :system.custom_item_event;
    IF (UPPER(ClickEvent) = ’CLICK’) then
      IndentVal := VBX.Get_Property(’OUTLINE’,’Indent[2]’);
      SetIndent(TO_NUMBER(IndentVal));
    END IF;
END;
```

Setting VBX Control Properties

VBX control properties describe the attributes of a VBX control. VBX properties can be used for many purposes. For example, you can specify a course of action by setting the value of a VBX control property.

You can set a VBX control property from the Designer, or you can programmatically set the VBX control property using the VBX.SET_PROPERTY built–in subprogram. For information about the properties of a particular VBX control, refer to the documentation for the specific VBX control. For information on the VBX.SET_PROPERTY built–in subprogram, refer to the *Oracle Forms Reference Manual, Vol. 1.*

Example 1:

```
/*
** Built–in:       VBX.SET_PROPERTY
** Example:        Uses the VBX.SET_PROPERTY built–in to set the
**                 Index property of the SpinButton VBX control.
** Trigger:        When–Button–Pressed
*/
DECLARE
    ItemName       VARCHAR2(40) := ’SPINBUTTON’;
    VBX_VAL_PROP   VARCHAR2(40);
    VBX_VAL        VARCHAR2(40);
```

```
BEGIN
   IF :System.Custom_Item_Event = 'SpinDown' THEN
     VBX_VAL_PROP := 'Index';
     VBX_VAL := '5';
     VBX.Set_Property(ItemName,VBX_VAL_PROP,VBX_VAL);
   END IF;
END;
```

Some VBX control properties are arrays of properties. Like scalar–valued properties, you can programmatically set the VBX control property using the VBX.SET_PROPERTY built–in subprogram.

Example 2:

```
/*
** Built-in:       VBX.SET_PROPERTY
** Example:        Uses the VBX.SET_PROPERTY built-in to set the
**                 value of an element in the Indent property
**                 array of the VBX control named OUTLINE.
** Trigger:        When-Button-Pressed
*/
DECLARE
   ItemName        VARCHAR2(40) := 'OUTLINE';
BEGIN
   IF :System.Custom_Item_Event = 'Click' THEN
     Set_Property(ItemName,'Indent[2]','5');
   END IF;
END;
```

Invoking VBX Methods

Some VBX controls include a set of methods as part of their definition. You can invoke a VBX control method using the VBX.INVOKE_METHOD built–in subprogram. Depending on the definition of the method, the appropriate course of action takes place. For example, you can add an item to a combo box VBX control by invoking the ADDITEM method. VBX method names are case sensitive.

For information about the methods of a particular VBX control, refer to the documentation for the specific VBX control. For information on the VBX.INVOKE_METHOD built–in subprogram, refer to the *Oracle Forms Reference Manual, Vol. 1*.

Example:

```
/*
** Built–in: VBX.INVOKE_METHOD_PROPERTY
** Example:  Adds an entry to a combo box. The entry to
**           add to the combo box is the first optional argument.
**           The position where the entry appears is the second
**           optional argument.
*/
DECLARE
   ItemName       VARCHAR2(40) := ’COMBOBOX’;
BEGIN
   VBX.Invoke_Method(ItemName,’ADDITEM’,’blue’,’2’);
END;
```

Creating a VBX Control in Oracle Forms

You can create a VBX control in the Oracle Forms Designer.

To create a VBX control:

1. In the Layout Editor tool palette, select the VBX control tool.
2. Locate and size the VBX control container on the canvas.
3. Open the Properties window for the VBX control container.
4. Specify the VBX control file name in the VBX Control File field of the Properties window. You can double–click on the VBX Control File Property field to browse the file system for .VBX files.
5. Specify the VBX control name in the VBX Control Name field of the Properties window. Double–click on the VBX Control Name field for a listing of valid VBX controls. You must explicitly select a VBX control name even if the VBX control file contains a single VBX control.

   Once you have defined the VBX control, you can associate a trigger to the VBX control by creating a VBX control item–level When–Custom–Item–Event trigger. Provide the appropriate built–in or user–defined program code in the trigger to suit your needs.

CHAPTER

12

Dynamic Data Exchange (DDE)

This chapter provides information on Dynamic Data Exchange(DDE) in Oracle Forms. The topics covered include:

About DDE

Oracle Forms for Microsoft Windows supports Dynamic Data Exchange(DDE). DDE is a mechanism by which applications can communicate and exchange data. DDE client support is a procedural extension to Oracle Forms for Microsoft Windows. A PL/SQL package for DDE support, consisting of the functions listed in this chapter, provides application developers with an Application Programming Interface(API) for accessing DDE functionality from within PL/SQL procedures and triggers.

The DDE functions enable Oracle Forms, a DDE client application, to communicate with DDE server applications in the following ways:

- by importing data
- by exporting data
- by executing commands against the DDE server

Note: The information in this chapter assumes that you have a working knowledge of DDE, as implemented under Microsoft Windows.

Limitations

Oracle Forms does not include the following:

- Data linking (advise transaction)

  Oracle Forms cannot automatically receive an update notice when a data item has changed.

- Server support

  Oracle Forms cannot respond to commands or requests for data from a DDE client; Oracle Forms must initiate the DDE conversation (although data may still be transferred in either direction).

Function Overview

As part of the DDE package, the DDE functions available from within PL / SQL procedures and triggers can be grouped into the following categories:

- Support functions
- Connect / Disconnect functions
- Transaction functions
- Data type Translation functions

Support Functions

These functions are used to start and stop other DDE Server applications.

DDE.APP_BEGIN	Begins a DDE server application.
DDE.APP_END	Ends a DDE server application.
DDE.APP_FOCUS	Focuses a DDE server application.

Connect/Disconnect Functions

These functions are used to connect to and disconnect from DDE server applications.

DDE.INITIATE	Starts a DDE conversation with a DDE server application.
DDE.TERMINATE	Ends a DDE conversation with a DDE server application.

Transaction Functions

These functions are used to exchange data with DDE server applications.

DDE.EXECUTE	Executes a command recognized by a DDE server application.
DDE.POKE	Supplies information to a DDE server application.
DDE.REQUEST	Requests information from a DDE server application.

Data Type Translation Functions

These functions are used to translate DDE data type constants to strings and back; in addition, DDE.GETFORMATNUM allows users to register a new data format that is not predefined by Microsoft Windows. Note that these functions do not translate the data itself (all DDE data is represented with the CHAR data type in PL/SQL), just data type constants.

DDE.GETFORMATNUM	Convert/register a data format string to a number.
DDE.GETFORMATSTR	Convert a data format number to a string.

DDE.APP_BEGIN

Syntax: DDE.APP_BEGIN (Command, Mode)

Parameters:

Parameter	Type	Description
Command	VARCHAR2	command line for executing a program
Mode	PLS_INTEGER	application window starting mode

Returns: PLS_INTEGER

Description: Begins a DDE server application program.

In the Command string, the DDE server application program name can include a path. If the DDE server application name does not include a path, then the following directories are searched:

- the current directory
- the Windows directory
- the Windows system directory
- the directory containing the executable file for the current task
- the directories listed in the PATH environment variable
- the directories mapped in a network

The DDE server application program name may be followed by arguments, which should be separated from the application program name with a space.

You can start the DDE server application in either normal, minimized, or maximized size with the *mode* parameter. The application mode constants for DDE.APP_BEGIN are the following:

Mode	*Description*
DDE.APP_MODE_NORMAL	Start application window in normal size.
DDE.APP_MODE_MINIMIZED	Start application window in minimized size.
DDE.APP_MODE_MAXIMIZED	Start application window in maximized size.

This function returns an application identifier that is a PLS_INTEGER. The application identifier is used in all subsequent calls to DDE.APP_END and DDE.APP_FOCUS for that application window.

Example:

```
DECLARE
   AppID   PLS_INTEGER;
BEGIN
   /* Start Microsoft Excel with spreadsheet emp.xls loaded. */
   AppID := DDE.App_Begin('C:\excel\excel.exe C:\emp.xls',
                          DDE.APP_MODE_MINIMIZED);
END;
```

DDE.APP_END

Syntax: DDE.APP_END (AppID)

Parameters:

Parameter	Type	Description
AppID	PLS_INTEGER	Application identifier that is returned from DDE.APP_BEGIN

Description: Ends a DDE server application program that DDE_APP_BEGIN starts.

The DDE server application can also be terminated in standard Microsoft Windows fashion, such as by double–clicking the Control menu.

Restrictions: To terminate a DDE server application, the DDE server application must have been previously started by calling DDE.APP_BEGIN.

Example:

```
DECLARE
    AppID  PLS_INTEGER:
BEGIN
    /* Start Microsoft Excel with spreadsheet emp.xls loaded. */
    AppID := DDE.App_Begin('C:\excel\excel.exe C:\emp.xls'
                           DDE.APP_MODE_NORMAL);
    ...
    /* End Microsoft Excel program. */
    DDE.App_End(AppID);
END;
```

DDE.APP_FOCUS

Syntax: DDE.APP_FOCUS (AppID)

Parameters:

Parameter	Type	Description
AppID	PLS_INTEGER	application identifier returned by DDE.APP_BEGIN

Description: Activates a DDE server application program started by DDE.APP_BEGIN.

The application can also be activated in standard Microsoft Windows fashion, such as by clicking within the application window.

Restrictions: To activate a DDE server application, the DDE server application must have been previously started by calling DDE.APP_BEGIN.

Example:

```
DECLARE
   AppID   PLS_INTEGER;
BEGIN
   /* Start Microsoft Excel application in fullscreen mode. */
   AppID := DDE.App_Begin('C:\excel\excel.exe,
                          DDE.APP_MODE_MAXIMIZED);
   /* Activate the application window. */
   DDE.App_Focus(AppID);
END;
```

DDE.EXECUTE

Syntax: DDE.EXECUTE (ConvID, Command, Timeout)

Parameters:

Parameter	Type	Description
ConvID	PLS_INTEGER	DDE conversation id
Command	VARCHAR2	Command string for the server to execute
Timeout	PLS_INTEGER	Timeout duration in milliseconds

The value of *Command* depends on what values are supported by the server application.

Timeout specifies the maximum length of time, in milliseconds, that this routine waits for a response from the DDE server application. If the user specifies an invalid number, such as negative number, then the default value of 1000 ms is used. The duration of the timeout depends on machine configuration and the DDE server application.

Description: Executes a command string that is acceptable in the receiving DDE server application.

Example:

```
DECLARE
   ConvID   PLS_INTEGER;
BEGIN
   /* Open a DDE conversation with Microsoft Excel for Windows */
   ConvID := DDE.Initiate('EXCEL', 'C:\abc.xls');
   /* Recalculate the Excel spreadsheet.*/
   DDE.Execute(ConvID, '[calculate.now()]', 1000);
END;
```

DDE.GETFORMATNUM

Syntax: DDE.GETFORMATNUM (FormatStr)

Parameters:

Parameter	Type	Description
FormatStr	VARCHAR2	Data format name string

Returns: PLS_INTEGER

Description: Translates or registers a specified data format name and returns a format number.

This function converts a data format from a string to a number. This number can be used in Poke and Request transactions to represent the DataFormat parameter.

If the specified name is not registered, this function registers it and returns a unique format number. This is the only way to use a format in a Poke or Request transaction that is not one of the predefined formats.

This function returns the numeric representation of the data format string that is a PLS_INTEGER.

See the section, "Microsoft Windows Predefined Data Formats," later in this chapter for the predefined data format constants.

Example:

```
DECLARE
    FormatNum   PLS_INTEGER;
    MyFormatNum PLS_INTEGER;
BEGIN
    /* Get predefined format number for "CF_TEXT"
      (should return CF_TEXT=1). */
    FormatNum := DDE.GetFormatNum('CF_TEXT');
    /* Register a user-defined data format called "MY_FORMAT". */
    MyFormatNum := DDE.GetFormatNum('MY_FORMAT');
END;
```

DDE.GETFORMATSTR

Syntax: DDE.GETFORMATSTR (FormatNum)

Parameters:

Parameter	Type	Description
FormatNum	PLS_INTEGER	Data format number

Returns: CHAR

Description: Translates a data format number into a format name string.

This function returns the string representation of the data format number that is a CHAR.

This function returns a data format name if the data format number is valid. Valid format numbers include the predefined formats in the section, “Microsoft Windows Predefined Data Formats,” and any user–defined formats that were registered with DDE.GETFORMATNUM.

Example:

```
DECLARE
    FormatStr   CHAR(20);
BEGIN
    /* Get a data format name
       (should return the string ’CF_TEXT’) */
    FormatStr := DDE.GetFormatStr(CF_TEXT);
END;
```

DDE.INITIATE

Syntax: DDE.INITIATE (Service, Topic)

Parameters:

Parameter	Type	Description
Service	VARCHAR2	DDE Server application’s DDE service code
Topic	VARCHAR2	Topic name for the conversation

The values of *Service* and *Topic* depend on the values supported by a particular DDE server application. *Service* is usually the name of the DDE server application program. For applications that operate on file–based documents, *Topic* is usually the document filename; in addition, the System topic is usually supported by each service.

Returns: PLS_INTEGER

Description: Opens a DDE conversation with a DDE server application.

An application can start more than one conversation at a time with multiple services and topics, provided the conversation identifiers are not interchanged.

This function returns the DDE conversation identifier that is a PLS_INTEGER. The conversation identifier that is returned must be used in all subsequent Execute, Poke, Request, and Terminate calls for that conversation.

To terminate the conversation, you should call DDE.TERMINATE.

Example:

```
DECLARE
   ConvID   PLS_INTEGER;
BEGIN
   /* Open a DDE conversation with Microsoft Excel for Windows
      on topic abc.xls. */
   ConvID := DDE.Initiate(’EXCEL’, ’c:\abc.xls’);
END;
```

DDE.POKE

Syntax: DDE.POKE (ConvID, Item, Data, DataFormat, Timeout)

Parameters:

Parameter	Type	Description
ConvID	PLS_INTEGER	DDE conversion identifier that is returned by DDE.INITIATE
Item	VARCHAR2	Data item name to which the data is to be sent
Data	VARCHAR2	Data buffer to send
DataFormat	PLS_INTEGER	Format of outgoing data
Timeout	PLS_INTEGER	Timeout duration in milliseconds

The value of *Item* depends on what values are supported by the DDE server application on the current conversation topic.

The predefined data format constants listed in the section, "Microsoft Windows Predefined Data Formats," can be used for *DataFormat.* A user–defined format that is registered with DDE.GETFORMATNUM can also be used, provided the DDE server application recognizes the format. It is your responsibility to ensure that the DDE server application processes the specified data format.

Timeout specifies the maximum length of time, in milliseconds, that this routine waits for a response from the DDE server application. If you specify an invalid number, such as negative number, then the default value of 1000 ms is used. The duration of the timeout depends on machine configuration and the DDE server application.

Description: Sends data to a DDE server application.

Example:

```
DECLARE
    ConvID   PLS_INTEGER;
BEGIN
    /* Open a DDE conversation with Microsoft Excel for Windows on
      topic abc.xls. */
    ConvID = DDE.Initiate(’EXCEL’, ’C:\abc.xls’);
    /* Send data ”foo” to cell at row 2, column 2. */
    DDE.Poke(ConvID, ’R2C2’, ’foo’, DDE.CF_TEXT, 1000);
END;
```

DDE.REQUEST

Syntax: DDE.REQUEST (ConvID, Item, Buffer, DataFormat, Timeout)

Parameters:

Parameter	Type	Description
ConvID	PLS_INTEGER	DDE conversion identifier returned by DDE.INITIATE
Item	VARCHAR2	Requested data item name
Buffer	VARCHAR2	Result data buffer
DataFormat	PLS_INTEGER	Format of requested data
Timeout	PLS_INTEGER	Timeout duration in milliseconds

The value of *Item* depends on what values are supported by the DDE server application on the current conversation topic.

It is your responsibility to ensure that the return data buffer is large enough for the requested data. If the buffer size is smaller than the requested data, the data is truncated.

The predefined data format constants listed in the section, "Microsoft Windows Predefined Data Formats," can be used for *DataFormat*. A user–defined format that is registered with DDE.GETFORMATNUM can also be used, provided the DDE server application recognizes this format. It is your responsibility to ensure that the DDE server application will process the specified data format.

Timeout specifies the maximum length of time, in milliseconds, that this routine waits for a response from the DDE server application. If you specify an invalid number, such as negative number, then the default value of 1000 ms is used. The duration of the timeout depends on machine configuration and the DDE server application.

Description: Requests data from a DDE server application.

Example:

```
DECLARE
    ConvID   PLS_INTEGER;
    Buffer   CHAR(100);
BEGIN
    /* Open a DDE conversation with Microsoft Excel for Windows on
      topic abc.xls. */
    ConvID := DDE.Initiate('EXCEL', 'C:\abc.xls');
    /* Request data from 6 cells between row 2, column 2
      and row 3, column 4. */
    DDE.Request(ConvID, 'R2C2:R3C4', Buffer, DDE.CF_TEXT,1000);
END;
```

DDE.TERMINATE

Syntax: DDE.TERMINATE (ConvID)

Parameters:

Parameter	Type	Description
ConvID	PLS_INTEGER	DDE conversion identifier returned by DDE.INITIATE

Description: Terminates the specified conversation with a DDE server application.

After the DDE.TERMINATE call, all subsequent Execute, Poke, Request, and Terminate requests using the terminated conversation identifier will result in an error.

Restrictions: You should use DDE.INITIATE to start a DDE conversation with a server application before attempting to use DDE.TERMINATE.

Example:

```
DECLARE
    ConvID   PLS_INTEGER;
BEGIN
    /* Open a DDE conversation with Microsoft Excel for Windows
      on topic abc.xls. */
    ConvID := DDE.Initiate('EXCEL', 'C:\abc.xls');
    ...
    /* Terminate the Microsoft Excel for Windows conversation */
    DDE.Terminate(ConvID);
END;
```

Microsoft Windows Predefined Data Formats

Format	*Description*
DDE.CF_BITMAP	The data is a bitmap.
DDE.CF_DIB	The data is a memory object containing a BITMAPINFO structure followed by the bitmap data.
DDE.CF_DIF	The data is in Data Interchange Format (DIF).
DDE.CF_DSPBITMAP	The data is a bitmap representation of a private format. This data is displayed in bitmap format in lieu of the privately formatted data.
DDE.CF_ DSPMETAFILEPICT	The data is a metafile representation of a private data format. This data is displayed in metafile–picture format in lieu of the privately formatted data.
DDE.CF_DSPTEXT	The data is a textual representation of a private data format. This data is displayed in text format in lieu of the privately formatted data.
DDE.CF_ METAFILEPICT	The data is a metafile.
DDE.CF_OEMTEXT	The data is an array of text characters in the OEM character set. Each line ends with a carriage return–linefeed (CR–LF) combination. A null character signals the end of the data.
DDE.CF_ OWNERDISPLAY	The data is in a private format that the clipboard owner must display.
DDE.CF_PALETTE	The data is a color palette.
DDE.CF_PENDATA	The data is for the pen extensions to the Microsoft Windows operating system.
DDE.CF RIFF	The data is in Resource Interchange File Format (RIFF).
DDE.CF_SYLK	The data is in Microsoft Symbolic Link (SYLK) format.
DDE.CF_TEXT	The data is an array of text characters. Each line ends with a carriage return–linefeed (CR–LF) combination. A null character signals the end of the data.
DDE.CF_TIFF	The data is in Tag Image File Format (TIFF).
DDE.CF_WAVE	The data describes a sound wave. This is a subset of the CF_RIFF data format; it can be used only for RIFF WAVE files.

Exceptions

Exception	*Description*
DDE.DDE_APP_FAILURE	An application program specified in a DDE.APP_BEGIN call could not be started.
DDE.DDE_APP_NOT_FOUND	An application ID specified in a DDE.APP_END or DDE.APP_FOCUS call does not correspond to an application that is currently running.
DDE.DDE_FMT_NOT_FOUND	A format number specified in a DDE.GETFORMATSTR call is not known.
DDE.DDE_FMT_NOT_REG	A format string specified in a DDE.GETFORMATNUM call does not correspond to a predefined format and could not be registered as a user–defined format.
DDE.DDE_INIT_FAILED	The application was unable to initialize DDE communications, which caused a call to the DDE layer to fail.
DDE.DDE_PARAM_ERR	An invalid parameter, such as a NULL value, is passed to a DDE package routine.
DDE.DMLERR_BUSY	A transaction failed because the server application is busy.
DDE.DMLERR_DATAACKTIMEOUT	A request for a synchronous data transaction has timed out.
DDE.DMLERR_EXECACKTIMEOUT	A request for a synchronous execute transaction has timed out.
DDE.DMLERR_ INVALIDPARAMETER	A parameter failed to be validated. Some of the possible causes are as follows: The application used a data handle initialized with a different item–name handle or clipboard data format than that required by the transaction. The application used an invalid conversation identifier. More than one instance of the application used the same object.
DDE.DMLERR_MEMORY_ERROR	A memory allocation failed.

Exception	*Description*
DDE.DMLERR_NO_CONV_ ESTABLISHED	A client’s attempt to establish a conversation has failed. The service or topic name in a DDE.INITIATE call may be in error.
DDE.DMLERR_NOTPROCESSED	A transaction failed. The item name in a Poke or Request transaction may be in error.
DDE.DMLERR_POKEACKTIMEOUT	A request for a synchronous poke transaction has timed out.
DDE.DMLERR_POSTMSG_FAILED	An internal call to the PostMessage function has failed.
DDE.DMLERR_SERVER_DIED	The server terminated before completing a transaction.
DDE.DMLERR_SYS_ERROR	An internal error has occurred in the DDE layer.

CHAPTER

13 PL / SQL Interface to Foreign Functions

This chapter examines the PL / SQL interface for invoking foreign functions. The topics covered in this chapter include:

About the PL/SQL Interface

Foreign functions are subprograms written in a 3GL programming language that allow you to customize your Oracle Forms applications to meet the unique requirements of your users. Foreign functions are often used to enhance performance or provide additional functionality to Oracle Forms.

In Oracle Forms, you can invoke a foreign function from a PL/SQL interface. A PL/SQL interface allows you to call foreign functions using PL/SQL language conventions.

Foreign functions that you can invoke from a PL/SQL interface must be contained in a dynamic library. Examples of dynamic libraries include dynamic link libraries on Microsoft Windows, and shared libraries on UNIX systems.

Creating a PL/SQL interface to foreign functions requires the use of the ORA_FFI built–in package (Oracle Foreign Function Interface). The ORA_FFI package consists of a group of PL/SQL built–in subprograms for creating a PL/SQL interface to foreign functions. For more details on the ORA_FFI package, refer to the *Oracle Procedure Builder Developer's Guide*. The ORA_FFI package documentation is also available in Oracle Forms online Help.

Note: An alternative approach for invoking a foreign function is from a user exit interface. Using a user exit interface to invoke foreign functions occasionally requires relinking Oracle Forms Runform. For more information about the user exit interface, refer to Chapter 3, "User Exit Interface to Foreign Functions." Because invoking foreign functions from a user exit interface can require relinking Oracle Forms Runform, using a PL/SQL interface provides a looser bind than that of a user exit interface.

About Foreign Functions

Foreign functions are subprograms written in a 3GL programming language for customizing Oracle Forms applications. Foreign functions can interact with Oracle databases, and Oracle Forms variables and items. Although it is possible to access Oracle Forms variables and items, you cannot call Oracle Forms built–in subprograms from a foreign function.

Foreign functions can be used to perform the following tasks:

- Replace default Oracle Forms processing when running against a non–Oracle data source using transactional triggers.
- Perform complex data manipulation.
- Pass data to Oracle Forms from operating system text files.
- Manipulate LONG RAW data.
- Pass entire PL / SQL blocks for processing by the server.
- Control real time devices, such as a printer or a robot.

  Note: You should not perform host language screen I / O from a foreign function. This restriction exists because the runtime routines that a host language uses to perform screen I / O conflict with the routines that Oracle Forms uses to perform screen I / O. However, you can perform host language *file* I / O from a foreign function.

Foreign functions that can be invoked from a PL / SQL interface must be contained in a dynamic library. The procedure for creating a dynamic library depends on your operating system. For more information on creating dynamic libraries in your environment, refer to the documentation for your operating system.

Types of Foreign Functions

You can develop the following types of foreign functions:

- Oracle Precompiler foreign functions
- OCI (ORACLE Call Interface) foreign functions
- non–ORACLE foreign functions

You can also develop foreign functions that combine both the ORACLE Precompiler interface and the OCI.

Oracle Precompiler Foreign Functions An Oracle Precompiler foreign function incorporates the Oracle Precompiler interface. This interface allows you to write a subprogram in one of the following supported host languages with embedded SQL commands:

- Ada
- C
- COBOL
- FORTRAN
- Pascal
- PL/I

Note: Not all operating systems support all of the listed languages. For more information on supported languages, refer to the Oracle Forms documentation for your operating system.

With embedded SQL commands, an Oracle Precompiler foreign function can access Oracle databases as well as Oracle Forms variables and items. You can access Oracle Forms variables and items by using a set of Oracle Precompiler statements that provide this capability.

Because of the capability to access both Oracle databases and Oracle Forms variables and items, most of your foreign functions will be Oracle Precompiler foreign functions. For more information on the Oracle Precompiler interface, refer to the *Programmer's Guide to the Oracle Precompilers.*

Oracle Call Interface(OCI) Foreign Functions An OCI foreign function incorporates the Oracle Call Interface. This interface allows you to write a subprogram that contains calls to Oracle databases. A foreign function that incorporates only the OCI (and not the Oracle Precompiler interface) cannot access Oracle Forms variables and items. For more information on the OCI, refer to the *Programmer's Guide to the Oracle Call Interface.*

Non–Oracle Foreign Functions A non–Oracle foreign function does not incorporate either the Oracle Precompiler interface or the OCI. For example, a non–Oracle foreign function might be written entirely in the C language. A non–Oracle foreign function cannot access Oracle databases, or Oracle Forms variables and items.

Precompiler Statements

All Oracle Precompiler foreign functions can use host language statements to perform procedural operations. Precompiler foreign functions can also use the following types of statements to perform additional functions such as accessing the database and manipulating Oracle Forms variables and items. For more information on the following Oracle Precompiler statements, refer to Chapter 3, "User Interface to Foreign Functions."

Statement	*Use*
EXEC SQL	Performs SQL commands.
EXEC TOOLS GET	Retrieves values from Oracle Forms to a foreign function.
EXEC TOOLS SET	Sends values from a foreign function to Oracle Forms.
EXEC TOOLS MESSAGE	Passes a message from a foreign function to display in Oracle Forms.
EXEC TOOLS GET CONTEXT	Obtains context information previously saved in a foreign function.
EXEC TOOLS SET CONTEXT	Saves context information from one foreign function for use in subsequent foreign function invocations.
EXEC ORACLE	Executes Oracle Precompiler options.

An Oracle Precompiler foreign function source file includes host programming language statements and Oracle Precompiler statements with embedded SQL statements. Precompiling an Oracle Precompiler foreign function replaces the embedded SQL statements with equivalent host programming language statements. After precompiling, you have a source file that you can compile with a host language compiler. For more information on a specific precompiler, refer to the appropriate precompiler documentation for your environment.

Creating a PL/SQL Interface to Foreign Functions

Creating a PL / SQL interface to a foreign function involves the following steps:

- Initializing the foreign function
- Associating a PL / SQL subprogram with the foreign function
- Mimicking the foreign function prototype with PL / SQL

A PL / SQL package encapsulates the components that are used in creating a PL / SQL interface to a foreign function. You can include PL / SQL interfaces to multiple foreign functions in a single PL / SQL package. For each PL / SQL interface to a foreign function, you must follow the procedure of initializing the foreign function, associating a PL / SQL subprogram with the foreign function, and mimicking the foreign function prototype in PL / SQL. Alternatively, you can opt to include a PL / SQL interface to a single foreign function in a PL / SQL package.

Initializing a Foreign Function

Initializing a foreign function identifies the location of a dynamic library and dissects the foreign function prototype. The initialization process provides a one–to–one match between foreign function host language data types and PL / SQL data types. Initialize a foreign function in a PL / SQL package body.

To initialize a foreign function:

1. Use ORA_FFI.LOAD_LIBRARY to obtain a library handle to the dynamic library containing the foreign function. You must provide the name and location of the dynamic library.
2. Use ORA_FFI.REGISTER_FUNCTION to obtain a function handle to the foreign function. You must provide the library handle and name of the foreign function.
3. Use ORA_FFI.REGISTER_PARAMETER to register the foreign function parameter types. For each parameter, provide the function handle to identify the foreign function and a corresponding PL / SQL equivalent parameter type. You must register parameters in the order they appear in the foreign function prototype.
4. Use ORA_FFI.REGISTER_RETURN to register the foreign function return type. You must provide the function handle to identify the foreign function and a corresponding PL / SQL equivalent return type.

For more details about the ORA_FFI package, refer to the *Oracle Procedure Builder Developer's Guide*. The ORA_FFI package documentation is also available in Oracle Forms online Help.

Here is an example of initializing a foreign function:

```
PACKAGE BODY calc IS
BEGIN
    /*
      This example shows how to initialize the foreign function
      int ADD(int X, int Y) that is contained in the dynamic
      library CALC.DLL;
    */
    lh_calc := ora_ffi.load_library('c:\mathlib\', 'calc.dll');
    fh_add := ora_ffi.register_function(lh_calc,'add',
                                         ora_ffi.C_STD);
    ora_ffi.register_return(fh_add,ORA_FFI.C_INT);
    ora_ffi.register_parameter(fh_add,ORA_FFI.C_INT);
    ora_ffi.register_parameter(fh_add,ORA_FFI.C_INT);
END;
```

Associating a PL/SQL Subprogram with a Foreign Function

By associating a PL / SQL subprogram with a foreign function, you can invoke the foreign function each time you call the associated PL / SQL subprogram. Associating a foreign function with a PL / SQL subprogram is necessary because an Oracle Forms uses PL / SQL constructs. The associated PL / SQL subprogram specifies the memory location of the foreign function code to be executed.

To associate a PL/SQL subprogram with a foreign function:

1. Obtain a function handle using ORA_FFI.FIND_FUNCTION or ORA_FFI.REGISTER_FUNCTION.
2. In the declaration section of the body of your PL / SQL package, define a PL / SQL subprogram that accepts a parameter of type ORA_FFI.FUNCHANDLETYPE as the first parameter followed by PL / SQL data type equivalents for all other foreign function parameters.
3. Include a PRAGMA interface in the PL / SQL subprogram.

   The PRAGMA statement invokes the foreign function by passing control to the memory location in Oracle Forms that can communicate with dynamic libraries.

For more details on the ORA_FFI package, refer to the *Oracle Procedure Builder Developer's Guide*. The ORA_FFI package documentation is also available in Oracle Forms online Help.

Here is an example of associating a subprogram with a foreign function:

```
PACKAGE BODY calc IS
    /*
      This example shows how to associate a PL/SQL subprogram with
      the given function prototype, int ADD(int X, int Y);
    */

    FUNCTION ff_ADD(ff_handle ORA_FFI.FUNCHANDLETYPE,
                    X IN BINARY_INTEGER,
                    Y IN BINARY_INTEGER);
      RETURN BINARY_INTEGER;
      PRAGMA interface(C, ff_ADD, 11265);

BEGIN
    ...
END;
```

Mimicking a Foreign Function Prototype with PL/SQL

Once you have associated a PL / SQL subprogram with a foreign function, Oracle Forms invokes the foreign function whenever you call the PL / SQL subprogram. The associated subprogram requires the first parameter to be a foreign function handle, a parameter type that does not have a host language equivalent in the foreign function prototype.

To provide a PL / SQL interface that has a one–to–one correspondence between the parameter and return data types of the foreign function, you must mimic the prototype of the foreign function using PL / SQL; this is the PL / SQL interface that is made public.

To mimic a foreign function prototype with PL/SQL:

1. In the declaration section of the body of your PL / SQL package, define a PL / SQL subprogram that accepts and returns PL / SQL data type equivalents of the foreign function parameters; this is the subprogram that mimics the foreign function prototype.
2. From the body of the mimicking subprogram, call the PL / SQL subprogram associated with the foreign function.
3. Enter a PL / SQL subprogram prototype in the specification of your PL / SQL package.

Here is an example of mimicking a foreign function definition:

```
PACKAGE calc IS
   FUNCTION ADD(X IN BINARY_INTEGER, Y IN BINARY_INTEGER)
         RETURN BINARY_INTEGER;
END;

PACKAGE BODY calc IS
   /*
     Given the foreign function prototype, int ADD(int X, int y),
     and the associated PL/SQL subprogram,
     FUNCTION ff_ADD(ff_handle,X,Y),
     this example shows how to mimic the foreign function
     definition in PL/SQL.
   */

   FUNCTION ADD(X IN BINARY_INTEGER, Y IN BINARY_INTEGER)
         RETURN BINARY_INTEGER IS
   BEGIN
     RETURN(ff_ADD(ff_ADD_HANDLE,X,Y));
   END;

BEGIN
   ...
END;
```

Invoking a Foreign Function from a PL/SQL Interface

After defining a PL / SQL interface to a foreign function, you can invoke the foreign function using PL / SQL from Oracle Forms. To invoke the foreign function from PL / SQL, you specify the PL / SQL package name followed by a dot and the PL / SQL function name. This example shows how to invoke the foreign functions Get_Image and Show_Image from the ImageLib PL / SQL package.

```
PROCEDURE display_image(keywrd IN OUT VARCHAR2) IS
   img_ptr        ORA_FFI.POINTERTYPE;
BEGIN
   img_ptr := ImageLib.Get_Image(keywrd);
   ImageLib.Show_Image(img_ptr,2);
END;
```

Invoking a foreign function from a PL / SQL interface passes process control to the foreign function. Upon completion of the foreign function, process control is returned to Oracle Forms.

Passing Parameter Values to a Foreign Function from Oracle Forms

Passing parameter values, such as Oracle Forms variables and items, to a foreign function that is invoked from a PL/SQL interface is similar to passing parameters to any PL/SQL subprogram. Be sure to register the parameter values when you create the PL/SQL interface.

After assigning an Oracle Forms variable or item value to a PL/SQL variable, pass the PL/SQL variable as a parameter value in the PL/SQL interface of the foreign function. The PL/SQL variable that is passed as a parameter must be a valid PL/SQL data type; it must also be the appropriate parameter type as defined in the PL/SQL interface.

```
/*
   The variables X_Val and Y_Val contain values obtained from
   Oracle Forms and are used as parameter values in the ADD
   foreign function.
*/
DECLARE
   X_Val   BINARY_INTEGER := :addblk.Xitm;
   Y_Val   BINARY_INTEGER := :addblk.Yitm;
   sum     BINARY_INTEGER;

BEGIN
   sum := ADD(X_Val, Y_Val);
END;
```

Returning a Value from a Foreign Function to Oracle Forms

Returning a value from a foreign function to an Oracle Forms variable or item is similar to returning a value from any PL/SQL function. Be sure to register the return value when you create the PL/SQL interface.

You obtain a return value from a foreign function by assigning the return value to an Oracle Forms variable or item. Make sure that the Oracle Forms variable or item is the same data type as the return value from the foreign function.

```
/*
   The BINARY_INTEGER that the ADD foreign function returns is
   assigned to the item sum_itm of the block addblk.
*/
BEGIN
   addblk.sum_itm:= ADD(X_Val, Y_Val);
END;
```

Simplifying Complex Parameter Data Types

The ORA_FFI package only supports simple PL / SQL parameter and return data types that correspond to the C programming language. When complex parameter and return data types are used in foreign function prototypes, you must simplify or substitute the parameter and return type. If simplification is not possible, use a user exit interface instead.

For example, there is no PL / SQL equivalent to a C language structure data type. You can simplify the structure data type by creating a foreign function that builds the structure. You cannot create a PL / SQL interface to a foreign function without simplifying the complex parameter type.

MainStruct is an example of a complex parameter type:

```
Foreign_Func(MainStruct a, int b, char c)
{
/* process MainStruct */
...
}
```

To simplify the the MainStruct parameter, consider the following approach:

```
New_Foreign_Func(int x, float y, char* z, int b, char c)
{
     struct MainStruct
    {
      int x;
      float y;
      char* z;
    }
/* process MainStruct */
...
}
```

An Example of Creating a PL/SQL Interface to a Foreign Function

Although creating a PL / SQL interface is similar on most platforms, this section describes aspects of Oracle Forms that is specific to its use in Microsoft Windows. For information about other environments, refer to the Oracle Forms documentation for your operating system.

This example invokes Windows Profile String functions from a PL / SQL interface. The PL / SQL package OraWinProfile contains the PL / SQL interface to the Windows Profile String functions.

This is the PL / SQL package specification:

```
PACKAGE OraWinProfile IS
    /*
      Function WritePrivateString calls the Windows
      WritePrivateProfileString function
    */
    FUNCTION WritePrivateString(pSection IN VARCHAR2,
                                pEntry IN VARCHAR2,
                                pString IN VARCHAR2,
                                pFilename IN VARCHAR2)
      RETURN BOOLEAN;

    /*
      Function GetPrivateString calls the Windows
      GetPrivateProfileString function
    */
    FUNCTION GetPrivateString(pSection IN VARCHAR2,
                              pEntry IN VARCHAR2,
                              pDefault IN VARCHAR2,
                              pReturnBuffer IN OUT VARCHAR2,
                              pReturnBufferN IN BINARY_INTEGER,
                              pFilename IN VARCHAR2)
      RETURN BINARY_INTEGER;
END OraWinProfile;
```

This is the PL / SQL package body:

```
PACKAGE BODY OraWindProfile IS
    lh_window      ora_ffi.libHandleType;
    lh_forms       ora_ffi.libHandleType;
    fh_wpps        ora_ffi.funcHandleType;
    fh_gpps        ora_ffi.funcHandleType;

    /*
      icd_wpps function acts as the interface to the
      WritePrivateString function in Windows
    */
    FUNCTION icd_wpps(funcHandle IN ora_ffi.funcHandleType,
                      pSection IN OUT VARCHAR2,
                      pEntry IN OUT VARCHAR2,
                      pString IN OUT VARCHAR2,
                      pFilename IN OUT VARCHAR2)
      RETURN BINARY_INTEGER;
      PRAGMA interface(c,icd_wpps,11265);

    /*
      icd_gpps function acts as the interface to the
      GetPrivateString function in windows
```

```
*/
FUNCTION icd_gpps(funcHandle IN ora_ffi.funcHandleType,
                  pSection IN OUT VARCHAR2,
                  pEntry IN OUT VARCHAR2,
                  pDefault IN OUT VARCHAR2,
                  pReturnBuffer IN OUT VARCHAR2,
                  pRetrunBufferN IN BINARY_INTEGER,
                  pFilename IN OUT VARCHAR2)
  RETURN BINARY_INTEGER;
  PRAGMA(c,icd_gpps,11265);

FUNCTION WritePrivateString(pSection IN VARCHAR2,
                            pEntry IN VARCHAR2,
                            pString IN VARCHAR2,
                            pFilename IN VARCHAR2)
  RETURN BOOLEAN IS
  /* WritePrivateString calls the Windows function */
  /* Make copies of in out arguments */
  lSection VARCHAR2(1024) := pSection;
  lEntry VARCHAR(1024) := pEntry;
  lString VARCHAR2(1024) := pString;
  lFilename VARCHAR(1024) := pFilename;

BEGIN
  /*
  validate arguments––although NULL is a valid argument
  for Windows functions, we are going to prohibit this
  case because of the format of the string that is
  returned
  */
  IF (lSection is NULL) OR (lEntry is NULL) OR
     (lString is NULL) OR (lFilename is NULL) THEN
      RAISE VALUE_ERROR;
  END IF;
  RETURN(icd_wpps(fh_wpps,lSection,lEntry,lString,
       lFilename)<>0);
END WritePrivateString;

FUNCTION GetPrivateString(pSection IN VARCHAR2,
                          pEntry IN VARCHAR2,
                          pDefault IN VARCHAR2,
                          pReturnBuffer IN OUT VARCHAR2,
                          pReturnBufferN IN BINARY_INTEGER,
                          pFilename IN VARCHAR2)
  RETURN BINARY_INTEGER IS
  /* GetPrivateString calls the Windows function */
  /* Make copies of in out arguments */
  lSection VARCHAR2(1024) := pSection;
  lEntry VARCHAR(1024) := pEntry;
```

```
    lDefault VARCHAR2(1024) := pDefault;
    lFilename VARCHAR(1024) := pFilename;

  BEGIN
    /*
    validate arguments––although NULL is a valid argument for
    Windows functions, we are going to prohibit this case
    because of the format of the string that is returned
    */
    IF (lSection is NULL) OR (lEntry is NULL) OR
       (lDefault is NULL) OR (lFilename is NULL) OR
       (pReturnBufferN <= 0) THEN
        RAISE VALUE_ERROR;
    END IF;
    /* Pad the buffer with spaces */
    pReturnBuffer := rpad(’ ’, pReturnBufferN);
    RETURN(icd_gpps(fh_gpps,lSection,lEntry,lDefault,
         pReturnBuffer,pReturnBufferN,lFilename));
  END GetPrivateString;

BEGIN
  /* Declare a library handle to the Windows 386 Kernel */
  lh_windows := ora_ffi.load_library(NULL,’krnl386.exe’);

  /*
    Register the components of WritePrivateString
     This is function prototype for WritePrivateProfileString:
     BOOL WritePrivateProfileString(LPCSTR lpszSection,
                                    LPCSTR lpszEntry,
                                    LPCSTR lpszString,
                                    LPCSTR lpszFilename)
  */
  fh_wpps := ora_ffi.register_function(lh_windows,
                               ’WritePrivateProfileString’,
                                ora_ffi.PASCAL_STD);

  ora_ffi.register_return(fh_wpps,ORA_FFI.C_SHORT);
  ora_ffi.register_parameter(fh_wpps,ORA_FFI.C_CHAR_PTR);
  ora_ffi.register_parameter(fh_wpps,ORA_FFI.C_CHAR_PTR);
  ora_ffi.register_parameter(fh_wpps,ORA_FFI.C_CHAR_PTR);
  ora_ffi.register_parameter(fh_wpps,ORA_FFI.C_CHAR_PTR);

  /*
    Register the components of GetPrivateString
    This is function prototype for GetPrivateProfileString:
    int GetPrivateProfileString(LPCSTR lpszSection,
                                LPCSTR lpszEntry,
                                LPCSTR lpszDefault,
                                LPCSTR lpszReturnBuffer,
                                int cbReturnBuffer,
                                LPCSTR lpszFilename)
    */
    fh_gpps := ora_ffi.register_function(lh_windows,
                                     'GetPrivateProfileString',
                                      ora_ffi.PASCAL_STD);
    ora_ffi.register_return(fh_gpps,ORA_FFI.C_INT);
    ora_ffi.register_parameter(fh_wpps,ORA_FFI.C_CHAR_PTR);
    ora_ffi.register_parameter(fh_wpps,ORA_FFI.C_CHAR_PTR);
    ora_ffi.register_parameter(fh_wpps,ORA_FFI.C_CHAR_PTR);
    ora_ffi.register_parameter(fh_wpps,ORA_FFI.C_CHAR_PTR);
    ora_ffi.register_parameter(fh_wpps,ORA_FFI.C_INT);
    ora_ffi.register_parameter(fh_wpps,ORA_FFI.C_CHAR_PTR);
END OraWinProfile;
```

Here is an example of how to invoke the WritePrivateProfileString and the GetPrivateProfileString functions from a PL / SQL interface in Oracle Forms:

```
DECLARE
    tmpb BOOLEAN;
    tmpn NUMBER;
    buffer VARCHAR(1024) := '';
    buffern BINARY_INTEGER := 1024;
BEGIN
    tmpb := OraWinProfile.WritePrivateString('Section',
                                             'Entry',
                                             'string1',
                                             'PROFILE.INI');
    tmpn := OraWinProfile.GetPrivateString('Section',
                                           'Entry',
                                           'Bad Entry',
                                            buffer,
                                            buffern,
                                           'PROFILE.INI');
   MESSAGE(buffer);
END;
```

You invoke the WritePrivateString and GetPrivateString functions from the OraWinProfile package. After executing the example PL / SQL code, the following two lines are appended to the PROFILE.INI file.

```
[Section]
Entry=String1
```

The identical results would have been obtained if the Windows Profile String functions, WritePrivateString and GetPrivateString, were called from outside Oracle Forms. From this example, you can see how a variety of foreign functions can be invoked in Oracle Forms from a PL / SQL interface.

Accessing the Microsoft Windows SDK from a PL/SQL Interface

Many functions from the Microsoft Windows Software Developer's Kit (SDK) are accessible from a PL/SQL interface. Creating a PL/SQL interface to a Microsoft Windows SDK function is similar to creating a PL/SQL interface to any foreign function.

The parameters and return types of the Microsoft Windows SDK function must be data types that have a PL/SQL equivalent; they cannot be complex data types that cannot be simplified. For more information on simplifying complex data types, refer to the section "Invoking a Foreign Function from a PL/SQL Interface."

A foreign function with complex parameters and return data types that cannot be simplified can only be invoked from a user exit interface. An example of a complex parameter data type that cannot be simplified is a window handle, a parameter data type recognized by many Microsoft Windows SDK functions. A window handle is a unique internal character constant that is used to refer to interface objects.

In instances where the Microsoft Windows SDK function requires a window handle, you cannot invoke the function from a PL/SQL interface. Instead, you should use a user exit interface and the Window_Handle property to provide the necessary parameters. Window_Handle is a property of Oracle Forms items and windows.

In cases where the Microsoft Windows SDK function call is typical of a C language function call without complex data types, you can invoke the function from a PL/SQL interface. Treat the Microsoft Windows SDK function like any other foreign function: during initialization, load the library and register the function name, return type, and parameters. After the creation of a PL/SQL interface to the Microsoft Windows SDK function, you can invoke the Windows SDK function from Oracle Forms using PL/SQL.

CHAPTER

14 Oracle Open Client Adapter for ODBC

The Oracle Open Client Adapter for ODBC (OCA) allows Developer / 2000 tools on MS Windows to communicate with ODBC data sources through ODBC drivers.

This chapter covers the following topics:

About OCA and ODBC

Using the Oracle Open Client Adapter, an application can access different database management systems in one consistent manner. This allows an application developer to develop, compile, and ship an application without targeting a specific DBMS.

The Oracle Open Client Adapter is level 1 ODBC–compliant. The Oracle Open Client Adapter also supports, on a driver–by–driver basis, some level 2 ODBC functionality.

Oracle Forms supports both Designer and Runform ODBC datasource connections. For information about restrictions when running Oracle Forms against an ODBC datasource, see ”Setting Up Applications to run with OCA” and ”Oracle Open Client Adapter Restrictions” later in this chapter.

Note: Currently, Oracle supports only Microsoft SQL Server, Microsoft Access, and Rdb.

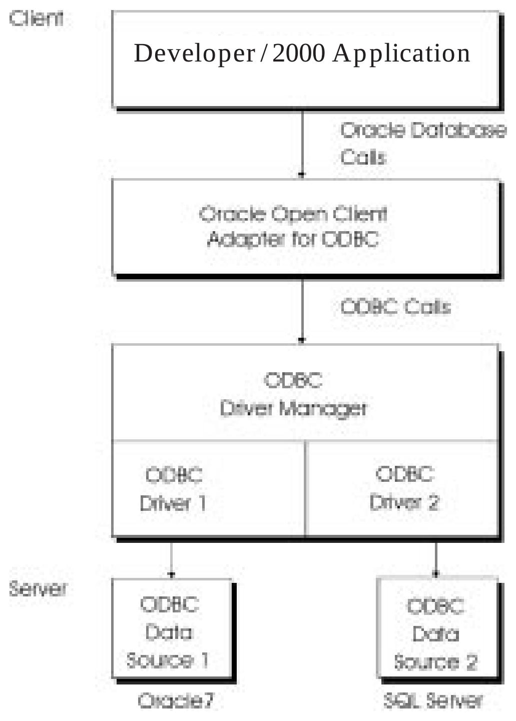
The Oracle Open Client Adapter for ODBC provides the following functionality:

- Supports both read–only and read–write access to ORACLE and non–ORACLE databases that provide ODBC drivers.
- Supports standard SQL features, as described in the ODBC SQL grammar.
- Supports Developer / 2000 tools.
- Represents ODBC database dictionary information through the standard ORACLE data dictionary views.
- ODBC / ORACLE data type conversions support most ORACLE datatypes.
- National Language Support
  - Character set translation between server side and client side character sets.
  - ASCII and EBCDIC.
  - Major 8–bit character sets supported in database object name.
- Unlike Open Gateway, no supporting Oracle kernel is required for Data Dictionary support.
- Full read / write capability limited only by the ODBC backend.
- Allows SQL pass through and supports native SQL.

Oracle Open Client Adapter Architecture

Your ODBC configuration consists of the following:

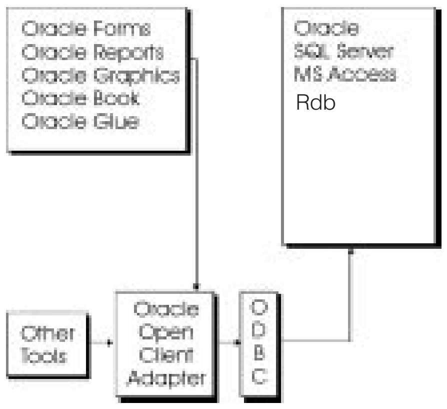
Developer / 2000 Application	Performs processing and calls ODBC functions to submit SQL statements and retrieve results.
Oracle Open Client Adapter	Translates ORACLE database calls to ODBC calls.
Driver Manager	Loads drivers on behalf of an application.
Driver	Processes ODBC function calls, submits SQL requests to a specific data source, and returns results to the application. If necessary, the driver modifies an application's request so that the request conforms to syntax supported by the target DBMS.
Data Source	Consists of the data the user wants to access and its associated operating system, DBMS, and network platform (if any) used to access the DBMS.



Oracle Open Client Adapter Communications

The following drawing shows the different data sources with which the Oracle Open Client Adapter can communicate.

Note: Currently, Oracle only supports Microsoft SQL Server, Microsoft Access, and Rdb.



Using Oracle Open Client Adapter with Oracle Forms

When running against an ODBC data source, Oracle Forms behaves as it normally behaves when running against ORACLE, with a few exceptions.

To interact with an ODBC data source via the Oracle Open Client Adapter, an Oracle Forms application must:

- connect to an ODBC data source

After connecting to an ODBC data source, an Oracle Forms application can:

- process and execute one or more SQL statements
- end each transaction by committing it or rolling it back
- terminate the connection when the application has finished interacting with the data source

For information on restrictions when running Oracle Forms against ODBC datasources, refer to ”Oracle Open Client Adapter Restrictions” later in this chapter. For information on setting up an Oracle Forms application to run against ODBC datasources, refer to ”Setting Up Applications to Run with Oracle Open Client Adapter” later in this chapter.

Establishing an ODBC Connection

To connect to an ODBC data source, the application or the user must provide (within a logon string) a user ID, a password, and the name of the data source.

Syntax: `userid/password@odbc:odbc_datasource_name`

where:

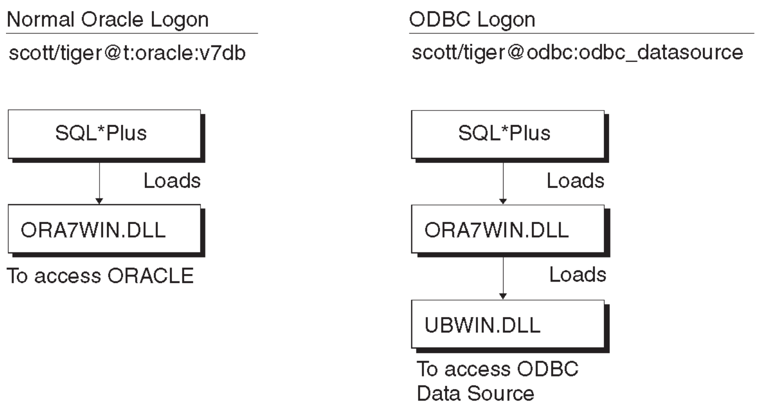
`userid`	The logon ID or account name for access to the data source (optional).
`password`	The user password that allows access to the requested data source (optional).
`odbc_datasource_name`	The name of the data source being requested by the application. If the user specifies a ” *” as a wildcard, a dialog listing all available data sources will appear.

Example:

```
scott/tiger@odbc:sqlserver
```

Note: If you omit the @odbc:database_name string or provide a normal connect string, ORACLE database calls are not mapped to ODBC calls; the adapter has no effect. Your Developer / 2000 tool behaves as it normally behaves when connecting to ORACLE.

The following SQL*Plus session illustrates the above concept:



Note: Connection information for each data source is stored in the ODBC.INI file.

Executing SQL Statements

An ODBC application can submit any SQL statement supported by the target DBMS. Submitting unsupported SQL will result in a DBMS error.

An Oracle Forms application submits both default–generated and developer–written SQL. In addition, Oracle Forms can submit SELECT statements that have been constructed dynamically by operators using the QUERY WHERE feature.

If an error occurs as a result of a SQL statement, the driver takes the appropriate action and returns the error information to the application. Error messages identify where the error originated, either in an ODBC component or at the data source.

The Oracle Open Client Adapter can execute and process these SQL statements with a few exceptions. For more information on ODBC SQL restrictions, refer to the section, ”Oracle Open Client Adapter Restrictions,” later in this chapter.

Note that ODBC defines a standard syntax for SQL statements. If an application submits a SQL statement that does not use the ODBC syntax, the driver passes it directly to the target DBMS unmodified.

Terminating Transactions and Connections

Exiting Runform and any called forms that were invoked by the CALL_FORM built–in subprogram terminates the Oracle Open Client Adapter connection to the data source. Terminating an ODBC connection frees all resources associated with the connection and allows the application to reconnect to the same data source or to a different data source.

Setting Up Applications to Run with OCA

When you design a form to run with Oracle Open Client Adapter, you must set the following properties in Oracle Forms:

- Column Security block property
- Cursor Mode form module property
- Key Mode block property
- Locking Mode block property
- Primary Key block and item property
- Savepoint Mode form property

Setting these properties configures a form to run against an ODBC data source.

Note: These properties (except Primary Key) can be set manually or by inheritance via two OCA–specific property classes, OCA_MODULE_PROPERTY and OCA_BLOCK_PROPERTY. These property class objects can be copied from the demo form OCA_PROP.FMB. OCA_PROP.FMB is located at \ORACLE_HOME\OCA\ODBC10\DEMO.

To specify a primary key, you must select the appropriate item and then define its Primary Key property.

For more information about property classes, refer to Chapter 4, "Setting Object Properties," in the Oracle Forms Developer's Guide.

Property	*Required Setting*
Column Security (block)	False By default, the Column Security property is set to False. Oracle Forms does not enforce column update privileges. Rather, the database enforces the privileges when Oracle Forms attempts to update the database. Do **not** set this property to True. The Column Security property is an ORACLE–specific property which requires the ORACLE data dictionary views: ALL_OBJECTS, TABLE_PRIVILEGES, ALL_SYNONYMS, and COLUMN_PRIVILEGES.

Cursor Mode (form)	Open Unlike ORACLE, some non–ORACLE databases do not allow the database state of cursors to be maintained across transactions. Setting Cursor Mode to Close specifies that cursors are closed at commit time by the data source.
Key Mode (block)	Updateable The Key Mode block property determines how Oracle Forms uniquely identifies rows in the database. ORACLE uses unique ROWID values to identify each row. Non–Oracle databases do not include the ROWID construct, but instead rely solely on unique primary key values to identify unique rows.
Locking Mode (block)	Delayed Specifies when Oracle Forms should attempt to obtain database locks on rows that correspond to queried records in the form. By default, Oracle Forms attempts to lock the corresponding row immediately after an operator modifies an item value in a queried record. When you set Locking Mode to Delayed, the record is locked only while the transaction is being posted to the database, not while the operator is editing the record.
Primary Key (block / item)	True Setting the Primary Key property to True for a block and item indicates that the item is a base table item in a base table block and that it corresponds to a primary key column in the base table. Oracle Forms requires values in primary key items to be unique.
Savepoint Mode (form)	False Setting Savepoint Mode to False specifies that Oracle Forms should not issue savepoints during a session.

OCA Support by Datasource

Functionality	*Oracle*	*SQL Server*	*Rdb*	*MS Access*
Data Dictionary	Yes	Yes	Yes	No
Data Conversions	Implicit	Explicit	Implicit	Explicit
Case Identifier	Maps to UC	Either	Maps to UC	Either
Security Features	Yes	Yes	No (13)	No
Column / Table Name Syntax	No Spaces	No Spaces	No Spaces	Spaces (6)
Key Mode	All Modes	PK Modes (1)	PK Modes (1)	PK Modes (1)
Cursor Mode	All Modes	Close	Close	Close
Savepoint Mode	All Modes	Off	Off	Off
Locking Mode	All Modes	Delayed	Delayed	None (9)
Non–Transactional DBMS Locking	All Modes	Delayed	All Modes	None (9)
Commit Processing	All Modes	All Modes	All Modes	Auto–Commit
Rollback Error	All Modes	(2)	All Modes	(2)
PL / SQL Supported Datatypes	All	All (10)(14)	All(14)	All (10)(14)
Supported State-ments	All	(3)	(3)	(3)
Cursor Manage-ment	All	(4)	(4)	(4)
Functions	All	(7)	(11)	(8)
Exceptions	Yes	Yes	Yes	Yes
Stored Procedures	Yes	No (5)	No	No
Table / View Loca-tion	All Modes (use <DB>)	Local DB	Local DB (12)	Local DB

(1) Either Updateable or Non–Updateable modes.

(2) Client must override the ON–ROLLBACK trigger to avoid a SQL rollback error.

(3) DECLARE, BEGIN, END, COMMIT WORK, ROLLBACK, OPEN, CURSOR, FETCH, CLOSE, GOTO, SELECT INTO, UPDATE, INSERT, DELETE, NULL .

(4) DECLARE, OPEN, CLOSE, explicit / implicit attributes (%FOUND, %NOTFOUND, %ROWCOUNT, %ISOPEN), cursor in loop constructs, and FOR UPDATE OF...WHERE CURRENT OF.

(5) Refer to the section, “SQL Server Restrictions” later in this chapter.

(6) In Microsoft Access, names can be up to 64 characters long and can include any combination of letters, numbers, spaces and special characters except a period (.), an exclamation mark (!), brackets ([]), leading spaces, or control characters (ASCII 0 through 32).

(7) SUM, AVG, MAX, MIN, COUNT. ABS, FLOOR, SIGN, SQRT, ROUND, ASCII, LOWER, UPPER, LTRIM, RTRIM, and SOUNDEX.

(8) SUM, AVG, MAX, MIN, COUNT, LTRIM, and RTRIM.

(9) Either mode should work. Microsoft Access does not support locking mechanisms through SQL. The only possible modes ’read only’ and ’exclusive’ are specified when the database is opened in the ’Open Database’ dialog box or ’ODBC Access Setup dialog’ box. Multiuser locking modes such as ’No Locks’, ’All Records’, and ’Edited Record’ are specified on a per–table basis, and there is no support to do this using SQL at runtime on an ODBC connection.

(10) Native datatypes are usable through PL/SQL, but mapping or conversion problems may exist, as well as certain restrictions on operations and length (e.g., MEMO (32k max) should map to LONG). **Note:** With PL/SQL(Version1.1.43.x.x), you cannot do select <memo col> into <a long var> from <tabname> because PL/SQL reports improper INTO var type. Also for OLE (128M max) Access datatypes, the same restriction applies; LONG RAW is limited to 32k.

(11) SUM, AVG, MAX, MIN, COUNT, UPPER, and LOWER. Rdb 6.x supports external functions. Users can execute external functions in SQL statements.

(12) If Rdb is being used with multi–schema naming, you must use the MULTISCHEMA IS OFF syntax to attach a file. For example, ATTACH ’FILENAME DISK$VDEV9:[ORACLE]RDBTEST.Rdb MULTISCHEMA IS OFF’.

If a table is created in a different schema with the same name, Rdb will uniquely identify each schema object by appending numeric digits. For more information, refer to your Rdb documentation.

(13) Rdb supports the GRANT and REVOKE commands.

Rdb does not support roles, privileges, or users. As a result, Developer/2000 tools cannot use this information in RdbVMS$PRIVILEGES for security features in this release.

(14) ODBC requires that the buffer for character data include space for the null termination byte. When selecting a character column into a PL/SQL variable, the PL/SQL buffer must have an additional byte to

store the null terminator, else data truncation occurs. For the same reason CURSOR%ROWTYPE will not work if the row contains columns of type CHAR or VARCHAR.

Oracle Open Client Adapter Restrictions

When developing applications, ODBC application developers should be aware of the restrictions and limitations imposed by ODBC, as well as those imposed by the target database. In general, these restrictions involve developer–written SQL (SQL, PL / SQL, and the Default Where clause).

Generic Restrictions

This section describes the generic restrictions which apply to each Developer / 2000 application when running with Oracle Open Client Adapter.

Case Sensitivity ORACLE converts names of all database objects to uppercase. Some databases, like SQL Server, are case sensitive. For example, ABC, abc, AbC are treated as different identifiers.

Suggestion: Install SQL Server with the Case Insensitive option.

Scalar Functions Most databases do not support the set of scalar functions provided by ORACLE, such as TO_CHAR() and DECODE(). Since PL / SQL only parses for ORACLE functions, the only functions that will work with non–Oracle databases are the SQL common to ORACLE and non–Oracle databases.

For a complete listing of Oracle scalar functions, refer to the *SQL Language Reference Manual*.

Suggestion: In general, DECODE() functionality can be replaced by the use of one or more relational views which contain joins that perform the scalar column value mapping normally performed by the DECODE(). It may also be possible to replace simple DECODE() functions with arithmetic functions, which may be more efficient than joins.

Data Dictionary Objects Oracle tools assume certain specific data dictionary views with specific column names and types. In particular, Oracle Forms requires the following views: ALL_OBJECTS, ALL_TABLES, ALL_VIEWS, ALL_TAB_COLUMNS, ALL_USERS, DUAL, TABLE_PRIVILEGES, ALL_SYNONYMS, and COLUMN_PRIVILEGES.

These ORACLE–specific views do not exist in non–ORACLE databases.

Suggestion: Currently, for SQL Server and Rdb, Oracle provides a limited set of views to map the native dictionary objects into the necessary ORACLE data dictionary views. When using these views, users should restrict their usage to the columns provided.

Note: The Tables / Columns feature in the Oracle Forms Designer will not work unless these views are present in the target data dictionary.

Column/Table Name Oracle Forms does not support spaces as part of a column or table name. For example, ”EMPNO” and ”EMP_NO” are valid column names, but ”EMP NO” is not. Column and table name identifiers must not include spaces.

List of Values Oracle Open Client Adapter does not support V2 style LOVs.

Comments within SQL Statements Although SQL Server supports embedded SQL /*hint*/ style comments, not all ODBC databases drivers have the same support.

Suggestion: Avoid embedding optimizer hint comments within SQL statements.

NOWAIT Clause for Locks Oracle tools allow the NOWAIT clause to be appended to SELECT statements. The NOWAIT option tells ORACLE not to wait if the table has been locked by another user.

For example, when Oracle Forms updates a record, it issues the following statement in order to lock the record for update.:

```
SELECT ... FOR UPDATE ... NOWAIT
```

If Oracle Forms cannot acquire the lock, the NOWAIT clause causes the database to issue an error stating that the record has been locked by another user. Control is then immediately returned to your application so that it can do other work before trying again to acquire the lock.

Suggestion: The NOWAIT option is ORACLE–specific. Do not use the NOWAIT parameter with the ENTER_QUERY or EXECUTE_QUERY built–ins.

When the NOWAIT parameter is not specified, Oracle Forms behaves as if it is in delayed locking mode; Oracle Forms waits to lock the corresponding row in the database until the transaction is about to be committed. With this setting, the record is locked only while the transaction is being posted to the database, not while the operator is editing the record.

If another user has a lock and, as a result, Oracle Forms is unable to obtain a requested lock, Oracle Forms will request (via the lock dialog) whether it should retry. The operator can respond Yes and wait for Oracle Forms to acquire the lock or terminate the locking procedure by pressing CTRL–C (or its equivalent).

Sequences A sequence is an Oracle–specific database object that generates sequential numbers.

An Oracle Forms application can reference sequence values for default item values by setting the Default item property to SEQUENCE.my_seq.NEXTVAL.

Suggestion**:**Avoid the use of sequences. Most databases do not support this functionality.

Commit/Rollback Some databases allow only one cursor per connection. As a result, any additional cursors must be allocated through separate connections.

Oracle Forms controls master–detail relations through multiple cursors. Since SQL Server only allows one active cursor per connection, another connection must be established for each detail. Because each cursor has an individual connection, committing a master–detail block application requires coordinating the various cursors. Consequently, a problem could arise if only a subset of the multiple required connections commit their work and a form could be inconsistently updated if a remaining subset of the connections fail to commit or roll back.

Suggestion: Use a driver that coordinates two–phase commits between connections.

Selecting LONG Columns Some databases do not permit revisiting columns to fetch LONG data piecemeal once data for the next column has been fetched.

For example, assume that you issue the following statement:

```
SELECT A, B, C FROM TEST
```

If column B is a LONG, data for column B cannot be fetched piecemeal, once data for column C is fetched. In order to fetch data piecemeal from a LONG, the query must be re–arranged so that the LONG column appears last.

```
SELECT A, C, B FROM TEST
```

The same restriction prohibits the inclusion of more than one LONG column in a query executed against such databases.

Suggestion: Modify your SELECT statements so that the LONG column appears last.

Programmatic Offset–Based Fetching of LONG Columns Incremental offset–based fetching of LONG field data is not supported by ODBC. The ODBC specification permits only sequential fetching of data.

Unsupported PL/SQL Statements The Oracle Open Client Adapter does not support the following PL / SQL statements:

ROLLBACK TO
SAVEPOINT
LOCK TABLE
SET TRANSACTION

Driver–Specific Restrictions

This section describes driver–specific restrictions which apply when running with Oracle Open Client Adapter.

SQL Server Restrictions

This section describes the Microsoft SQL Server restrictions which apply when running with Oracle Open Client Adapter. For information about additional restrictions, refer to your MS SQL Server Driver documentation.

GROUP BY, HAVING, and UNION Clauses The Microsoft ODBC Driver for SQL Server (Version 1.02.3231 and later) does not support the GROUP BY, HAVING, or UNION clause correctly. GROUP BY, HAVING, or UNION SQL statements that do not contain a WHERE clause result in incorrect statement modification by the Microsoft SQL Server ODBC driver for SQLDescribeCol operations.

Suggestion: When using a GROUP BY, HAVING, or UNION clause in a SQL statement without a WHERE clause, include a dummy WHERE clause with the predicate WHERE 1 = 1.

SELECT FOR UPDATE If you are using a driver that does not support the FOR UPDATE in SELECT statements, set the Locking Mode property to Delayed and override (nullify) any default On–Lock triggers. The Microsoft SQL Server driver does not support the FOR UPDATE clause in SELECT statements.

DDL Statements The Transaction option is turned on for SQL Server in Oracle Open Client Adapter. When the Transaction option is on, Oracle Open Client Adapter restricts the usage of the following DDL statements:

CREATE DATABASE, CREATE TABLE, CREATE INDEX, CREATE VIEW, all DROP statements, SELECT INTO (because it creates a table),GRANT, REVOKE, ALTER DATABASE, TRUNCATE TABLE, UPDATE STATISTICS, RECONFIGURE, LOAD DATABASE, LOAD TRANSACTION, and DISK INIT.

When issuing these DDL statements (via the FORMS_DDL built–in subprogram), you must not issue any subsequent DML statements until you commit or roll back your transaction.

For example, issuing the following will result in an error:

```
SELECT ... CREATE ...
```

However, if you add a rollback, the statement will succeed:

```
SELECT ... ROLLBACK TRANSACTION ... CREATE ...
```

ORDER BY in SQL Server Within ORDER BY clauses, SQL Server permits positional references to columns only if the columns are explicitly listed.

For example,

```
SELECT * FROM TABLE .. ORDER BY <column_number>
```

will raise a SQL Server error.

Suggestion: Replace the ”*” with explicit column names, or substitute the <column_number>with a <column_identifier>.

In addition, SQL Server does not permit duplicate column references in the ORDER BY clause, such as:

```
SELECT C1, C2, C3 FROM TABLE .. ORDER BY C2, 2
```

FLOAT Datatype An Oracle Forms item defined as a NUMBER cannot accept values greater than 2,147,483,647 if the corresponding SQL Server column is a FLOAT datatype.

In addition, entering a floating point number as an integer may cause an error. For example, entering 12345 into a float field will result in an error. However, entering the floating point number as a decimal (e.g.,12345.0) is valid as it indicates to the Oracle Open Client Adapter that the number is a floating point number.

Suggestion: Floating point numbers should be entered as decimals.

Oracle PL/SQL % Predicate Because SQL Server does not support the TO_CHAR datatype, the Oracle PL / SQL % predicate cannot be used at runtime to query date or numeric datatypes.

Stored Procedure Support Currently, SQL Server stored procedures are not supported by Oracle Forms or PL / SQL.

Suggestion: To use a stored procedure which returns a result set, your stored procedure must deposit its results in a temporary table. Oracle Forms can then select from the temporary table. For more information, refer to the sprocbild.sql and sprocrun.sql example scripts.

Global Variables SQL Server global variables cannot be used with Oracle Forms or PL/SQL.

Transact SQL Statements Use of the following Transact SQL statements is restricted:

- The PRINT statement cannot be used with PL/SQL, although I/O routines can be called from within user exits.
- The WAIT FOR {DELAY 'time' | TIME 'time' | ERROREXIT |PROCESSEXIT | MIRROR EXIT statement cannot be used with PL/SQL.
- The FOR UPDATE ... WHERE CURRENT OF statement does not work with SQL Server.

SQL Functions As a rule, SQL functions that exist in ORACLE and SQL Server work with OCA. This includes the following:

- aggregates or SQL functions: ALL, DISTINCT, SUM, AVG, MAX, MIN, and COUNT
- date functions: none
- mathematical functions: ABS, FLOOR, SIGN, SQRT, and ROUND
- string functions: ASCII, LOWER, UPPER, LTRIM, RTRIM, and SOUNDEX
- system functions: none
- text image functions: none
- type conversion: none

Multiple Inserts in a PL/SQL Block If multiple insert statements on a table refer to the same database page for the table in a PL/SQL block, the application will deadlock. This situation exists because the ODBC driver executes the insert statements through separate connections, and they lock each other out.

Suggestion: To perform multiple commits in the same PL/SQL block, you could commit between inserts, or execute them in a PL/SQL LOOP, or put the insert statements in a stored procedure and pass the values as arguments to the procedure.

Microsoft Access Restrictions

This section describes the MS Access restrictions which apply when running with Oracle Open Client Adapter. For information about additional restrictions, refer to your MS Access Driver documentation.

SQL Statements Access SQL supports mainly DML commands. Access specific SQL statements such as DISTINCTROW, the IN clause, INNER JOIN, LEFT JOIN, RIGHT JOIN, PARAMETERS, TRANSFORM, WITH OWNERACCESS OPTION, and PROCEDURE can be used only with the FORMS_DDL built–in subprogram.

Datatypes All native datatypes are supported with the exception that memo and OLE fields cannot be mapped to LONG and LONG RAW columns in the current version of PL/SQL supported by Oracle Forms. However, MEMO and OLE columns can be retrieved by mapping them to CHAR and RAW datatype fields in PL/SQL, though updating those columns will not be possible through CHAR/RAW datatypes until OCA version 2.0 becomes available.

Locking Applications cannot perform any locking on Access database tables unless programmed through Access Basic. In order to perform locking on an Access database table, the operator must specify the appropriate locking mode in the ODBC setup panel when selecting the database. Locking modes in Oracle Forms have no effect on Access.

PL/SQL All PL/SQL programming constructs and datatypes can be used with Access with the usual limitations when interfacing with non–Oracle databases.

Rdb Restrictions

This section describes the Rdb restrictions which apply when running with Oracle Open Client Adapter. For information about additional restrictions, refer to your Rdb Driver documentation.

SQL Statements Rdb SQL does not support the following ORACLE–specific features: sequences, synonyms, database links, security roles, pseudo–columns such as ROWID and ROWNUM, SQL functions, row comparisons, set difference operators (MINUS and INTERSECT), recursive queries, order by expressions, savepoints, and disabled constraints.

Rdb–specific SQL statements can be executed using the FORMS_DDL built–in function.

Datatypes Rdb and ORACLE datatypes can be mapped as follows:

Rdb	*ORACLE*
CHAR	CHAR
VARCHAR	VARCHAR
TINYINT	NUMBER
SMALLINT	NUMBER (SMALLINT)
INTEGER	NUMBER (INTEGER)
BIGINT	NUMBER
REAL	NUMBER (REAL)
FLOAT	NUMBER (FLOAT)
DATE	DATE
LIST OF BYTE VARYING	RAW, LONG RAW

PL/SQL All PL / SQL programming constructs and datatypes can be used with Rdb.

Note: Currently, a stored procedure in Rdb cannot be invoked from PL / SQL unless it is parameterless or contains only input parameters. Rdb stored procedures that do not contain parameters or contain only input parameters can be called from a PL / SQL procedure by using the FORMS_DDL built–in function.

Using UBT

UBT is an interactive command driver for testing and exercising the Oracle Open Client Adapter. UBT enables you to issue SQL commands within the interpreter or through command files. You can perform data migration by using the database table–to–table COPY command.

SQL Command Syntax

You can divide your SQL commands into separate lines at any point, as long as individual words are not split between lines.

SQL commands are terminated in one of two ways:

- with a semicolon (;). This indicates that you want to run the command. Type the semicolon at the end of the last line of the command.
- with a blank line. The command is ignored and the prompt appears.

UBT Command Syntax

You may continue a long UBT command by typing a hyphen (–) at the end of the line and pressing [Return]. If you wish, you may type a space before typing the hyphen. The line number is displayed for the next line.

Do not end a UBT command with a semicolon. When you finish entering the command, just press [Return].

Note: Any non–UBT command is interpreted as a SQL command and will be passed to the underlying database.

UBT Commands

The following section provides a description of each UBT command.

CONNECT

Description: Connects to a given username and database.

Syntax:

```
CONNECT username[/password][@database_specification]
```

Parameters:

`username [/password]`	Represents the username and password with which you wish to connect to the database.
`database_ specification`	Consists of a SQL*Net or ODBC connection string.

COPY

Description: Copies the data from a query to a table in a local or remote database.

Syntax:

```
COPY {FROM username[/password]@database_specification |

TO username[/password]@database_specification |
FROM username[/password]@database_specification TO
username[/password]@database_specification}
{APPEND/CREATE/INSERT/REPLACE} destination_table
[(column, column, column ...)] USING query
```

Note: | means OR

Parameters:

`username [/password]`	Represents the username / password you wish to COPY FROM and TO. In the FROM clause, username / password identifies the source of the data; in the TO clause, username / password identifies the destination.
`database_ specification`	Consists of a SQL*Net or ODBC connection string. In the FROM clause, database_specification represents the database at the source; in the TO clause, database_specification represents the database at the destination. The exact syntax for SQL*Net depends on the communications protocol your ORACLE installation uses. The syntax for an ODBC connection is ”ODBC:data_source” where the

	ODBC keyword is followed by a colon and the name of the installed data source.
`destination_table`	Represents the table you want to create or to which you want to add data.
`(column, column, column, ...)`	Specifies the names of the columns in destination_table. If you specify columns, the number of columns must equal the number of columns selected by the query. If you do not specify any columns, the copied columns will have the same names in the destination table as they had in the source, if COPY creates destination_table.
`USING query`	Specifies a SQL SELECT statement that determines which rows and columns COPY copies.
`FROM username [/password] database_ specification`	Specifies the username, password, and database that contains the data to be copied. If you omit the FROM clause, the source defaults to the database UBT is connected to (the database that other commands address). You must include a FROM clause to specify a source database other than the default.
`TO username [/password] database_ specification`	Specifies the database containing the destination table. If you omit the TO clause, the destination defaults to the database UBT is connected to. You must include a TO clause to specify a destination database other than the default.
`APPEND`	Inserts the rows from query into destination_table if the table exists. If destination_table does not exist, COPY creates it.
`CREATE`	Inserts the rows from query into destination_table after creating the table first. If destination_table already exists, COPY returns an error.
`INSERT`	Inserts the rows from query into destination_table if the table exists.
`REPLACE`	Replaces destination_table and its contents with the rows from query. If destination_table does not exist, COPY creates it. If destination_table exists, COPY drops the existing table and replaces it with a table containing the copied data.

Example: The following command copies the entire EMP table to a table named WESTEMP. Note that the tables are located in two different databases. If WESTEMP already exists, both the table and its contents are replaced. The columns in WESTEMP have the same names as the columns in the source table, EMP.

```
COPY FROM scott/tiger@ODBC:INFO TO jack/smith@ODBC:HQ –
REPLACE westemp –
USING SELECT * FROM EMP
```

The following command copies selected records from EMP to the database to which UBT is connected. Table SALESMEN is created during the copy. UBT copes only the columns EMPNO and ENAME and at the destination names them EMPNO and SALESMAN.

```
COPY FROM scott/tiger@ODBC:INFO –
CREATE salesmen (empno,salesman) –
USING SELECT empno, ename FROM emp –
WHERE job='SALESMAN'
```

@

Description: Runs the specified UBT command file.

Syntax: `@file_name.ext`

DISCONNECT

Description: Commits pending changes to the database and logs the user off the database, but does not exit UBT.

Syntax: `DISCONNECT`

QUIT/EXIT

Description: Commits all pending database changes, terminates UBT, and returns control to the operating system.

Syntax: `QUIT | EXIT`

SET AUTOCOMMIT

Description: Toggles Autocommit to On or Off. Some databases allow certain types of SQL statements only when Autocommit is On.

For example, SQL Server only allows DDL statements when Autocommit is On.

Syntax: `SET AUTOCOMMIT {ON | OFF}`

SPOOL

Description: Stores query results in an operating system file.

Syntax: `SPOOL [file_name | OFF]`

Parameters:

`file_name`	Represents the name for the file to which you wish to spool.
`OFF`	Stops spooling

Example: To record your displayed output in a file named DIARY.OUT, enter:

```
SPOOL DIARY.OUT
```

To stop spooling, type:

```
SPOOL OFF
```

APPENDIX

A Using Oracle Terminal

This appendix explains how to use Oracle Terminal to redefine the mapping of Oracle Forms runtime keys to physical device keys. This appendix also explains how to set the logical attributes of interface objects by using Oracle Terminal to modify Oracle Forms resource files. The main sections in this chapter are:

About Oracle Forms Resource Files

Resources are collections of related data. An application built with Oracle Terminal support includes resources that describe its behavior and appearance. Specifically, Oracle Forms resource files define design time and runtime logical attributes and key bindings for Oracle Forms.

Using Oracle Terminal to edit the Oracle Forms resource file for your platform, you can:

- associate a key sequence with an application function by defining a key binding
- associate a key sequence with an Oracle Forms key trigger by defining a key binding
- modify the visual appearance (face, pattern, color, etc.) of an Oracle Forms–specific element

Note: Resource file names are platform–specific. On MS Windows, the look and feel of an Oracle Forms application is defined by the resource file, FMRUSW.RES. For more information about platform specific resource files, refer to the Oracle Forms documentation for your operating system.

Using Oracle Terminal to Remap Oracle Forms Key Mappings

Resource files associate application functions with specific keys, using what are known as key bindings.

A key binding connects a key to an application function. When you bind a key to a function, the program performs that function when you type that keystroke.

By defining key bindings, you can integrate a variety of keyboards to make an application feel similar on each of them.

In some instances, you may want to define your own key mapping for a specific Oracle Forms function. For example, on MS Windows, the default key mapping for [Show Keys] is Control+F1. Using Oracle Terminal, you could remap the [Show Keys] function to Control+F2 or to some other desired key mapping.

The key bindings that you create using Oracle Terminal automatically appear in the Show Keys window when the operator invokes the Show Keys function.

Note: On some platforms, the [Help] key binding cannot be remapped. For example, on MS Windows, [Help] is mapped to the F1 key and cannot be remapped.

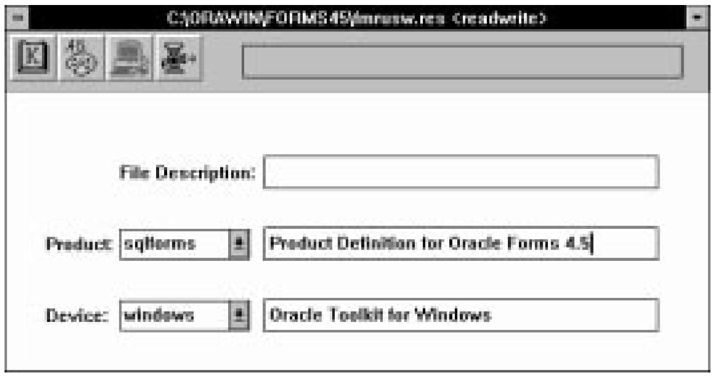
To remap an Oracle Forms key binding:

1. Start Oracle Terminal.

   The Open dialog appears.

2. Open the appropriate Oracle Forms resource file (on MS Windows, open FMRUSW.RES). For more information about platform–specific resource files, refer to the Oracle Forms documentation for your operating system.

   Oracle Terminal appears.



C:\ORAWIN\FORMS45\fmrusw.res <readwrite>

File Description:

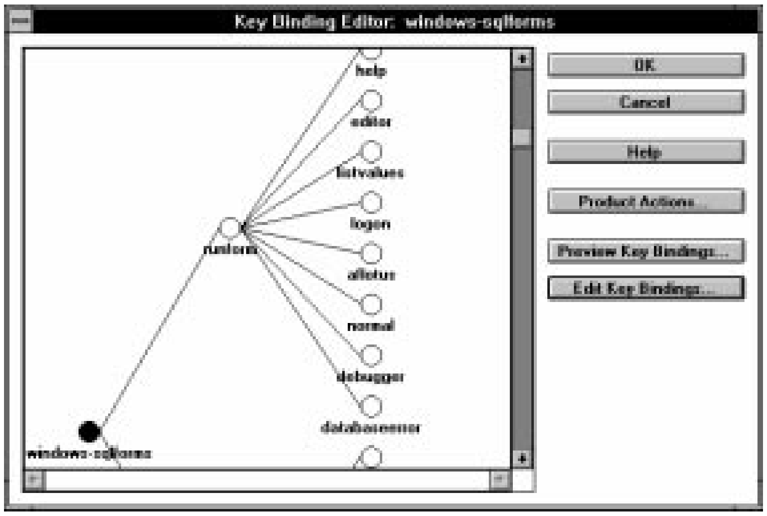
Product: sqlforms Product Definition for Oracle Forms 4.5

Device: windows Oracle Toolkit for Windows

Note: Before modifying an Oracle Forms resource file, you should create a backup copy.

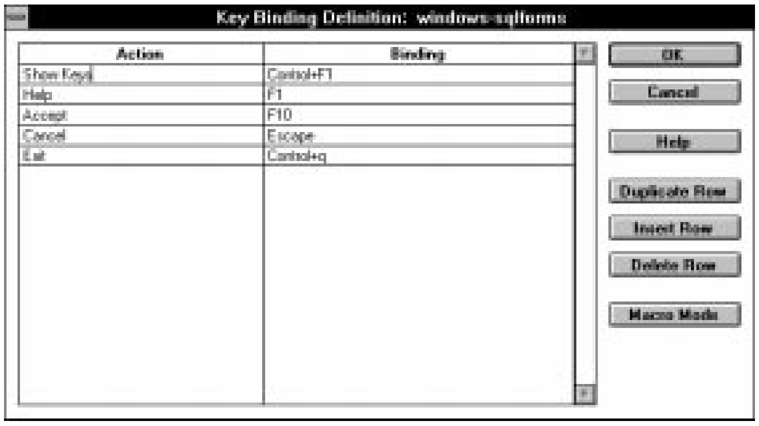
3. Choose Functions–>Edit Keys.

   The Key Binding Editor appears and lists Oracle Forms key bindings (Designer and Runform) by category: windows–sqlforms, runform, help, editor, listvalues, logon, aflotus, normal, debugger, databaseerror, etc.



   For more information about Runform key binding groups, refer to "Oracle Forms Runform Key Bindings" later in this chapter.

4. Double–click on a category and modify the desired "Action" and "Binding."



Action	Binding
Show Keys	Control+F1
Help	F1
Accept	F10
Cancel	Escape
Exit	Control+q

For example, to modify the key mapping for [Show Keys], double–click on “windows–sqlforms,” specify the desired key binding for Show Keys, then click OK.

Note: Key bindings must be unique. When you define a key binding, verify it against each bind category to ensure uniqueness.

In addition, there are functions which have multiple key bindings.

For example, [Accept] is mapped to both F10 and the Return key, depending on the context. As a result, in order to consistently bind a sequence of keys with a function, you must modify the key binding for each instance.

5. Choose File–>Save to save your modifications.
6. Choose Functions–>Generate to generate your modifications and to create a new resource file, which incorporates the changes you made to the key binding.

   An alert appears, informing you that your resource file has been successfully generated. Accept the alert.

   Note: If you receive an error message instead, consult your *Oracle Terminal User’s Guide*.
7. Choose File–>Save to save your changes to the new resource file.
8. Run Oracle Forms to use your new key binding.

Using Oracle Terminal to Define Key Bindings for Key–Fn and Key–Other Triggers

When you create a Key–Fn or Key–Other Triggers, you must use Oracle Terminal to associate a key with your Key–Fn or Key–Other function. Creating a key binding for your trigger allows it to fire when the operator presses the key sequence associated with your trigger.

1. In the Oracle Forms Designer, create your Key–Fn or Key–Other trigger.
2. Using Oracle Terminal, open the appropriate Oracle Forms resource file (on MS Windows, open FMRUSW.RES). For more information about platform specific resource files, refer to the Oracle Forms documentation for your operating system.

   Oracle Terminal appears.
3. Choose Functions–>Edit Keys to invoke the Key Binding Editor.
4. In the Key Binding Editor, double–click on windows–sqlforms to invoke the Key Binding Definition window.
5. Click on the Insert Row button and then enter the action that corresponds to your Key–Fn or Key–Other trigger. (The chart below applies only to MS Windows.)

Key–Fn Trigger	*MSW Action*	*MSW Code*
KEY–F0	User Defined Key 0	82
KEY–F1	User Defined Key 1	83
KEY–F2	User Defined Key 2	84
KEY–F3	User Defined Key 3	85
KEY–F4	User Defined Key 4	86
KEY–F5	User Defined Key 5	87
KEY–F6	User Defined Key 6	88
KEY–F7	User Defined Key 7	89
KEY–F8	User Defined Key 8	90
KEY–F9	User Defined Key 9	91

For example, if you created a KEY–F3 trigger, enter:

```
User Defined Key 3
```

into the Action field.

6. Specify a key binding for your Key–Fn trigger, then click OK.

   For example, you could enter:

Action	*Binding*
User Defined Key 3	Control+F2

   Note: Key bindings must be unique. When you define a key binding, verify it against each bind category to ensure uniqueness.

7. Click the Product Actions button to invoke the Product Actions Editor.

8. Double–click on the sqlforms category.

   The Product Action Definition window appears.

9. Click on the Insert Row button, enter the Action, Code, and Description for your Key–Fn trigger, then click OK.

   For example, if you create a KEY–F3 trigger that invokes a help system, enter the following on MS Windows:

Action	*Code*	*Description*
User Defined Key 3	85	[Help System]

10. Dismiss the Product Action and Key Binding Editors by clicking on the OK buttons, then choose File–>Save.

11. Choose Functions–>Generate to generate your modifications and to create a new resource file, which incorporates the additions you made.

12. Choose File–>Save to save your changes to the new resource file.

13. (Optional) Within the Forms Designer, set the Show Keys property for your key trigger to True if you want the key binding for your trigger to appear in the Show Keys window.

Oracle Forms Runtime Key Bindings

The following chart lists the default Action, Code, Key bindings, and category listings for Oracle Forms Runform on **MS Windows**. Refer to this chart when creating Key–Fn or Key–Other triggers.

Action/Function	*MSW Code*	*MSW Key Binding*
windows–sqlforms		
Show Keys	1003	Control+F1
Help	1004	F1
Accept	1000	F10
Cancel	1001	Escape
Exit	1002	Control+q
Refresh	1005	
help:		
Accept	1000	Return
Up	1306	UpArrow
Down	1307	DownArrow
Scroll Up	1302	PageUp
Scroll Down	1303	PageDown
Scroll Left	1304	LeftArrow
Scroll Right	1305	RightArrow
editor:		
Clear Item	3	Control+End
Search / Replace	1200	
Search Next Occurrence	1201	
Replace Next Occurrence	1202	
Toggle Insert / Replace	1203	

Action/Function	*MSW Code*	*MSW Key Binding*
listvalues:		
Delete Backward	1303	Backspace
Pop Criteria	1301	F9
Accept	1000	Return
Page Up	1302	PageUp
Page Down	1303	PageDown
Scroll Left	1304	LeftArrow
Scroll Right	1305	RightArrow
Scroll Up	1306	UpArrow
Scroll Down	1307	DownArrow
Change Focus	1308	Tab
logon:		
Next Item	1400	
Previous Item	1401	
aflotus:		
Accept	1000	DownArrow
Accept	1000	Return
Accept	1000	F10
Cancel	1001	Escape
Exit	1002	Control+q
Previous Item	1403	LeftArrow
Previous Item	1403	Shift+Tab
Next Item	1404	RightArrow
Next Item	1404	Tab
Up	1405	UpArrow
Delete Backward	1406	Backspace
normal		
Commit	1000	
Next Item	1	Tab
Previous Item	2	Shift+Tab

Action/Function	*MSW Code*	*MSW Key Binding*
Clear Item	3	Control+u
Up	6	UpArrow
Down	7	DownArrow
Scroll Up	12	DownArrow
Scroll Down	13	DownArrow
Edit	22	Control+e
Search	24	Control+s
Toggle Insert / Replace	25	
Select	26	Return
Delete Backward	16	Backspace
Return	27	Return
List of Values	29	F9
Next Primary Key	61	Shift+F3
Clear Record	62	Shift+F4
Delete Record	63	Shift+F6
Duplicate Record	64	F4
Insert Record	65	F6
Next Set of Records	66	Control+>
Next Record	67	Shift+DownArrow
Previous Record	68	Shift+UpArrow
Clear Block	69	Shift+F5
Block Menu	70	F5
Next Block	71	Control+PageDown
Previous Block	72	Control+PageUp
Duplicate Item	73	
Clear Form	74	Shift+F7
Enter Query	76	F7
Execute Query	77	F8
Display Error	78	Shift+F1
Print	79	Shift+F8
Count Query Hits	80	Shift+F2
Update Record	81	
User Defined Key0	82	
User Defined Key1	83	

Action/Function	*MSW Code*	*MSW Key Binding*
User Defined Key2	84	
User Defined Key3	85	
User Defined Key4	86	
User Defined Key5	87	
User Defined Key6	88	
User Defined Key7	89	
User Defined Key8	90	
User Defined Key9	91	
Clear End–of–Line	92	
Debug	93	
Toggle Query Mode	94	
Redefine Username / Password	11001	Control+n
Where Display	11002	Control+w
Debug Mode	11003	Control+?
Application Menu	11004	Control+,
Previous Menu	11005	Control+Return
Main Menu	11006	Control+.
Enter>1 OS Command	11007	
Enter>2 OS Command	11008	
Show Background Menu	11009	Control+ /
Background Menu 1	1101	Control+1
Background Menu 2	11011	Control+2
Background Menu 3	11012	Control+3
Background Menu 4	11013	Control+4
Background Menu 5	11014	Control+5
Background Menu 6	11015	Control+6
Background Menu 7	11016	Control+7
Background Menu 8	11017	Control+8
Background Menu 9	11018	Control+9
Background Menu 10	11019	Control+0
Enter Application Parameters	11020	Control+F6
Enter Menu Parameters	11021	Control+F5

Action/Function	*MSW Code*	*MSW Key Binding*
Accelerator1	11022	Control+F2
Accelerator2	11023	Control+F7
Accelerator3	11024	Control+F8
Accelerator4	11025	Control+F9
Accelerator5	11026	Control+F10
Display Full Screen Menu	11032	Control+f

Using Oracle Terminal to Modify Oracle Forms Logical Attributes

In addition to setting default, custom, and named visual attributes in the Oracle Forms Designer, you can also modify the look (logical attributes) of an Oracle Forms application by modifying Oracle Forms resource files.

Oracle Forms logical attributes are organized into two groups: character mode (cm_logicals) and GUI (forms_logicals).

Note: Because of the limited number of GUI logical attributes that can be defined using Oracle Terminal, you should use the Oracle Forms Designer to create custom visual attributes for the desired GUI elements rather than using Oracle Terminal to modify an Oracle Forms resource file.

When you edit Oracle Forms resource files, you specify font and color attributes for character mode or GUI logical attributes, such as field–current, field–non–current, pushbuttoncurrent, field–queryable, etc.

About Attribute Precedence

In addition to modifying Oracle Forms resource files, there are several other methods for setting display attributes for Oracle Forms objects. You can:

- accept Oracle Forms default attributes
- define custom visual attributes
- define logical attributes using Oracle Terminal
- define display attributes using your window manager

In many cases, depending on the method chosen, you can create an attribute that overrides an attribute defined at a higher level.

When defining object attributes, consider the following rules regarding attribute precedence:

1. Window manager (lowest level)

   Window manager definitions take precedence over Oracle Terminal definitions, visual attributes defined in the Oracle Forms Designer, local environment variable definitions (ORACLE.INI, CONFIG.ORA, etc.), and default Oracle Forms attributes.

For example, on MS Windows, color attributes for buttons are handled exclusively by Windows. Any color attributes that you create and apply to buttons in the Oracle Forms Designer have no effect on buttons.

To change the color attributes of a button in a Microsoft Windows environment, use the MSW Control Panel facility.

2. Oracle Terminal definitions

   Oracle Terminal definitions take precedence over visual attributes defined in the Oracle Forms Designer, local environment variable definitions, and default Oracle Forms attributes.

   For example, assume that you create a custom visual attribute for the text items in your form so that text items appear gray. Assume also that you use Oracle Terminal to define all queryable items as cyan. When you run your form, all text items appear gray, but when you invoke Enter Query mode, text items appear cyan.

   In this example, the Oracle Terminal definition takes precedence and overrides the higher level attribute (the custom visual attribute created at design time).

3. Visual attributes created within the Oracle Forms Designer

   Designer–created visual attributes take precedence over local environment variable definitions (ORACLE.INI, CONFIG.ORA, etc.) and default Oracle Forms attributes.

   For example, assume that you create and apply a named visual attribute to the text items in your form. Assume also that you define the FORMS45_DEFAULTFONT local environment variable in CONFIG.INI so that item labels are displayed with 10 point Courier.

   At runtime, your named visual attribute takes precedence, and Oracle Forms displays all text items according to the properties defined by your named visual attribute.

4. Local environment variable definitions

   Local environment variable definitions (ORACLE.INI, CONFIG.ORA, etc.) take precedence over default Oracle Forms attributes.

For example, if you edit your ORACLE.INI file and define the default font (FORMS45_DEFAULTFONT) as 10 point Courier, Oracle Forms displays item labels as 10 point Courier.

In this example, the default font in ORACLE.INI takes precedence and overrides the default font attributes for items.

5. Oracle Forms default attributes (highest level)

   Oracle Forms applies default attributes to every object. At runtime, Oracle Forms looks up the specific color, pattern, and font characteristics for a particular object and then displays it with its default attributes.

Modifying Oracle Forms Logical Attributes

You can use the Oracle Terminal program to edit or delete the definitions of logical attributes in an existing resource file.

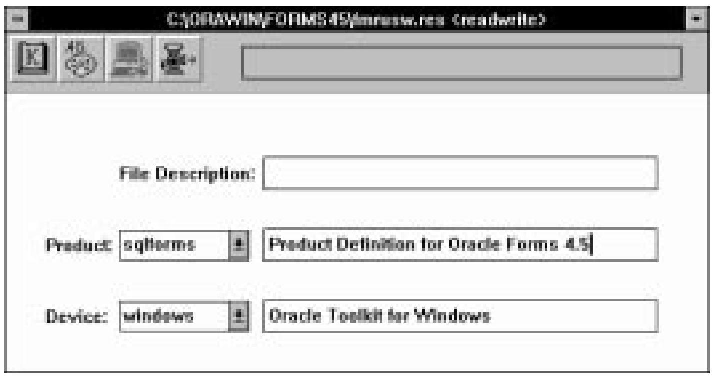
To modify an Oracle Forms logical attribute:

1. Start Oracle Terminal.

   The Open resource file dialog appears.

2. Using Oracle Terminal, open the appropriate Oracle Forms resource file (on MS Windows, open FMRUSW.RES). For more information about platform–specific resource files, refer to the Oracle Forms documentation for your operating system.

   Oracle Terminal appears.



File Description:

Product: sqlforms — Product Definition for Oracle Forms 4.5

Device: windows — Oracle Toolkit for Windows

3. Choose Functions–>Edit Attributes.

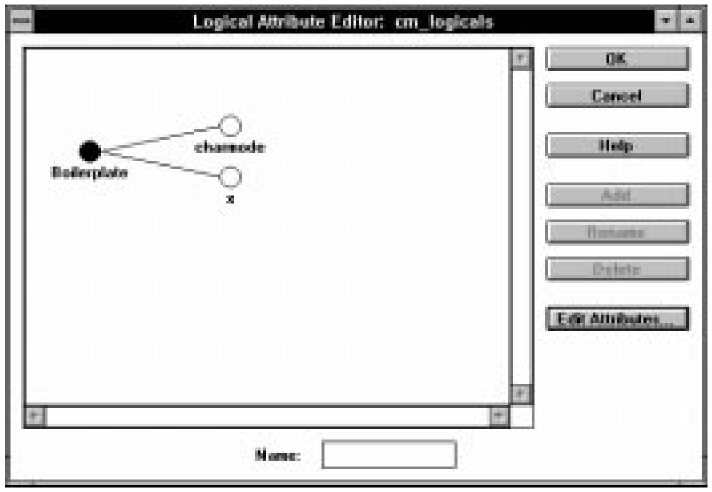
   The Oracle Terminal Attributes window appears.

4. Select cm_logicals or form_logicals from the Attribute List pop list field.

   Oracle Terminal displays the logical attributes associated with your selection.

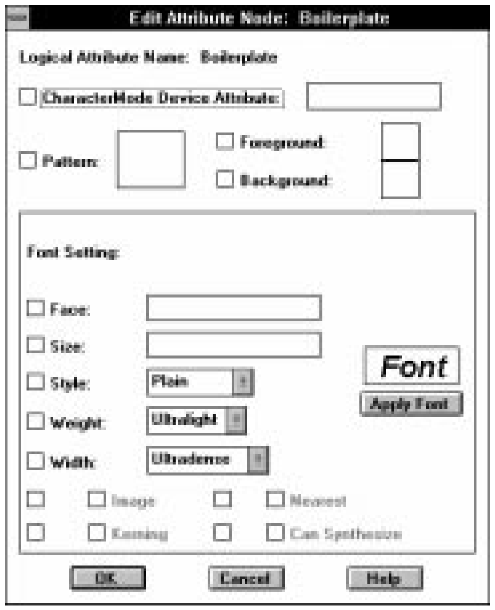
5. Select the desired attribute from the Attribute field, then click on the Edit button.

   Oracle Terminal displays the Logical Attribute Editor.



6. Select the desired node and then click on the Edit Attributes button.

   The Edit Attribute Node window appears.



7. Define the desired attributes.

   For example, you can define the pattern, color, size, style, and weight of the desired logical attribute. For more information about options in the Edit Attribute Node window, see “Edit Attribute Node Descriptions” later in this chapter.

8. After making the desired modifications, click on the Apply Font button.

9. Save your modifications by choosing File–>Save.

For more information on changing visual attributes, see the *Oracle Terminal User's Guide*.

Edit Attribute Node Options

The Edit Attribute Node window offers the following options to define the font and color elements of a logical attribute.

Font Attributes

Font Attributes include:

- Face
- Size
- Style
- Weight
- Width

At runtime, Oracle Forms passes these settings to the window manager. If the font that you specify is unavailable in the runtime operating system, Oracle Forms uses the font that most nearly matches the characteristics of the font specified.

Color and Fill Pattern Attributes

You can set the following color and pattern attributes in a custom or named visual attribute definition:

- Foreground / Background Color
- Fill Pattern

Foreground and Background Color Color is applied to objects in two layers—a foreground layer and a background layer. You specify the color of each layer by setting the Foreground Color and Background Color attributes.

Pattern You can set the Pattern attribute to any of the predefined patterns available on your system. Patterns are identified by name. For example, the pattern named 45thinline is a pattern of thin, diagonal lines on a solid background.

How Fill Pattern Affects Color Attributes Oracle Forms renders the selected pattern in the foreground and background colors applied to an object. The current pattern selection determines how color is applied:

- When the current pattern is Solid (the black square in the pattern palette), the object displays in the color specified by the Foreground Color attribute.
- When the current pattern is Transparent (the white square in the Layout Editor pattern palette), the object displays in the color specified by the Background Color attribute.
- When the current pattern is Clear, the object is transparent, and neither the foreground color or background color is applied.

  Note: The Clear pattern is not valid for items. For best results, use the Transparent pattern for items (the solid white square in the Layout Editor pattern palette).
- When the current pattern is any other pattern, the black elements of the pattern are displayed in the current foreground color and the white elements of the pattern are displayed in the current background color.

Logical Attribute Descriptions

Oracle Forms logical attributes are organized into two groups: GUI (forms_logicals) and character mode (cm_logicals).

Cm_Logicals

Use Oracle Terminal to change the defaults for the Oracle Forms–specific visual attributes for character–mode terminals, which are listed below.

Attribute Name	*Context Where Attribute Is Used*
ToolkitDisabled	Generic attribute
ToolkitEnabled	Generic attribute
ToolkitCurrent	Generic attribute
ToolkitDisabledMnemonic	Generic attribute
ToolkitEnabledMnemonic	Generic attribute
ToolkitCurrentMnemonic	Generic attribute
NormalAttribute	Normal background for windows
Normal	Text item
Bold	Bold for all items (including check boxes)
Bold–text	Boilerplate
Bold–inverse	Inverse bold for all items
Underline	Underline for all items
Boilerplate	Constant text
WindowTitleCurrent	Title of active window
Menu	Selected menu
Sub–menu	Selected submenu
Full–screen–title	Screen title
Menu–title	Current menu title
Menu–subtitle	Current menu subtitle
Menu–bottom–title	Current title at bottom of menu
MenuItemEnable	Enabled, non–current menu item
MenuItemDisabled	Disabled menu item
MenuItemSelect	Current menu item
MenuItemEnableMnemonic	Mnemonic of an enabled menu item
MenuItemDisableMnemonic	Mnemonic of a disabled menu item

Attribute Name	*Context Where Attribute Is Used*
MenuItemSelectMnemonic	Mnemonic of the current menu item
TextControlCurrent	Current field or text editor
Field–current	Color for current text item
TextControlNonCurrent	Disabled or non–current field or text editor
Field–non–current	Color for text item that is not currently selected
TextControlSelect	Selected text in an enabled field or text editor
Field–selected–current	Currently selected text item
Field–selected–non–current	Text item that is not currently selected
Field–queryable	Field that operator can query
PushButtonDefault	Default or current button
PushButtonNonDefault	Button that is not default
Button–non–current	Non–current button
Button–current	Current button
Alert	Alert text
AlertIcon	Icon in an alert window
AlertMessage	Message text in an alert window
AlertBackground	Alert background
Listtitle	List of Values (LOV) title
ListItemSelect	Selected item in a text list
ListItemNonSelect	Unselected item in a text list
ListPrefix	List prefix
ScrollThumb	”Elevator box” on scrollbar

Forms_Logicals

Use Oracle Terminal to change the GUI defaults for the Oracle Forms–specific visual attributes, which are listed below.

Attribute Name	*Context Where Attribute Is Used*
TextControlFailValidation	Text item that fails validation
ItemQueryDisabled	Text item that is explicitly disabled by the operator, so it cannot be included as query criteria
Status–Message	Message appearing on status line

Attribute Name	*Context Where Attribute Is Used*
Status–Hint	Hint appearing on status line
Status–Empty	Status line with no text
Status–Items	Indicators (lamps) on status line
Scroll–bar–fill, Inverse, Inverse–underline, Bold–underline, Bold–inverse–underline	These logical attributes are not unique to Oracle Forms. As a result, these logical attributes can be overridden by the visual attributes defined by the window manager.

APPENDIX

B National Language Support

This appendix discusses National Language Support (NLS) as it affects developers of Oracle Forms applications. Topics covered in this appendix include:

About National Language Support

When you design applications for international use, you want those applications to interact with form operators in their native language, using their specific local conventions for displaying data. Oracle's National Language Support (NLS) includes three main layers:

- Support for national language character encoding schemes
- Support for language and territory conventions
- Interface translation

This combination of support at the character level, the data display level, and the interface level ensures that your forms will be usable in most European, Middle Eastern, and Asian languages, including multi–byte character encoding schemes and bidirectionality.

Designing and deploying multi–lingual Oracle Forms applications requires a two–part strategy:

- translating the Oracle Forms user interface
- translating the application–specific messages

In general, there are two types of application–specific data you'll need to translate:

- ”translatable text,” such as form menus, boilerplate text, item labels, messages, and hints defined on item property sheets
- strings in triggers and procedures, including alerts and messages defined in triggers

The Oracle Translation Manager product helps you translate the ”translatable text” strings in the Oracle Forms user interface into multiple languages. (Translatable text appears as a hexadecimal representation in the .FMT file in order to support multi–byte character sets, so you cannot translate it directly).

You can use Oracle Translation Manager to assist with translation for almost all multi–lingual forms applications. Unless your application requires operators to toggle between languages at runtime, you can use Oracle Translation Manager and then generate separate executable files for each language.

While the Oracle Translation Manager helps you find and translate strings in the user interface, the tools cannot pull out string constants in PL/SQL triggers and procedures. Manual translation is required for constant text within PL/SQL block, because that text is not clearly delimited, but is often built up from variables and pieces of strings.

For a complete description of the Oracle Translation Manager and its translation tools, refer to the *Oracle Translation Manager User's Guide*.

NLS Architecture

National Language Support for Oracle Forms leverages off Oracle's NLS architecture, which has two main parts:

- Language–independent features
- Language–dependent data

Oracle Forms uses language–independent features to handle manipulation of string data in an appropriate manner, depending on the language and territory of the runtime operator, and automatically format that data according to local date–and–time conventions.

Isolating the language–dependent data allows your application to deal only with translating the strings that are unique to your application.

Error Messages and Boilerplate Text

All error messages and boilerplate text should be in the same language. When running an Oracle Forms application, the operator views messages and boilerplate text from three sources:

- Error messages from the database
- Runtime error messages produced by Oracle Forms
- Messages and boilerplate text defined as part of the application

Oracle's NLS handles translation for the first two categories. You, as the application developer, are responsible for translating the messages in the third category. This rest of this appendix outlines several approaches to producing multilingual Oracle Forms applications.

Character Encoding Schemes

Oracle Forms and PL / SQL offer complete internal support for both single and multi–byte character encoding schemes.

At the lowest level, national language support requires that the terminal be able to display the correct characters for the local character set. To display the correct characters requires that the software use the character encoding scheme expected by the hardware. In general, computers use one of two groups of encoding schemes, ASCII or EBCDIC. Within each group, all schemes normally use the same encoding for the standard Latin alphabet (a to z), but use different encoding for special characters used in languages other than English.

The language used for operator input is determined by the operator and limited only by the characters available in the character encoding scheme supported by the terminal. For example, ISO8859–1 is an extended Latin character set that supports more than 40 Western European languages.

Oracle's NLS features solve the problems that result from the fact that different encoding schemes use different binary values to represent the same character. With Oracle's NLS, data created with one encoding scheme can be correctly processed and displayed on a system that uses a different encoding scheme. This feature is automatic and requires no programming, as it relies on the NLS_LANG character set component to describe the user's environment.

About the NLS Language Environment Variables

Use the NLS_LANG environment variable to control the language–dependent operation of applications. NLS_LANG sets the language for Oracle Forms messages displayed to the operator, such as the ”Working...” message. In addition, the NLS_LANG environment variable determines the default format masks used for DATE and NUMBER datatypes, sorting sequence, and the characters that make up the character set.

The syntax for NLS_LANG has three components:

```
NLS_LANG=language_territory.char_set
```

where:

language	Specifies the language and its conventions for displaying messages and day and month names.
territory	Specifies the territory and its conventions for default date format, decimal character used for numbers, currency symbol, and calculation of week and day numbers.
char_set	Specifies the character set used for the UPPER, LOWER, and INITCAP functions. This argument also controls the character set used for displaying messages.

Examples:

```
NLS_LANG=French_France.WE8DEC
NLS_LANG=French_Switzerland.WE8DEC
NLS_LANG=Norwegian_Norway.NDK7DEC
NLS_LANG=Norwegian_Norway.WE8DEC
NLS_LANG=Japanese_Japan.JA16SJIS
NLS_LANG=America_.AR8ISO8859P6
NLS_LANG=America_.WE8ISO8859P1
```

In most cases, setting a single value for NLS_LANG will meet your needs. However, if you need to use two sets of resource and message files on the same machine at the same time, two other environment variables are available:

- DEVELOPER_NLS_LANG
- USER_NLS_LANG

Use these two variables, instead of the single NLS_LANG, in these circumstances:

- If you are a developer who prefers to use the Designer in English, but you are developing an application for another language, the two variables allow you to use different language settings for the Designer and Runform.
- If you are a developer creating an application to run in a language for which a local–language version of the Designer is not currently available.

Note: If these environment variables are not specifically set, they take their default values from NLS_LANG, so for the rest of this discussion, ”the language environment variable” will refer to the current language environment variable, whether it’s set using NLS_LANG or one of the other variables.

Additional NLS Environment Variables

Specifying the NLS_LANG environment variable sets default values for additional NLS environment variables. If you want override the defaults defined by a specific value of NLS_LANG, you can specifically set the following NLS environment variables.

Environment Variable	Specifies	Example
NLS_DATE_FORMAT	default date format	MM/DD/YYYY
NLS_DATE_LANGUAGE	language for day and month names	GERMAN
NLS_SORT	sort sequence	SWEDISH
NLS_NUMERIC_CHARACTERS	decimal character and group separator for numeric characters	., (specify both characters)
NLS_CURRENCY	local currency symbol (L)	$
NLS_ISO_CURRENCY	ISO currency symbol (C)	FRANCE (territory name)
*NLS_MONETARY_CHARACTERS	decimal character and group separator for monetary characters	., (specify both characters)
*NLS_LIST_SEPARATOR	separator for a list of items (if comma is used within items)	:
*NLS_DEBIT	debit amounts	DB
*NLS_CREDIT	credit amounts	CR
*NLS_CALENDAR	calendar format	Persian

*These environment variables are supported for Oracle7.2 only.

For example, to specify the NLS_DATE_FORMAT in a Windows environment, add this setting to your ORACLE.INI file:

```
NLS_DATE_FORMAT=MM/DD/YYYY
```

For more information about NLS environment variables, see the *Oracle7 Server Reference*, Chapter 3, ”National Language Support.”

Oracle Forms NLS Parameters

You can use Oracle Forms built–in functions to obtain the current value of the NLS environment variables for use in trigger code:

Environment Variables	NLS_LANG	
	DEVELOPER_NLS_LANG	USER_NLS_LANG
Built–in	GET_FORM_PROPERTY	GET_APPLICATION_ PROPERTY
Parameter	Module_NLS_Lang	User_NLS_Lang

Because both USER_NLS_LANG and DEVELOPER_NLS_LANG default to the value of NLS_LANG, the Oracle Forms NLS parameters will hold the value of NLS_LANG if either variable is not specifically set.

Both Oracle Forms NLS parameters have four variations, to allow you to retrieve either the complete environment variable or a specific portion of it. This table shows the four parameters of the GET_APPLICATION_PROPERTY built–in that return the USER_NLS_LANG environment variable:

Parameter	*Returns*
USER_NLS_LANG	Entire USER_NLS_LANG variable
USER_NLS_LANGUAGE	Language portion only
USER_NLS_TERRITORY	Territory portion only
USER_NLS_CHARACTER_SET	Character set portion only

To retrieve the DEVELOPER_NLS_LANG environment variable, call GET_FORM_PROPERTY, using the MODULE_NLS_LANG parameter.

For more information on the GET_APPLICATION_PROPERTY and GET_FORM_PROPERTY built–ins, see the *Oracle Forms Reference Manual*, *Vol. 1*, Chapter 3, ”Built–ins.”

Character Set Design Considerations

When you begin to design forms for use in multiple languages, you need to consider both the character sets used by form operators, and those used by form developers. While conversion at the character set level at runtime is automatic, this conversion logically does impact performance.

For example, if you design and generate a form in one character set and run it in another character set, you may notice a degradation in performance. In addition, if a character set does not contain an equivalent for a special character, it will display a question mark instead of that character. However, even if you don't see any visible difference, as a developer you will want to pay attention to the character sets you use to generate, because of performance issues.

If you're designing forms to run in more than one language, or even in the same language with different character sets, for best performance you will want to match your generate environment to your runtime environment:

- Determine the character set most widely used at runtime.
- Generate with the NLS environment variable set to that character set.

Tip: If you want to use a font that belongs to a character set other than the font set in the NLS environment variable, set it through the Layout Editor Format–>Font dialog, rather than the Properties window.

Language and Territory Default Format Masks

While the character encoding schemes ensure that the individual characters needed for each language are available, support for national conventions provides correct localized display of data items, such as:

- day and month names
- number formatting
- date formatting
- currency formatting
- sorting sequence for character data

The language environment variable establishes not only the language used for the Oracle Forms interface (both Designer and Runtime), but also the set of default format masks used to display that data, unless explicit format masks override the default. As a form developer, you also have additional NLS–related format mask characters available if you choose to override the default display masks.

Format Mask Design Considerations

All data input into the Designer or Runform will use the default format masks associated with the specified NLS_LANG, so you can change languages when designing a form as well as when running the form, without needing to change the date and number formats.

Specifically, Oracle Forms will use the default format masks associated with the territory specified in the current language environment variable:

- in the Designer: When Oracle Forms displays default values for items, ranges, or parameters
- at runtime: If a user enters data into a text item whose type is territory–specific, such as DATE or NUMBER

For example, suppose that you create an item of type DATE in the Designer, and then enter a default value of 20–DEC–93, using the NLS_LANG default of American_America. If you then change NLS_LANG to Norwegian, the default value for the item will automatically change to 20.12.1993.

Tip: For implicit datatype conversions, PL/SQL always expects items in the American_America default format (such as DD–MON–YY), so if you use an item whose type is territory–specific in PL/SQL, you will need to specify the correct format masks. Oracle recommends that you use TO_DATE to translate territory–specific items in PL/SQL.

Examples: For NLS–compliant applications, avoid hard–coding a string containing a month name. However, if a hard–coded month name is essential, avoid using the COPY built–in: If you use COPY, the month name may be incorrect, depending on which language is specified. (To specify a date in a library, use a variable for the date and COPY will work.)

Language–dependent example (not recommended):

```
:emp.hiredate := ’30–DEC–93’;
copy (’30–DEC–90’,’emp.hiredate’);
```

Instead, use TO_DATE, as shown in the following example.

Language–independent example (recommended):

```
:emp.hiredate :=  to_date(’30–12–1990’,’DD–MM–YYYY’);
```

Format Mask Characters

The NLS–related format mask characters allow you to override the default format masks:

Character	*Returns*
C	International currency symbol
L	Local currency symbol
D	Decimal separator
G	Group (thousands) separator

While working with date and currency fields for multilingual applications, you will want to consider:

- Whether to make all screen items (boilerplate, text items, interface objects such as buttons and lists of values) longer to accommodate text translation and language–specific conventions for numeric display.

  For example, if you develop an application in US–English with a 9–character DD–MON–YY date, and plan to run the application in Norwegian, you'll want to increase the size of the field to allow for the NLS translation to a 10–character Norwegian date: DD.MM.YYYY. By default, Oracle Forms increases the maximum length but not the display length for dates. You must explicitly increase the display length, if necessary.

- Whether you need to use the NLS–related format mask characters to create special format masks, or if the default format masks invoked by NLS_LANG are sufficient.

For more information about NLS–related format mask characters, see the Format Mask property description in the *Oracle Forms Reference Manual*, *Vol. 2*, Chapter 5, "Properties."

Screen Design Considerations

When designing an application that will be translated, remember to leave extra space in the base screen design for widgets and boilerplate labels. To accommodate multiple character sets and allow for expansion caused by translation, a rule–of–thumb is to leave 30% white space around fields, borders, and boilerplate text.

For example:

- Prompt on left of field: Allow 30% expansion room to left of prompt.
- Prompt above field: Allow 30% expansion room to right of prompt.
- Buttons, checkboxes, radio groups and poplists: Allow for 30% expansion.
- Form titles: Size any bounding box so the title can expand to right by 30%.
- Display–only fields: Size 30% wider than needed for base language.
- All widgets: Make widgets large enough to accommodate translation. For example, buttons need to be large enough to hold translated labels. (Check button height as well as length, to be sure the height of the button will accommodate the tallest character you need to use. For example, calculate pixels needed to render Kanji characters.)

Oracle Forms Interface Translation

Translation of the Oracle Forms Designer user interface requires that appropriate message files be installed during the Oracle Forms installation procedure. With the appropriate message files installed for a given language, the entire Oracle Forms interface will be translated into the local language you set with the NLS language environment variable, including:

- messages
- menus and menu items
- dialog boxes
- prompts and hints
- alerts

When you start the Designer in a different language, you'll see prompts in that language.

Note: It is not necessary to translate numbers or dates in default values, ranges, and parameters, because Oracle Forms translates them automatically, based on the value of the language environment variable.

Message Files

When you change languages, Forms searches for the correct message file, based on the NLS_LANG variable. If Forms does not find a corresponding message file, it defaults to the US message file.

Message file names specify both the Oracle Forms component and the language used. For example:

- FMDUS.MSB contains default (American) Designer messages
- FMGUS.MSB contains default (American) Generate messages
- FMFUS.MSB contains default (American) Runform messages
- FMFI.MSB contains Italian Runform messages
- FMFD.MSB contains German Runform messages

For more information about message file names, see the Installation and User Guide for your platform.

Using the Oracle Translation Manager Approach

You can use Oracle Translation Manager to assist with translation for almost all multi–lingual forms applications. Unless your application requires operators to toggle between languages at runtime, you can use Oracle Translation Manager and then generate separate executable files for each language.

To change from one language to another at runtime, the user exits Oracle Forms, changes the NLS environment variable, and then restarts Oracle Forms.

Using the Oracle Translation Manager approach, you would develop your application in the following stages:

- Create one basic form definition (.FMB, .MMB) in the source language.
- Use the Oracle Translation Manager to extract strings for translation, allow translation into one or several languages, and then store the strings back into the form definition.
- Manually translate standard messages in PL/SQL libraries.
- Use the Oracle Forms Generate component to generate an executable version of the form for each target language.

At runtime, the form is displayed in the appropriate language, depending on which translated .FMX file is first in the search path you set in the ORACLE.INI parameter, FORMS45_PATH.

Advantages: This is the simplest way to implement multiple language applications quickly. With this approach, you can use the Oracle Translation Manager translation tools for maximum efficiency. If you have a stable application, this approach will work well for you.

For example, if you created an application in four languages and then needed to change the text of a button label, you would make the change in the Designer, save the change in the .FMB file, return to the Oracle Translation Manager translation tools to translate the new button label, then insert the new messages into the .FMB file and regenerate to create an .FMX file containing the new button.

Disadvantages: If your applications must support multiple languages simultaneously, you must use the runtime language switching approach, instead.

Using the Runtime Language Switching Approach

A small number of applications must support multiple languages simultaneously. For example, the application may begin by displaying a window in English which must stay up throughout the application, while an operator may press a button on that window to toggle the prompts into French, and then back into English.

If your application requires runtime language switching, you can include more than one language in a single application as long as they share the same character set, but you cannot use the Oracle Translation Manager to locate translatable text if you are dynamically populating the text at runtime. Instead, you would build case structures (IF...THEN...ELSIF) to change the application to another language by checking the value of the NLS environment variable using the GET_FORM_PROPERTY built–in.

Using the runtime language switching approach, you would develop your application in the following stages:

- Develop the entire form for one language, including libraries.
- Manually translate each library.
- Design boilerplate labels as appropriately–sized display items that are dynamically populated at runtime.

Oracle Forms supports attaching multiple libraries, so you can use one library specifically for messages that will be translated, and other libraries for other purposes.

Advantages: The main advantage of this approach is it allows you to support sophisticated applications in which the forms may be highly dynamic. In these cases, this approach avoids some maintenance problems, because you do not have to generate separate files for each language each time the form changes.

Disadvantages: This approach is more complicated, because it involves considerable effort to create the language–specific message storage, population, and maintenance involved and to perform the translation manually. For example, you would set up a WHEN–NEW–FORM–INSTANCE trigger to set the labels for each button, pulling the correct labels from an attached library, based on the value of the NLS environment variable.

Using PL/SQL Libraries for Strings in Code

While the Oracle Translation Manager assists in translating messages that are part of standard Oracle Forms interface, messages that are displayed programmatically require special treatment. You can use the PL / SQL libraries to implement a flexible message structure for messages in triggers or procedures.

For application–specific messages—those displayed programmatically to the screen by the built–in routines MESSAGE or CHANGE_ALERT_MESSAGE, or by assigning a message to a display item from a trigger or procedure—you can use the attachable PL / SQL libraries to implement a flexible message function. The library can be stored on the host, then dynamically attached at runtime. At runtime, based on a search path, you can pull in the library attached to the form. (For example, a library might hold only the Italian messages.)

Example:

```
FUNCTION nls_appl_mesg(index_no NUMBER)
RETURN CHAR
IS
   msg CHAR(80);
BEGIN
   IF     index_no = 1001 THEN
      msg := ’L’’impiegato che Voi cercate non esiste...’;
   ELSIF  index_no = 1002 THEN
      msg := ’Lo stipendio non puo essere minore di zero.’;
   ELSIF  ...
   :
   ELSE
     msg := ’ERRORE: Indice messaggio inesistente.’;
   END IF;
   RETURN msg;
END;
```

A routine like this could be used anywhere a character expression would normally be valid: for example, to display an alert with the appropriately translated application message you might include the following code in your form:

```
Change_Alert_Message(’My_Error_Alert’, nls_appl_mesg(1001));
n := Show_Alert(’My_Error_Alert’);
```

To change the application to another language, simply replace the PL / SQL library containing the `nls_appl_mesg` function with a library of the same name containing the `nls_appl_mesg` function with translated text.

If the application needs to support multiple languages simultaneously, the function above could accept an additional parameter indicating the desired language and the code would then contain an additional level of IF...THEN...ELSIF...ELSE...END IF to handle the various languages.

Using Bidirectional Support

NLS support for Middle Eastern and North African languages includes bidirectional support for languages whose natural writing direction is right–to–left.

Bidirectional support allows developers to control:

- layout direction, which includes displaying items with labels at the right of the item and correct placement of check boxes and radio buttons
- reading order, which includes right–to–left or left–to–right text direction
- alignment, which includes switching point–of–origin from upper left to upper right
- initial keyboard state, which controls whether Local or Roman characters will be produced automatically when the operator begins data entry (the operator can override this setting)

In addition to the NLS_LANG environment variable, bidirectional applications may use two additional NLS_LANG environment variables:

- DEVELOPER_NLS_LANG
- USER_NLS_LANG

For more information, refer to ”About the NLS_LANG Environment Variables,” earlier in this chapter.

Four properties are used to specify the appearance of objects in bidirectional applications:

- Direction
- Alignment
- Reading Order
- Initial Keyboard State

Direction is an umbrella property that provides as much functionality for each object as possible. For all objects except text items and display items, the Direction property is the only bidirectional property, and its setting controls the other aspects of bidirectional function. (List items, however, include an Initial Keyboard State property.)

Text items and display items do not have a Direction property; instead, you can specifically set Alignment, Reading Order, and Initial Keyboard State properties for these items.

When the bidirectional properties are set to Default, those properties inherit their values from the natural writing direction specified by the NLS language environment variable. In most cases, this will provide the desired functionality. You only need to specify the bidirectional properties when you want to override the inherited default values.

This chart summarizes inheritance for bidirectional properties.

	Default Setting Derives Value From This Object
Form	NLS environment variable
All objects, such as Alert, Block, LOV, Window, and Canvas–view	Form
All items, such as Text Item, Display Item, Check Box, Button, Radio Group, and List Item	Canvas–view

For more information, see the property descriptions for Direction, Alignment, Reading Order, and Initial Keyboard State in the *Oracle Forms Reference Manual, Vol. 2*, Chapter 5, ”Properties.”

Most properties related to bidirectional function are programmatically gettable and settable. For more information, see the corresponding built–in subprogram descriptions. For example, for information about getting the value of the Direction property for buttons, consult the description for GET_ITEM_PROPERTY in the *Oracle Forms Reference Manual, Vol. 1*, Chapter 3, ”Built–ins.”

Oracle Reports and Oracle Graphics also implement bidirectional support, so you can have reports and graphics that include both local and Roman languages.

APPENDIX

C

PECS: Performance Event Collection Services

This chapter discusses Performance Event Collection Services (PECS) as it affects developers of Oracle Forms applications. Topics covered in this chapter include:

About PECS

As an Oracle Forms developer or a system administrator, you may have questions about the performance of both Oracle Forms and your own applications. Oracle Forms' Performance Event Collection Services (PECS) will help you answer questions such as these:

- How much elapsed time does it take to use LOVs on three fields in my form?
- How long does it take to execute these triggers?
- How many invoices per hour can we produce?
- Have I tested all the triggers in my form?
- Have I tested all the lines of my PL/SQL code?

The PECS System

The PECS system consists of three main parts:

- PECS built–ins, used to define application–specific events
- PECS Assistant, an Oracle Forms application you can use to load and view PECS data
- PECS reports, which let you view the results of your tests on line or print out hard copy

PECS Measurements

PECS is a low–overhead performance measurement tool you can use to perform the following tasks:

- Measure resource usage for Oracle Forms events

  The simplest way to use PECS is to measure only Oracle Forms events. If you only want to measure Oracle Forms events, you do not need to add PECS built–ins to your PL / SQL code because Oracle Forms includes pre–defined PECS events to measure the performance of objects such as triggers, windows, canvases, LOVs, and alerts.

- Measure resource usage for application–specific events (CPU time and elapsed time)

  If you add calls to PECS built–ins to your PL / SQL code, you can measure the performance of application–specific events, as well as Oracle Forms events.

- Locate performance problems
- Measure coverage

  When you're testing an Oracle Forms application, you may be interested in what kind of coverage your test cases provide. The PECS Object Coverage Report gives you information about the coverage provided by your current test suite.

  The PECS Line Coverage Report provides information about coverage of PL / SQL code.

Collecting Performance Data on Oracle Forms Events

Because Oracle Forms includes pre–defined PECS events, you can measure both performance and coverage without making any changes in your application. PECS measures:

- The *performance* of the entire application, triggers, and LOVs
- The *coverage* of procedures, windows, canvases, editors, and alerts, and line–by–line coverage for PL/SQL code

The process has three steps:

1. Run your application with PECS on to collect performance data in a .DAT file, using the following command:

   ```
   f45run module=myform userid=scott/tiger pecs=on
   ```

   Or, at the Runform Preferences dialog "Collect PECS data?" option, select ON.

   Running Oracle Forms with PECS creates a PECS binary file with a default name of the module name with a .DAT extension:

   ```
   myform.dat
   ```

   The data is written to a file rather than directly to the database.

2. Load data from the .DAT file and your form into the database.

   Use the PECS Assistant to load both your form and the PECS binary (.DAT) file into the database.

   For information about the PECS Assistant, see "Using the PECS Assistant" later in this chapter.

3. Analyze the data.

   Once the PECS database file is created, review the results:

   - Use the PECS Assistant Data or Summary windows to view the results on line, or
   - Run the Performance, Object Coverage, or Line Coverage Reports and view the results on line or print out hard copy

Although you don't need to use the PECS built–ins to collect performance data on Oracle Forms events, you can use the built–ins to limit the focus of your PECS experiments. For example, if you are only interested in LOV events, you can use the PECS.DISABLE_CLASS built–in to turn off all events except the LOV events.

Collecting Performance Data on Application–Specific Events

To use PECS to measure application–specific events, first define the events you want to measure by adding PECS built–ins to your PL / SQL code. (For best results, define only a few most important events.)

Using PECS to collect performance data on application–specific events is a five–stage process:

- Group application–specific events.
- Define the events you want to measure, using PECS built–ins.
- Run your application with PECS.
- Load the form and the PECS data from the file into the database.
- Analyze the data.

Group Application–Specific Events

Application–specific events are the business transactions that you define: for example, in a banking application, you might define Deposit, Withdrawal, and Transfer events.

You can group like events into classes in order to create reports that display your test results meaningfully. In a banking application, you might create two classes:

- A Banking class, composed of three events:
  - Deposit event
  - Withdraw event
  - Transfer event
- A Customer class, composed of three events:
  - AddCustomer event
  - DeleteCustomer event
  - ChangeCustomer event

Define PECS Events

Once you’ve chosen the set of events you want to measure, you’ll prepare your application to use PECS by adding PECS built–ins to your PL/SQL code:

- Start PECS:
  - At design time, by adding the PECS.COLLECTOR built–in to your application (to collect data on application–specific events only), or
  - At runtime, by invoking PECS from the command line (to collect data on Oracle Forms events, as well as on application–specific events)
- Add any classes and events you need with the PECS.ADD_CLASS and PECS.ADD_EVENT built–ins
- Enable the class or specific event you want to measure using the PECS.ENABLE_CLASS built–in
- Define the events you want to measure:
  - Bracket them with the PECS.START_EVENT and PECS.END_EVENT built–ins, or
  - Mark them with PECS.POINT_EVENT

To invoke PECS from the command line, set PECS=ON for object coverage and PECS=FULL for object and line coverage.

Note: You can use PECS built–ins wherever you can use other Oracle Forms built–ins: triggers, user–named routines, menu PL/SQL commands, and stored procedures. While you can’t directly measure the duration of a PL/SQL procedure, you can define a trigger that calls the procedure and then measure the duration of that trigger.

For more information about the PECS built–ins, refer to the section ”PECS Built–ins” later in this chapter.

Run Application with PECS

Once you have added PECS built–ins to your application, when you run the form PECS automatically creates a binary file containing the test data:

- If you started PECS from the command line, the name of the PECS binary file defaults to the form name with a .DAT extension: `myform.dat`.
- If you started PECS by adding the PECS.COLLECTOR built–in to your application, the name of the PECS binary file defaults to the name of the first class you specified with the PECS.ADD_CLASS built–in: `banking.dat`.

Load the Form and the PECS Data into the Database

Use the PECS Assistant application to load both your form and the PECS binary file into the database. For more information, see "Using the PECS Assistant" later in this chapter.

Note: Unless you want coverage statistics, PECS does not require that you store your form in the database. However, PECS depends on some of the tables that are created automatically by the Forms 4.5 table scripts that are run during installation when you choose the "Create database tables" option. If you did not choose that option, you or your DBA must logon as SYSTEM and run those SQL scripts, which are found in the Oracle Forms SQL directory.

Analyze the Data

Once the PECS .DAT file has been loaded into the database, you can review the results using the PECS Assistant:

- Use the Data or Summary windows to view the results on line, or
- Run the Performance, Object Coverage, or Line Coverage Reports and view the results on line or print out hard copy

For more information, see "PECS Reports" later in this chapter.

Next Steps

Once you've analyzed the data, you may decide to proceed in any of several directions:

- If you are debugging, you may want to change the application and run the same tests again.
- If you are testing *performance*, you may want to create multiple runs of the same test, average the statistics, and compare results with previous tests.
- If you are testing *coverage*, you may want to change the test to add test cases to cover more conditions, then re–run the test and compare results with earlier tests.

For more information about using PECS data files, see "Using .DAT Files" later in this chapter.

Example:

Assume that you've decided to measure both internal Oracle Forms events and three application–specific events: Deposit, Withdrawal, and Transfer. This example illustrates adding the PECS built–ins to define these events to PECS.

(Because you want to measure both types of events, you could have started PECS on the command line, but this example includes starting PECS with the PECS.COLLECTOR built–in for purposes of demonstration.)

```
PACKAGE globals IS
/* We will create a package of global variables instead of     */
/* local variables because the pecs.event and pecs.class types */
/* may be spread across various procedures, triggers, and      */
/* even forms.                                                 */

/*                                                             */
/*   Add the classes.                                          */
/*                                                             */
banking PECS.class  := PECS.Add_Class('Banking');
myform  PECS.class  := PECS.Add_Class(PECS.FORMS);
/*                                                            */
/*   Add the events for the class 'Banking'.                  */
/*   No need to add events for the class 'Form'               */
/*   because they are pre–defined internal events.            */
withdraw pecs.event  := PECS.Add_Event(banking,'Withdraw');
deposit pecs.event   := PECS.Add_Event(banking,'Deposit');
transfer pecs.event  := PECS.Add_Event(banking,'Transfer');

END globals;
```

```
/*   Trigger: When–New–Form–Instance                        */
/*   Enable PECS                                            */

BEGIN
PECS.Collector(COLLECT_ON);

/*                                                          */
/*   Enable the classes                                     */
/*                                                          */

PECS.Enable_Class(GLOBALS.banking);
/*                                                          */
/* After the forms class has been enabled,                  */
/* internal forms events will be measured.                  */
/* NOTE: This line is not needed if you use the             */
/*       PECS=on command line option.                      */

PECS.Enable_Class(GLOBALS.myform);
END;

PROCEDURE do_withdraw

/*                                                          */
/*   This is the code that does the Withdraw Event.         */
/*                                                          */

DECLARE
withdraw_handle number;

BEGIN

/*                                                          */
/*   Start the Withdraw event               .               */
/*                                                          */

withdraw_handle := PECS.Start_Event(GLOBALS.withdraw,
’Withdrawing cash’);

.
.   /do the withdraw/
.

PECS.End_Event(GLOBALS.withdraw,withdraw_handle);
END;
```

Collecting Object Coverage Data

When you’re testing an Oracle Forms application, you can also use PECS as a quality assurance tool. The PECS Object Coverage Report lets you evaluate the percentage of coverage provided by your current set of test cases.

When you run PECS to collect information on Oracle Forms events, either separately or in conjunction with collecting information on application–specific events, you can run the Object Coverage Report as well as the Performance Report.

By examining the Object Coverage Report, you can see whether your current set of test cases exercises all of the Oracle Forms objects in your application. For example, you can check whether each trigger, procedure, alert, window, and canvas are visited during execution of your current set of test cases, and find any areas where you may need to construct new test cases for increased coverage.

Note: If you want coverage statistics, you must store your form in the database.

Collecting Line Coverage Data

In addition to testing for simple coverage of test cases, you can also test for line–by–line coverage of PL/SQL code in triggers and procedures. The PECS Line Coverage Report lets you evaluate the percentage of PL/SQL line coverage provided by your current set of test cases.

To obtain line coverage data:

- Generate the form with `debug=ON`.
- Run with `PECS=FULL`.
- In the PECS Assistant, select the Line Coverage check box.
- Select the Line Coverage Report.

Note: To collect line coverage data, you must use the PECS command line option (`PECS=FULL`) rather than invoking PECS with the PECS.ENABLE_CLASS built–in.

Using the PECS Assistant

The PECS Assistant application provides the following functions:

To do this:	*Use this window:*
Load PECS data into tables	Load
Display data	Data, Summary
Display information about classes and experiments	Class, Experiment

For an overview of each window and detailed information about each field, click on the Help button.

About the PECS Hierarchy

The PECS Assistant asks you to specify names of occurrences, events, classes, runs, and experiments.

These terms are used in the PECS hierarchy as follows:

This group:	*Consists of:*
Experiment	One or more test runs. Experiments own runs.
Run	One pass through your application from start to finish. A single .DAT file maps to a single run of an experiment.
Class	A group of related events. Classes own events.
Event	An application–specific or form–specific transaction with a defined start and end.
Occurrence	One example of a given event.

Steps for Using the PECS Assistant

Once you’ve run a form with PECS enabled, you’re ready to use the PECS Assistant application:

1. At the system prompt, enter this command to run the PECS Assistant:

   ```
   f45run module=pecs
   ```

   The PECS Assistant displays the Main window. You can navigate to other windows using the buttons or the commands on the Windows menu.

2. Click Load.

   The PECS Assistant displays the Load window.

3. Fill in the ”PECS Data” section to select the experiment you want to load into the database.

   If the file does not exist, an Open error message is displayed.

   Note: Fill in the ”Connect String” section only if you want to use a connect string that differs from the one you used to start the form. For more information, press Help.

4. Check the appropriate check boxes.

   The ”Load PECS data in the database” and ”Summarize data” check boxes are checked, as default.

   If you want to include line coverage data, check the ”Prepare for Line Coverage” check box.

   Note: Preparing for line coverage is a time–consuming task, so the ”busy” cursor will be displayed for a while.

   If you want to delete the detailed data once the summary files are created, check the ”Delete details after summary” check box.

5. Click Start.

   PECS displays confirmation messages for each check box you’ve selected.

   Once you’ve accepted each confirmation, PECS processes the data, displaying status messages in the ”Status” area during processing, and prepares the results for you to examine.

6. To view the results, choose Navigation–>Experiment or Navigation–>Class.

7. From the Experiment or Class window, query to select the experiment or class you want to review.

   PECS displays the results of your query.

8. Highlight the specific experiment or class.

9. Click Zoom In.

   PECS displays the Summary window, showing the summarized version of the data. The Summary window displays the information in a format similar to the PECS reports.

10. Click Zoom In again to examine the detailed results of a PECS experiment.

    PECS displays the detailed results in the Data window.

PECS Reports

To run the PECS reports, choose the Reports menu:

- Choose the Performance Report to show elapsed time and CPU time.
- Choose the Object Coverage Report to show that certain events or occurrences have been reached.
- Choose the Line Coverage Report to show coverage of your PL / SQL code.

By default, the Performance Report displays data in the sequence of columns as displayed in the Data or Summary screen. To change the sequence in the report, change the Order By clause in the Data or Summary Performance Report window.

PECS Maintenance

To review data about classes and experiments you defined, return to the Main window and click Class or Exp (Experiment):

- Use the Class window to view data about the classes and events you defined using the PECS built–ins.

  You can perform minimal maintenance on classes using this window. For instance, you can change class names and add version information.

- Use the Experiment window to view data about the experiments and runs you defined using PECS built–ins.

You can perform minimal maintenance on experiments using this window. For instance, you can change experiment names and add comment information. (You specified the experiment name earlier, in the Load window or on the command line using the –n option.)

Using .DAT Files

PECS allows you to load multiple previously–created .DAT files in order to re–run experiments or to average the performance statistics. For example, for Form A, you might use .DAT files in a sequence like this:

- Experiment #1, Run #1: Produces the original A.DAT file. You evaluate the coverage results of Run #1, and decide to add more test cases to improve the coverage.
- Experiment #1, Run #2: Produces a new A.DAT file.
- When you also load the results of Run #2 to the database, check Summarize on the Load window to summarize the data from all runs of this experiment.

Using PECSLOAD

You can also access the PECS system directly from the command line, without using the PECS Assistant. Invoke the PECSLOAD utility from the command line to load the PECS binary file into the database. During PECSLOAD, both classes and events will also be saved to the database, if they are not already there.

The syntax for the PECSLOAD command requires three arguments: the module name, the experiment name or number, and userID / password.

Example 1: Using PECSLOAD with an experiment name:

```
PECSLOAD –f mymod.dat –n my_experiment scott/tiger
```

Example 2: Using PECSLOAD with an experiment number:

```
PECSLOAD –f mymod.dat –e 3 scott/tiger
```

Location of the PECS Data File

When you use the PECSLOAD command, PECS must first locate where the binary (.DAT) file was saved. The PECS data file is usually saved in a temporary directory set by an environment variable, as shown below:

Environment	*PECS Default Directory/Environment Variable*
MS Windows	TMP
UNIX	FORMS_PECS
VMS	SYS$SCRATCH

If the environment variable is not set, PECS uses the current working directory.

If you issue the PECSLOAD command and receive an error message similar to ”Error opening file,” probably PECS is searching for the data file and cannot find it. In this case, re–issue the PECSLOAD command and provide the full pathname in place of the module name.

PECS Database Tables

You may want to consult the PECS database tables to perform data manipulation or obtain specific PECS information, such as an experiment name or number. PECS uses the following database tables:

- PECS_run
- PECS_experiment
- PECS_class
- PECS_class_events
- PECS_data
- PECS_summary
- PECS_plsql

Use SQL commands to obtain a list of columns in each table.

PECS Built–ins

The PECS built–ins allow you to define application–specific events to the PECS system:

To do this:	*Use this PECS built–in:*
Start and stop PECS	PECS.COLLECTOR
Register classes and events	PECS.ADD_CLASS PECS.ADD_EVENT
Enable and disable classes	PECS.ENABLE_CLASS PECS.DISABLE_CLASS
Define event instances	PECS.START_EVENT PECS.END_EVENT PECS.POINT_EVENT

PECS.ADD_CLASS

Syntax:

```
PECS.ADD_CLASS(class_name);
PECS.ADD_CLASS(class_type);
```

Built–in Type: unrestricted procedure

Returns: PECS.CLASS

Description: PECS.ADD_CLASS creates the class, or group of events, in PECS, and returns the ID for this class. Use this class ID to identify the class in future calls to PECS. The datatype of the class ID is PECS.CLASS.

Parameters:

class_type Specifies the type of class to add. Valid numeric constant for this parameter:

PECS.FORMS	Specifies Oracle Forms.

class_name Specifies the name of the class to add. Specify the CHAR class name for application–specific classes that do not use one of the class types listed above.

Usage Notes: If you started PECS by adding the PECS.COLLECTOR built–in to your application, the name of the PECS binary file defaults to the name of the first class you specified with the PECS.ADD_CLASS built–in: banking.dat. (If you started PECS from the command line, the name of the PECS binary file defaults to the form name with the .DAT extension: myform.dat.)

Example:

```
/*
/*   This will add the class FORMS                           */
/*                                                           */
forms_class pecs.class := PECS.Add_Class(pecs.FORMS);

/*                                                           */
/*   This will add the class ’Banking’                       */
/*                                                           */
banking_class pecs.class := PECS.Add_Class(’Banking’);
```

PECS.ADD_EVENT

Syntax:

```
PECS.ADD_EVENT(class_id, event_type);
PECS.ADD_EVENT(class_id, event_description);
```

Built–in Type: unrestricted procedure

Returns: PECS.EVENT

Description: PECS.ADD_EVENT adds an event to a specified class.

For an Oracle Forms event, specify the event type from the list of Oracle Forms event types. For an application–specific event, specify the event using *event_description*, which must be unique among events in the class.

Parameters:

class_id	Specifies the ID of this class. The class ID is created with a call to PECS.ADD_CLASS, and is of datatype PECS.CLASS.
event_type	For Oracle Forms events, specifies the event type. Valid numeric constants for this parameter are PECS.LINE, PECS.FORMS_TRIGGER, PECS.FORMS_LOV, PECS.FORMS_20TRIGGER (Oracle Forms V2–style trigger event), PECS.FORMS_PROCEDURE, PECS.FORMS_ALERT, PECS.FORMS_CANVAS, PECS.FORMS_WINDOW, PECS.FORMS_EDITOR. For application–specific events, specify the *event_type* from the PECS_EVENTS table, or use the *event_description* parameter.
event_description	For application–specific events, specifies the event. The description is limited to 32 datatype CHAR characters.

Example:

```
/*
/*   This will add the class ’Banking’                       */
/*                                                           */

banking_class pecs.class := PECS.Add_Class(’Banking’);

/*                                                           */
/*   This will add the event Withdraw.                       */
/*   banking_class was previously defined by PECS.ADD_CLASS  */

withdraw pecs.event := PECS.Add_Event(banking_class, ’Withdraw’);
```

PECS.COLLECTOR

Syntax: `PECS.COLLECTOR(collector_status);`

Built–in Type: unrestricted procedure

Description: Use PECS.COLLECTOR to turn PECS on or off. This built–in allows you to start PECS later, if you haven't invoked PECS from the command line. PECS.COLLECTOR needs to be called only once per session.

Parameters: *collector_status* A NUMBER parameter that specifies whether PECS is on or off:

PECS.COLLECT_ON	Turns the collector on.
PECS.COLLECT_OFF	Turns the collector off.

Usage Notes:

- You don't need to use PECS.COLLECTOR if you turned PECS on using the command line option pecs=on.
- Although you can toggle the status of PECS.COLLECTOR, doing this may produce mismatched events. For example, if PECS is turned off after a trigger event has started, there may be no corresponding end event. During the loading of the .DAT file, PECS will detect this and issue a warning saying that a start event is missing an end event, or vice versa.

Example:

```
PECS.COLLECTOR(pecs.collect_on);
```

PECS.DISABLE_CLASS

Syntax:

```
PECS.DISABLE_CLASS(class_id);
PECS.DISABLE_CLASS(class_id, event_type);
PECS.DISABLE_CLASS(class_id, event_id);
```

Built–in Type: unrestricted procedure

Description: To disable all of the events associated with a class, specify only the *class_id*:

```
PECS.DISABLE_CLASS(class_id);
```

To disable only a specific event, supply the *class_id* and the *event_type* or *event_id*:

```
PECS.DISABLE_CLASS(class_id, event_type);
PECS.DISABLE_CLASS(class_id, event_id);
```

Parameters:

class_id	Specifies the ID of this class. The class ID is created with a call to PECS.ADD_CLASS, and is of datatype PECS.CLASS.
event_type	For Oracle Forms events, specifies the event type. Valid numeric constants for this parameter are PECS.LINE, PECS.FORMS_TRIGGER, PECS.FORMS_LOV, PECS.FORMS_20TRIGGER (Oracle Forms V2–style trigger event), PECS.FORMS_PROCEDURE, PECS.FORMS_ALERT, PECS.FORMS_CANVAS, PECS.FORMS_WINDOW, PECS.FORMS_EDITOR.
event_id	Specifies the ID of this event name. The event ID is created with a call to PECS.ADD_EVENT, and is of datatype PECS.EVENT.

Example:

```
/*   Disable the internal Oracle Forms LOV event.            */

PECS.Disable_Class(forms_class, PECS.FORMS_LOV);

/*  Disable the Withdraw event for Banking_class.            */
/*  Banking_class was previously defined with PECS.ADD_CLASS */
/*  Withdraw was previously defined with PECS.ADD_EVENT      */

PECS.Disable_Class(banking_class, withdraw);
```

PECS.ENABLE_CLASS

Syntax:

```
PECS.ENABLE_CLASS(class_id);
PECS.ENABLE_CLASS(class_id, event_type);
PECS.ENABLE_CLASS(class_id, event_id);
```

Built–in Type: unrestricted procedure

Description: To enable all of the events associated with a class, specify only the *class_id*:

```
PECS.ENABLE_CLASS(class_id);
```

To enable only a specific event, supply the *class_id* and the *event_type* or *event_id*:

```
PECS.ENABLE_CLASS(class_id, event_type);
PECS.ENABLE_CLASS(class_id, event_id);
```

Parameters:

class_id	Specifies the ID of this class. The class ID is created with a call to PECS.ADD_CLASS, and is of datatype PECS.CLASS.
event_type	For Oracle Forms events, specifies the event type. Valid numeric constants for this parameter are PECS.LINE, PECS.FORMS_TRIGGER, PECS.FORMS_LOV, PECS.FORMS_20TRIGGER (Oracle Forms V2–style trigger event), PECS.FORMS_PROCEDURE, PECS.FORMS_ALERT, PECS.FORMS_CANVAS, PECS.FORMS_WINDOW, PECS.FORMS_EDITOR.
event_id	Specifies the ID of this event name. The event ID is created with a call to PECS.ADD_EVENT, and is of datatype PECS.EVENT.

Example:

```
/*  Enable the internal Oracle Forms LOV event.            */

PECS.Enable_Class(forms_class);

/*  Enable the Withdraw event for Banking_class.              */
/*  Banking_class was previously defined with PECS.ADD_CLASS  */
/*  Withdraw was previously defined with PECS.ADD_EVENT       */

PECS.Enable_Class(banking_class, withdraw);
```

PECS.END_EVENT

Syntax:

```
PECS.END_EVENT(event_id, event_handle);
```

Built–in Type: unrestricted procedure

Description: Marks the end of a PECS event.

Parameters:

event_id	Specifies the ID of this event name. The event ID is created with a call to PECS.ADD_EVENT, and is of datatype PECS.EVENT.
event_handle	Specifies handle of this event occurrence. The event handle is created with a call to PECS.START_EVENT, and is of datatype NUMBER.

Example:

```
/*  Measure the performance of the Withdraw event            */
/*  Banking_class was previously defined with PECS.ADD_CLASS */
/*                                                           */

withdraw_handle := PECS.Start_Event(withdraw, ’Withdrawing Cash’);
.
.   /do the withdraw/
.
PECS.End_Event(withdraw, withdraw_handle);
```

PECS.POINT_EVENT

Syntax: `PECS.POINT_EVENT(event_id, event_comment);`

Built–in Type: unrestricted procedure

Description: Use PECS.POINT_EVENT to determine whether an event was reached during code execution.

Parameters:

event_id	Specifies the ID of this event name. The event ID is created with a call to PECS.ADD_EVENT, and is of datatype PECS.EVENT.
event_comment	A CHAR comment to identify the event.

Usage Notes:

- Use PECS.POINT_EVENT when you need to know only that a given application–specific event occurred, rather than its duration.
- You can use PECS.POINT_EVENT to gather coverage data for specific parts of your application where you insert PECS point events.

Example:

```
/*   Start the Withdraw event               .          */
/*                                                     */
...

...
IF (do_withdraw = TRUE) then
...
message('Success collecting data for event Withdraw');
PECS.Point_Event(GLOBALS.withdraw,'Withdraw Success');
ELSE
...
 message('Error collecting data for event Withdraw');
PECS.Point_Event(GLOBALS.withdraw,'Withdraw Error');
END IF;
```

PECS.START_EVENT

Syntax: `PECS.START_EVENT(event_id,event_ comment);`

Built–in Type: unrestricted procedure

Returns: NUMBER

Description: Marks the beginning of a PECS event. PECS.START_EVENT returns a unique identifier for this occurrence of this particular event, which is needed to end the event.

Parameters:

event_id	Specifies the event ID. Datatype is PECS.EVENT.
event_comment	A CHAR comment to identify the event.

Usage Notes: PECS.START_EVENT returns an identifier, or handle, to the event that must be used with PECS.END_EVENT to mark the end of the event.

Example:

```
/*  Measure the performance of the Withdraw event            */
/*  Banking_class was previously defined with PECS.ADD_EVENT */
/*                                                           */
withdraw_handle := PECS.Start_Event(withdraw, ’Withdrawing Cash’);
.
.   /do the withdraw/
.
PECS.End_Event(withdraw, withdraw_handle);
```

Index

Reader’s Comment Form

Name of Document: Forms™ Advanced Techniques
Part No. A32506–2

Oracle Corporation welcomes your comments and suggestions on the quality and usefulness of this publication. Your input is an important part of the information used for revision.

- Did you find any errors?
- Is the information clearly presented?
- Do you need more information? If so, where?
- Are the examples correct? Do you need more examples?
- What features did you like most about this manual?

If you find any errors or have any other suggestions for improvement, please indicate the topic, chapter, and page number below:

Please send your comments to:

Forms Documentation Manager
Oracle Corporation
500 Oracle Parkway
Redwood City, CA 94065 U.S.A.
Fax: (415) 506–7200

If you would like a reply, please give your name, address, and telephone number below:

Thank you for helping us improve our documentation.

A32506–2

Forms™ Advanced Techniques

Release 4.5