Oracle ® Applications Developer's Guide

RELEASE 11

April 1998

ORACLE®

Enabling the Information Age™

Oracle® Applications Developer's Guide Release 11

The part number for this guide is A58187–01.

Major Contributors: Anne Carlson, Christopher Lunt, Elizabeth Mitcham, Lisa Nordhagen, Peter Wallack, Millie Wang, Sara Woodhull

Contributors: Steve Carter, Robert Hoexter, John Kucera, Doug McMahon, Anil Mukundan, Tom Morrow, Susan Stratton, Dan Tow

Contents

Preface

This manual contains the coding standards followed by the Oracle Applications development staff. It describes the code needed to implement the Oracle Applications user interface described in the *Oracle Applications User Interface Standards.* It also provides information necessary to help you integrate your Oracle Forms 4.5 forms with Oracle Applications. This preface includes the following topics:

- Documentation Set
- Audience
- Related Publications
- Typographic Conventions
- Your Comments Are Welcome

Documentation Set

The documentation set for creating new applications, forms, and other components that conform to Oracle Applications standards consists of the following documents:

Document	***Part Number***
Oracle Applications Developer's Guide	A58187
Oracle Applications User Interface Standards	A58193
Oracle Applications Installation	varies by platform

Audience

The manuals in the documentation set are written for the application developer and assume familiarity with PL/SQL and Oracle Forms 4.5.

Related Publications

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

As an application designer using Version 4.5 of Forms, you should also be familiar with the documentation for the Oracle8 Server, PL/SQL, and Oracle Forms documentation for your operating system.

Typographic Conventions

This manual uses the following typographic conventions to distinguish important elements from the body of the manual.

Function Keys

Forms function keys are represented by the key name enclosed in square brackets: [Next Item].

For key mappings for your particular keyboard type, refer to the following sources:

- online Forms Show Keys screen (for most keyboards, [Ctrl–K])
- the keypad diagram

  For more information on the keypad diagram, refer to the Forms documentation for your operating system.

Screen Messages

Hint messages and error messages appear in a monotype font:

```
This is a monotype font.
```

Command and Example Syntax

Commands and examples appear in a monotype font, as follows:

Syntax:

```
SET_CANVAS_PROPERTY(canvas_name, property, value);
```

Example:

```
/*
** Built-in: SET_CANVAS_PROPERTY
** Example:  Change the "background color" by setting the
**           canvas color dynamically at runtime to the name
**           of a visual attribute you created.
*/
BEGIN
Set_Canvas_Property('my_main_cnv',VISUAL_ATTRIBUTE,'blue_text');
END;
```

Command and example syntax adhere to the following conventions:

Convention	Explanation
plain monotype	Used for code fragments and examples.
italic monotype	Indicates user–supplied items such as variables, exceptions, and actual parameters.

underlined monotype	Indicates a default parameter. If you indicate no parameter in a parameter set, Forms applies the default parameter.
...	An ellipsis shows that statements or clauses were left out. The ellipsis can appear horizontally as shown, or in vertical format.
/*	A slash and asterisk begin a C–style comment.
*/	An asterisk and slash end a C–style comment.
—	Two consecutive hyphens begin an ANSI–style comment, which extends to the end of the line.
indentation	Indentation helps show structure within code examples, but is not required.

Case Sensitivity

Although neither PL/SQL nor Forms commands are case sensitive (that is, you can enter text in upper or lower case without restriction), in the documentation both upper and lower case are used for ease in reading.

In syntax examples, built–in names are usually shown in all caps; user–defined values are usually shown in lower case.

Syntax: `SET_CANVAS_PROPERTY(canvas_name, property, value);`

Syntax Examples

This example illustrates first how the syntax is presented in this manual, followed by an example of how you actually enter a built–in procedure into your triggers.

Example Syntax: `SET_FORM_PROPERTY(formmodule_name, property, value);`

With Values: `Set_Form_Property('my_form', savepoint_mode, PROPERTY_ON);`

Example Syntax: `SET_TIMER(timer_name, milliseconds, iterate);`

With Values: `Set_Timer('my_timer', 12000, REPEAT);`

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.

Oracle Applications Documentation Manager
Oracle Corporation
500 Oracle Parkway
Redwood City, CA 94065
U.S.A.
FAX: 650–506–7200

CHAPTER

1

Overview of Coding Standards

This chapter describes the general principles on which the Oracle Applications Coding Standards are based, and introduces basic coding standards that apply to all forms.

The following topics are covered:

- Importance of these Standards
- Coding Principles
- Coding with Handlers
- Performance
- The Standard Development Environment
- Shared Objects
- Libraries
- Property Classes
- Visual Attributes
- Overview of Building an Application
- Overall Design Issues to Consider
- Overview of Application Development Steps
- Overview of Form Development Steps

Overview of Coding Standards

Importance of these Standards

The coding standards described in this manual, together with the user interface standards described in the *Oracle Applications User Interface Standards*, are used by Oracle Corporation developers to build Oracle Applications. If you want to build custom application code that integrates with and has the same look and feel as Oracle Applications, you must follow these standards. If you do not follow these standards exactly as they are presented, you may not achieve an acceptable result.

This manual makes no attempt to analyze the consequences of deviating from the standards in particular cases. The libraries and procedures that are packaged with Oracle Applications all assume adherence to these standards. In fact, since the behavior of Oracle Forms, the Oracle Applications standard libraries, and the standards are so tightly linked, a deviation from standards that appears to be minor may in fact have far–reaching and unpredictable results. Therefore, we recommend that when you develop custom application code, you follow the standards exactly as they are described in this manual and in the *Oracle Applications User Interface Standards*.

Coding Principles

Oracle Applications coding standards are guided by the following principles:

- Code must be readable to be maintained
- Tools such as Oracle Forms and PL/SQL are used whenever possible (avoid complex user exits using other coding languages)
- Fast performance on a wide area network (WAN) is critical
- Platform–specific code should be avoided except where absolutely necessary
- Reusable objects should be employed wherever possible

Coding With Handlers

Oracle Applications uses groups of packaged procedures, called handlers, to organize PL/SQL code in forms so that it is easier to develop, maintain, and debug.

In Oracle Forms, code is placed in triggers, which execute the code when that trigger event occurs. Implementing complex logic may require scattering its code across multiple triggers. Because code in triggers is not located in one place, it cannot be written or reviewed comprehensively, making development, maintenance, and debugging more difficult. To determine what code and events affect a particular item, a developer must scan many triggers throughout the form. Code that affects multiple items can be extremely difficult to trace.

To centralize the code so it is easier to develop, maintain, and debug, place the code in packaged procedures and call those procedures from the triggers. Pass the name of the trigger as an argument for the procedure to process. This scheme allows the code for a single business rule to be associated with multiple trigger points, but to reside in a single location.

There are different kinds of procedures for the different kinds of code you write: item handlers, event handlers, table handlers, and business rules. Code resides in these procedures; do not put any code in the triggers other than calls to the procedures.

Item Handlers

An item handler is a PL/SQL procedure that encapsulates all of the code that acts upon an item. Most of the validation, defaulting, and behavior logic for an item is typically in an item handler.

Coding Item Handlers (See page 4 – 19)

Event Handlers

An event handler is a PL/SQL procedure that encapsulates all of the code that acts upon an event. Usually event handlers exist to satisfy requirements of either Oracle Forms or the *Oracle Applications User Interface Standards*, as opposed to particular business requirements for a product.

Coding Event Handlers (See page 4 – 21)

Table Handlers

A table handler encapsulates all of the code that manages interactions between a block and its base table. When an updatable block is based on a view, you must supply procedures to manage the insert, update, lock and delete. Referential integrity checks often require additional procedures. Table handlers may reside on either the client or server, depending on their size and the amount of interaction with the database.

Coding Table Handlers (See page 4 – 22)
Server side versus Client side (See page 4 – 6)

Business Rules

A business rule describes complex data behavior. For example, one business rule is: "A discount cannot be greater than 10% if the current credit rating of the buyer is less than 'Good'." Another business rule is: "A Need–By Date is required if a requisition is made for an inventory item."

A business rule procedure encapsulates all of the code to implement one business rule when the business rule is complex or affects more than one item or event. The business rule procedure is then called by the item or event handlers that are involved in the business rule. If the business rule is simple and affects only one item or event, implement the business rule directly in the item or event handler.

Libraries

Libraries contain reusable client–side code. They enforce these form coding standards by allowing the same code to be used by all forms to enforce specific validation, navigation and cosmetic behaviors and appearances.

Libraries allow code to be written once and used by multiple forms. Additionally, because the executables attach at runtime, they facilitate development and maintenance without being invasive to a form.

Every form requires several standard triggers and procedures to link the form with a library. Many of these triggers and procedures have a default behavior that a developer overrides for specific items or blocks.

Special Triggers in the TEMPLATE form (See page 0–6)

Application–Specific Libraries

Each application is strongly encouraged to create its own libraries. Typically, each application creates a central library that governs behaviors of objects found throughout many of its forms. Additional libraries should be created for each major transaction form to facilitate the following:

- Multiple developers can work on a single module, with some developers coding the actual form and others coding the supporting libraries.
- Shipment and installation of patches to code is vastly simplified if the correction is isolated in a library. Libraries do not require any porting or translation.

All libraries should reside in the $AU_TOP/res/plsql directory (or its equivalent).

Attaching Libraries

Sometimes library attachments can be lost on platforms that have case–sensitive filenames. By Oracle Applications standards, library names must be in all uppercase letters (except for the file extension). However, for forms developed using Microsoft Windows, the library filename may be attached using mixed case letters, making the attachment invalid on case–sensitive platforms such as Unix. If you attach a library to a form in the Oracle Forms Designer on Microsoft Windows, you should avoid using the Browse mechanism to locate the file. Instead, type in just the filename, in uppercase only, with no file extension (for example, CUSTOM). Oracle Forms will then preserve the attachment exactly as you typed it. Note that your attachment should never include a directory path.

Performance

Performance is a critical issue in any application. Applications must avoid overloading the network that connects client and server computers, since often it is network performance that most influences users' perceptions of application performance.

Oracle Applications are designed to minimize network traffic to enable WAN operation. For example, they try to limit network round trips to one per user–distinguishable event by following the following coding standards:

- Base blocks on views that denormalize foreign key information
- Use database stored procedures when extensive SQL is required
- Code all non–SQL logic on the client side where possible
- Cache data on the client side where practical

Views (See page 3 – 7)
Server Side versus Client Side (See page 4 – 6)

Platform–Specific Code

In general, avoid using platform–specific code as much as possible. In some cases however, it is unavoidable. If you provide functionality that is specific to one platform, provide equivalent functionality on other platforms.

You can use the APP_STANDARD.PLATFORM package variable to test for the current user interface system (MSWINDOWS or WEB). Use this value to write platform specific code.

APP_STANDARD Package (See page 26 – 25)

Coding for Web and NCA Compatibility

Following Oracle Applications standards carefully will help ensure that your forms can be deployed on the Web using the Network Computing Architecture (NCA).

You should avoid using the following features in your forms, as they are not applicable in this architecture:

- ActiveX, VBX, OCX, OLE, DDE (Microsoft Windows–specific features that would not be available for a browser running on a Macintosh, for example, and cannot be displayed to users from within the browser)
- Timers (all timers will be treated as timers that fire immediately, so avoid coding logic that relies on a timer having a specific duration)
- WHEN–MOUSE–MOVE, WHEN–MOUSE–ENTER / LEAVE and WHEN–WINDOW–ACTIVATED / DEACTIVATED triggers
- Open File dialog box

  - It would open a file on the applications server, rather than on the client machine (where the browser is) as a user might expect
- Combo boxes
  - Our standards do not use combo boxes anyhow
- Text_IO and HOST built–in routines
  - These would take place on the applications server, rather than on the client machine (where the browser is) as a user might expect

The Standard Development Environment

These coding standards assume that you are developing code in the appropriate Oracle Applications development environment, which includes compatible versions of several products. You can ensure that you have all the correct versions of the Oracle Applications and Oracle Developer/2000 products by installing all products from one set of Oracle Applications Release 11 CD–ROMs.

Run the Oracle Forms executable that contains additional Oracle Application Object Library user exits referred to by the libraries. Both the libraries and the user exits assume a full installation of the Oracle Application Object Library database schema, as they reference tables, views, and packages contained therein.

Apply specific settings to the appropriate environment files for your platform (see the Release 11 *Oracle Applications Installation Manual* for your platform). These settings control properties outside the control of Oracle Forms, such as the appearance of certain Oracle Application Object Library popups.

Mandatory Settings

At form generation time, make sure you designate the character set designed for your language in the NLS_LANG variable in your *oracle.ini* file (or platform equivalent such as Windows NT registry). You should not set USER_NLS_LANG.

You must also set the value of your FORMS45_PATH environment variable in your *oracle.ini* file (or platform equivalent such as Windows NT registry) to include any directory that contains forms, files, or libraries you use to develop and generate your forms. Specifically, you must include a path to the <$AU_TOP>/forms/US directory to be able to find the APPSTAND.fmb file, and a path to the <$AU_TOP>/resource directory to be able to find the Oracle Applications library files you need (where <$AU_TOP> is the appropriate directory path, not the variable).

For any machine on which you generate your forms, set the following environment variable:

- On MS Windows: Add the following line to the [Oracle] section of your *oracle.ini* file (or do the equivalent using the NT Registry):

```
DEVELOPER_NLS_LANG = ”AMERICAN_AMERICA.US7ASCII”
```

- For Unix systems: Place the following command in your environment script

```
setenv DEVELOPER_NLS_LANG AMERICAN_AMERICA.US7ASCII
```

After generating the forms, you may set the language back to the original value.

Oracle Application Object Library for Release 11

Oracle Application Object Library (AOL) for Release 11 includes:

- Starting forms
  – Template form with standard triggers (TEMPLATE)
  – Form containing standard property classes for your runtime platform (APPSTAND)
- PL / SQL libraries
  – Routines for Flexfields, Function security, User Profiles, Message Dictionary (FNDSQF)
  – Standard user interface routines (APPCORE)
  – Routines for Calendar widget (APPDAYPK)
- Development standards
  – *Oracle Applications User Interface Standards*
  – *Oracle Applications Developer's Guide* (this manual)

Shared Objects

These standards rely extensively on the object referencing capabilities of Oracle Forms. These capabilities allow objects to be reused across multiple forms, with changes to the master instance automatically inherited by forms that share the object. Additionally, these shared objects provide flexibility for cross–platform support, allowing Oracle Applications to adhere to the look and feel conventions of the platform they run on.

APPSTAND Form

The APPSTAND form contains the master copy of the shared objects. It contains the following:

- Object group STANDARD_PC_AND_VA, which contains the Visual Attributes and Property Classes required to implement much of the user interface described in the *Oracle Applications User Interface Standards*. A property class exists for almost every item and situation needed for development.

Property Classes (See page 1 – 12)
Setting the Properties of Container Objects: page 5 – 1
Setting the Properties of Widget Objects: page 6 – 1

- Object group STANDARD_TOOLBAR, which contains the windows, canvasses, blocks, and items of the Applications Toolbar. This group also contains other items which are required in all forms but are not necessarily part of the Toolbar.

Pulldown Menus and the Toolbar (See page 10 – 2)

- Object group STANDARD_CALENDAR, which contains the windows, canvasses, blocks, and items of the Applications Calendar.

The Calendar (See page 9 – 23)

- Object group QUERY_FIND, which contains a window, canvas, block, and items used as a starting point for coding a Find Window. This object group is copied into each form, rather than referenced, so that it can be modified.

Query Find Windows (See page 8 – 2)

Warning: Additional objects in the APPSTAND form are for internal use by Oracle Applications only, and their use is not supported. Specifically, the object group STANDARD_FOLDER is not supported.

APPSTAND is altered slightly on each platform to adjust properties such as button height, colors, and Toolbar locations. Application Object Library ships with the correct APPSTAND file in the $AU_TOP/res/US directory (or its equivalent) for the specified platform. You should not be concerned about the APPSTAND form while following these coding standards as its objects are processed automatically. When you move a form to a different platform and regenerate it, it inherits the proper settings for that platform (so long as you have the proper APPSTAND for that platform).

TEMPLATE Form

The TEMPLATE form is the required starting point for all development of new forms. It includes references to many APPSTAND objects, several attached libraries, required triggers, and other objects.

Start developing each new form by copying this file, located in $AU_TOP/forms/US (or your language and platform equivalent), to a local directory and renaming it as appropriate.

Overview of the TEMPLATE Form (See page 0–2)

FNDMENU

The Oracle Applications default menu (with menu entries common to all forms, such as Action, Edit, Query, Help, and so on) is contained in the $AU_TOP/res/US directory (or its equivalent) as the file FNDMENU. You should never modify this file, nor should you create your own menu for your forms.

Standard Libraries

Application Object Library contains four libraries that support the *Oracle Applications User Interface Standards*:

- FNDSQF contains packages and procedures for Message Dictionary, flexfields, profiles, and concurrent processing. It also has various other utilities for navigation, multicurrency, WHO, etc.
- APPCORE contains the packages and procedures that are required of all forms to support the menu, Toolbar, and other required standard behaviors.

Oracle Applications APIs (See page 26 – 1)

- APPDAYPK contains the packages that control the Applications Calendar.

The Calendar (See page 9 – 23)

- APPFLDR contains all of the packages that enable folder blocks.

Warning: Oracle Applications does not support use of the APPFLDR library for custom development.

The TEMPLATE form automatically attaches the FNDSQF, APPCORE and APPDAYPK libraries.

Libraries in the TEMPLATE Form (See page 0–3)

Any code you write within a form that is based on the TEMPLATE form may call any (public) procedure that exists in these libraries. If you code your own library, you will need to attach the necessary libraries to it.

There may be many other libraries attached to any given form that support product–specific functionality, globalization or localization features required by different countries, features required for specific vertical markets, or customizations.

Property Classes

Property classes are sets of attributes that can be applied to almost any Oracle Forms object. The TEMPLATE form automatically contains property classes, via references to APPSTAND, that enforce standard cosmetic appearances and behaviors for all widgets and containers as described in the *Oracle Applications User Interface Standards*.

Property Classes
Oracle Applications User Interface Standards

In addition to enforcing standards, property classes allow a single form to behave differently on each platform, because a single change in the master copy of the property class is automatically inherited by widgets that use that class.

Rules for attaching the property classes to specific objects are discussed in Chapters 5 and 6.

Setting the Properties of Container Objects (See page 5 – 1)
Setting the Properties of Widget Objects (See page 6 – 1)

Do not override any attribute set by a property class unless this manual explicitly states that it is acceptable, or there is a compelling reason to do so. Overriding an inherited attribute is very rarely required.

Application–specific Property Classes, Object Groups and Objects

Each application should take advantage of the referencing capabilities of Oracle Forms to help implement standards for their particular application in the same manner as APPSTAND.

For example, the General Ledger application might have specified standard widths and behaviors for ”Total” fields throughout the application. A GL_TOTAL Property Class, referenced into each form, could set properties such as width, format mask, etc. A General Ledger developer, after referencing in this set of property classes, can then simply apply the GL_TOTAL property class to each item in the form that is a Total field and it inherits its standard appearance and behavior automatically. Entire items or blocks can also be reused.

Further, property classes can be based on other property classes, so the GL_TOTAL class could be based on the standard TEXT_ITEM_ DISPLAY_ONLY class in APPSTAND. Such subclassing allows the application–specific object to inherit changes made within APPSTAND automatically.

Most Oracle Applications products also have a ”standard” form (typically called [Application short name]STAND, such as GLSTAND or BOMSTAND) in the same directory if you install the development versions of those products. These files are used for storing application–specific object groups, property classes, and other objects that are referenced into Oracle Applications forms.

Visual Attributes

All of the visual attributes described in the *Oracle Applications User Interface Standards* are automatically included in the TEMPLATE form via references to APPSTAND. Each visual attribute is associated with a property class or is applied at runtime by APPCORE routines.

For detailed information about the specific color palettes and effects of the visual attributes, see the *Oracle Applications User Interface Standards*.

Visual Attributes
Oracle Applications User Interface Standards

Overview of Building an Application

An application that integrates with Oracle Applications consists of many pieces, including but not limited to forms, concurrent programs and reports, database tables and objects, messages, menus, responsibilities, flexfield definitions, online help, and so on.

Building an application also requires that you consider many overall design issues, such as what platforms and languages your application will run on, what other applications you will integrate with, maintenance issues, and so on.

Overall Design Issues to Consider

When designing your application, you should keep in mind that many Oracle Applications features affect various facets of your application including database objects, forms, concurrent programs, and so on, and including these features should be considered from the beginning of your application design process. These features include but are not limited to:

- Flexfields
- User profiles
- Multiple organizations
- Oracle Workflow integration
- Multiple platform support
- National language support
- Flexible date formats
- Multiple currency support
- Year 2000 support
- CUSTOM library support
- Object naming standards

Overview of Application Development Steps

This is the general process of creating an application that integrates with Oracle Applications.

1. Register your application. See: Applications Window: page 2 – 6.
2. Set up your application directory structures. See: Overview of Setting Up Your Application Framework: page 2 – 2.
3. Modify the appropriate environment files. See: *Oracle Applications Installation Manual*.
4. Register your custom Oracle schema. See: *Oracle Applications System Administrator's Guide.*
5. Include your custom application and Oracle schema in data groups. See: *Oracle Applications System Administrator's Guide.*.
6. Create your application tables and views. See: Tables: page 3 – 2. See: Views: page 3 – 7.
7. Integrate your tables and views with the Oracle Applications APPS schema. See: *Oracle Applications Installation Manual*.
8. Register your flexfields tables. See: Table Registration API: page 3 – 11.
9. Build your application libraries and forms. See: Overview of Form Development Steps: page 1 – 16.
10. Build your application functions and menus. See: Overview of Menus and Function Security: page 0 – 2.
11. Build your application responsibilities. See: *Oracle Applications System Administrator's Guide.*
12. Build concurrent programs and reports. See: Overview of Concurrent Processing: page 15 – 2.
13. Customize Oracle Applications forms if necessary using the CUSTOM library. See: Customizing Oracle Applications with the CUSTOM Library: page 25 – 2.

Overview of Form Development Steps

This is the general process of building a form that integrates with Oracle Applications.

1. Copy the form TEMPLATE and rename it. See: Overview of the TEMPLATE Form: page 0–2.
2. Attach any necessary libraries to your copy of TEMPLATE. TEMPLATE comes with several libraries already attached. See: Overview of the TEMPLATE Form: page 0–2.
3. Create your form blocks, items, LOVs, and other objects and apply appropriate property classes. See: Setting Properties of Container Objects: page 5 – 1. See: Setting Properties of Widget Objects: page 6 – 1.
4. Create your window layout in adherence with the *Oracle Applications User Interface Standards*.
5. Add table handler logic. See: Coding Table Handlers: page 4 – 22.
6. Code logic for window and alternative region control. See: Controlling Window Behavior: page 0 – 2.
7. Add Find windows and / or Row–LOVs and enable Query Find. See: Overview of Query Find: page 8 – 2.
8. Code logic for item relations such as dependent fields. See: Item Relations: page 9 – 2.
9. Code any messages to use Message Dictionary. See: Overview of Message Dictionary: page 12 – 2.
10. Add flexfields logic if necessary. See: Overview of Flexfields: page 14 – 2.
11. Add choices to the Special menu and add logic to modify the default menu and toolbar behavior if necessary. See: Pulldown Menus and the Toolbar: page 10 – 2.
12. Code any other appropriate logic.
13. Test your form by itself.
14. Register your form with Oracle Application Object Library. See: Forms Window: page 11 – 20.
15. Create a form function for your form and register any subfunctions. See: Overview of Menus and Function Security: page 0 – 2.
16. Add your form function to a menu, or create a custom menu. See: Overview of Menus and Function Security: page 0 – 2.

17. Assign your menu to a responsibility and assign your responsibility to a user. See: *Oracle Applications System Administrator's Guide*.
18. Test your form from within Oracle Applications (especially if it uses features such as user profiles or function security).

CHAPTER

2

Setting Up Your Application Framework

This chapter describes what you need to do to set up your application framework, including creating directory structures, registering your application, registering your Oracle schema, and so on.

The following topics are covered:

- Application Directory Structures
- Registering Your Application
- Registering Your Oracle Schema
- Setting Up Your Application and Integrating It with Oracle Applications
- Applications window

Overview of Setting Up Your Application Framework

Oracle Applications and custom applications that integrate with Oracle Applications rely on having their components arranged in a predictable structure. This includes particular directory structures where you place reports, forms, programs and other objects, as well as environment variables and application names that allow Oracle Application Object Library to find your application components.

Definitions

Application

An application, such as Oracle General Ledger or Oracle Inventory, is a functional grouping of forms, programs, menus, libraries, reports, and other objects. Custom applications group together site–specific components such as custom menus, forms, or concurrent programs.

Application Short Name

The application short name is an abbreviated form of your application name used to identify your application in directory and file names and in application code such as PL/SQL routines.

Oracle Schema

Database username used by applications to access the database. Also known as Oracle ID (includes password) or Oracle user.

Environment Variable

An operating system variable that describes an aspect of the environment in which your application runs. For example, you can define an environment variable to specify a directory path.

- $APPL_TOP: An environment variable that denotes the installation directory for Oracle Application Object Library and your other Oracle applications. $APPL_TOP is usually one directory level above each of the product directories (which are often referred to as $PROD_TOP or $PRODUCT_TOP or $<prod>_TOP)

Application Basepath

An environment variable that denotes the directory path to your application–level subdirectories. You include your application basepath in your application environment files and register it with Oracle Application Object Library when you register your application name. Corresponds to the $PRODUCT_TOP directory.

Set Up Your Application Directory Structures

When you develop your application components, you must place them in the appropriate directories so that Oracle Application Object Library can find them. For example, reports written using Oracle Reports are typically placed in a subdirectory called reports, while forms belong in separate subdirectories depending on their territory and language (such as US for American English, D for German, and so on).

The directory structure you use for your application depends on the computer and operating system platform you are using, as well as the configuration of Oracle Applications at your site. For example, you may be using a client–server configuration that uses a Unix server with a Microsoft Windows 95 client, or you may be using a configuration that includes a Unix database server, a Microsoft Windows NT forms server, and Web browsers as clients. These configurations would have different directory setups. See your *Oracle Applications Installation Manual* for directory setup information for your particular platforms and configuration. For a description of the contents and purpose of each of the subdirectories, see your *Oracle Applications Installation Manual.*

Register Your Application

You must register your application name, application short name, application basepath, and application description with Oracle Application Object Library. Oracle Application Object Library uses this information to identify application objects such as responsibilities and forms as belonging to your application.

This identification with your custom application allows Oracle Applications to preserve your application objects and customizations during upgrades. When you register your application, your application receives a unique application ID number that is included in Oracle Application Object Library tables that contain application objects such as responsibilities. This application ID number is not visible in any Oracle Applications form fields.

To reduce the risk that your custom application short name could conflict with a future Oracle Applications short name, we recommend that your custom application short name begins with ”XX”. Such a conflict will not affect application data that is stored using the application ID number (which would never conflict with application IDs used by Oracle Applications products). However, a short name conflict may affect your application code where you use your application short name to identify objects such as messages and concurrent programs (you include the application short name in the code instead of the application ID).

Applications Window (See page 2 – 6)

Modify Your Environment Files

You must add your application basepath variable to the appropriate Oracle Applications environment files. The format and location of these files depends on your operating system and Oracle Applications configuration. See your *Oracle Applications Installation Guide* for information about your environment files.

Set Up and Register Your Oracle Schema

When you build custom forms based on custom tables, typically you place your tables in a custom Oracle schema in the database. You must register your custom schema with Oracle Application Object Library. See your *Oracle Applications Installation Guide* and your *Oracle Applications System Administrator's Guide.*

Create Database Objects and Integrate with APPS Schema

To integrate your application tables with Oracle Applications, you must create the appropriate grants and synonyms in the APPS schema. See your *Oracle Applications Installation Guide* for information about integrating with the APPS schema.

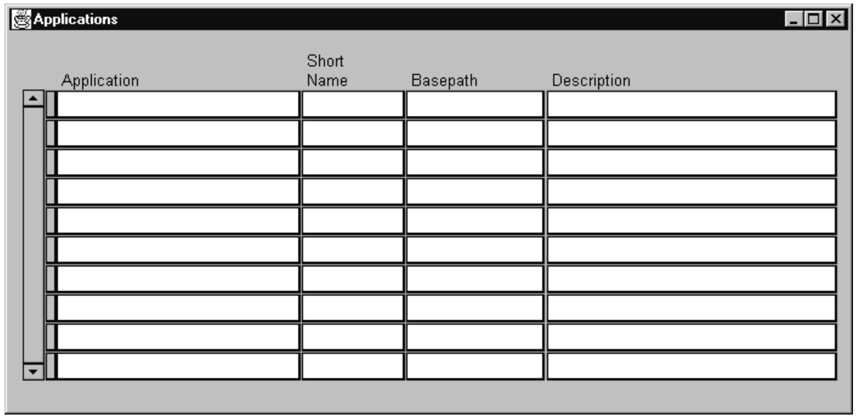
Add Your Application to a Data Group

Oracle Applications products are installed as part of the Standard data group. If you are building a custom application, you should use the Data Groups window to make a copy of the Standard data group and add your application–Oracle ID pair to your new data group. Note that if you have integrated your application tables with the APPS schema, then you would specify APPS as the Oracle ID in the application–Oracle ID pair (instead of the name of your custom schema). See your *Oracle Applications System Administrator's Guide.*

Set Up Concurrent Managers

If your site does not already have a concurrent manager setup appropriate to support your custom application, you may need to have your system administrator set up additional concurrent managers. See your *Oracle Applications System Administrator's Guide.*

Applications Window



When you define a custom application, you supply several pieces of information to Oracle Applications. You must register your application name, application short name, application basepath, and application description with Oracle Application Object Library. Oracle Application Object Library uses this information to identify application objects such as responsibilities and forms as belonging to your application. This identification with your custom application allows Oracle Applications to preserve your application objects and customizations during upgrades. The application basepath tells Oracle Application Object Library where to find the files associated with your custom application.

You can use your custom application to name your custom menus, concurrent programs, custom responsibilities, and many other custom components. For some objects, the application part of the name only ensures uniqueness across Oracle Applications. For other components, the application you choose has an effect on the functionality of your custom object.

Prerequisites

❑ Define an environment variable that translates to your application's basepath (see the *Oracle Applications Installation Guide* for your operating system).

❑ Set up a directory structure for your application (see the *Oracle Applications Installation Manual* for your operating system)

❑ If your application resides in a database other than the database where Oracle Alert resides, you must create a database link.

Applications Block

When you register a custom application, you provide the information Oracle uses to identify it whenever you reference it. Although you can change the name of an application, doing so may cause a change in the application code where you hardcode your application name. For example, if you pass program arguments through the menu that have application name hardcoded, you will also have to update them.

Attention: You should not change the name of any application that you did not develop, as you cannot be sure of the consequences. You should never change the name of any Oracle Applications application, because these applications may contain hardcoded references to the application name.

Application

This user–friendly name appears in lists seen by application users.

Short Name

Oracle Applications use the application short name when identifying forms, menus, concurrent programs and other application components. The short name is stored in hidden fields while the name displays for users.

Your short name should not include spaces. You use an application short name when you request a concurrent process from a form, and when you invoke a subroutine from a menu.

Suggestion: Although your short name can be up to 50 characters, we recommend that you use only four or five characters for ease in maintaining your application and in calling routines that use your short name. To reduce the risk that your custom application short name could conflict with a future Oracle Applications short name, we recommend that your custom application short name begins with ”XX”.

Basepath

Enter the name of an environment variable that represents the top directory of your application’s directory tree. Oracle Applications search specific directories beneath the basepath for your application's executable files and scripts when defining actions that reside in external files.

In general, your application's basepath should be unique so that separate applications do not write to the same directories.

However, you may define custom applications that will be used only for naming your custom responsibilities, menus and other components. In this case, you can use the basepath of the Oracle application that uses the same forms as your application. For example, if you are defining a Custom_GL application, you could use the GL_TOP basepath for your custom application.

See: Development Environment
(*Oracle Applications Installation Manual)*

CHAPTER

3

Building Your Database Objects

This chapter provides you with information you need to build tables, views, and sequences.

The following topics are covered:

- Tracking Data Changes with WHO
- Oracle8 Declarative Constraints
- LONG, LONG RAW, and RAW Datatypes
- Views
- Sequences
- Table Registration API

Tables

This section describes specifications for how to define your tables and the required columns to add. It also covers special data types such as LONG and LONG RAW, and declarative constraints.

Tracking Data Changes with WHO

The WHO feature reports information about who created or updated rows in Oracle Applications tables. Oracle Applications upgrade technology relies on WHO information to detect and preserve customizations.

If you add special WHO columns to your tables and WHO logic to your forms and stored procedures, your users can track changes made to their data. By looking at WHO columns, users can differentiate between changes made by forms and changes made by concurrent programs.

You represent each of the WHO columns as hidden fields in each block of your form (corresponding to the Who columns in each underlying table). Call FND_STANDARD.SET_WHO in PRE–UPDATE and PRE–INSERT to populate these fields.

Adding WHO Columns

The following table lists WHO columns and the sources for the values of those columns. Set the CREATED_BY and CREATION_DATE columns only when you insert a row.

WHO Columns					
Column Name	**Type**	**Null?**	**Foreign Key?**	**Description**	**Value**
CREATED_BY	NUMBER(15)	NOT NULL	FND_ USER	Keeps track of which user created each row	TO_NUMBER (FND_ PROFILE. VALUE ('USER_ID'))
CREATION_ DATE	DATE	NOT NULL		Stores the date on which each row was created	SYSDATE

Column Name	Type	Null?	Foreign Key?	Description	Value
LAST_UPDATED_BY	NUMBER(15)	NOT NULL	FND_USER	Keeps track of who last updated each row	TO_NUMBER (FND_PROFILE.VALUE ('USER_ID'))
LAST_UPDATE_DATE	DATE	NOT NULL		Stores the date on which each row was last updated	SYSDATE
LAST_UPDATE_LOGIN	NUMBER(15)		FND_LOGINS	Provides access to information about the operating system login of the user who last updated each row	TO_NUMBER (FND_PROFILE.VALUE ('LOGIN_ID'))

Table 3 – 1 (Page 2 of 2)

Any table that may be updated by a concurrent program also needs the following columns:

Concurrent Program WHO Columns				
Column Name	**Type**	**Null?**	**Foreign Key to Table?**	**Description**
REQUEST_ID	NUMBER(15)		FND_CONCURRENT_REQUESTS	Keeps track of the concurrent request during which this row was created or updated
PROGRAM_APPLICATION_ID	NUMBER(15)		FND_CONCURRENT_PROGRAMS	With PROGRAM_ID, keeps track of which concurrent program created or updated each row

Table 3 – 2 (Page 1 of 2)

Column Name	Type	Null?	Foreign Key to Table?	Description
PROGRAM_ID	NUMBER(15)		FND_ CONCURRENT_ PROGRAMS	With PROGRAM_ APPLICATION_ID, keeps track of which concurrent program created or updated each row
PROGRAM_ UPDATE_DATE	DATE		PROGRAM_ UPDATE_ DATE	Stores the date on which the concurrent program created or updated the row

Table 3 – 2 (Page 2 of 2)

Use Event Handlers to Code WHO in Your Forms

Some operations that must be done at commit time do not seem designed for a table handler. For example, event handlers are preferred to table handlers for setting WHO information for a record, or determining a sequential number. The logic for these operations may be stored in a PRE_INSERT and / or PRE_UPDATE event handler, which is called from PRE–INSERT and PRE–UPDATE block–level triggers during inserts or updates.

FND_STANDARD: Standard APIs (See page 27 – 8)

Property Classes For WHO Fields

Apply the CREATION_OR_LAST_UPDATE_DATE property class to the form fields CREATION_DATE and LAST_UPDATE_DATE. This property classes sets the correct attributes for these fields, including the data type and width.

WHO Column Misuse

Never use WHO columns to qualify rows for processing. Never depend on WHO columns containing correct information.

In general, you should not attempt to resolve WHO columns to HR_EMPLOYEES; if you must attempt such joins, they must be outer joins.

Tables Without WHO Information

For blocks that are based on a table, but do not have WHO information, disable the menu entry HELP–>ABOUT_THIS_RECORD (all other cases are handled by the default menu control).

Code a block–level WHEN–NEW–BLOCK–INSTANCE trigger (style ”Override”) with these lines:

```
app_standard.event('WHEN-NEW-BLOCK-INSTANCE');
app_special.enable('ABOUT', PROPERTY_OFF);
```

APP_SPECIAL: Menu and Toolbar Control (See page 10 – 9)

Oracle8 Declarative Constraints

This section discusses the declarative constraints Oracle8 permits on tables, and when to use each feature with your Oracle Applications tables.

For the most part, any constraint that is associated with a table should be duplicated in a form so that the user receives immediate feedback if the constraint is violated.

Warning: You should not create additional constraints on Oracle Applications tables at your site, as you may adversely affect Oracle Applications upgrades. If you do create additional constraints, you may need to disable them before upgrading Oracle Applications.

NOT NULL

Use wherever appropriate. Declare the corresponding fields within Oracle Forms as ”Required” = True.

DEFAULT

In general, do not use this feature due to potential locking problems with Oracle Forms. You may be able to use this feature with tables that are not used by forms (for example, those used by batch programs), or tables that contain columns that are not maintained by forms. For example, defaulting column values can make batch programs simpler. Possible default values are SYSDATE, USER, UID, USERENV(), or any constant value.

UNIQUE

Use wherever appropriate. A unique key may contain NULLs, but the key is still required to be unique. The one exception is that you may have any number of rows with NULLS in *all* of the key columns.

In addition, to implement a uniqueness check in a form, create a PL/SQL stored procedure which takes ROWID and the table unique key(s) as its arguments and raises an exception if the key is not unique. Only fields that the user can enter should have a uniqueness check within the form; system–generated unique values should be derived from sequences which are guaranteed to be unique.

Uniqueness Check (See page 9 – 19)

CHECK

Use this feature to check if a column value is valid only in simple cases when the list of valid values is static and short (i.e., 'Y' or 'N').

CHECK provides largely duplicate functionality to database triggers but without the flexibility to call PL/SQL procedures. By using triggers which call PL/SQL procedures instead, you can share constraints with forms and coordinate validation to avoid redundancy.

CHECK does provide the assurance that all rows in the table will pass the constraint successfully, whereas database triggers only validate rows that are inserted/updated/deleted while the trigger is enabled.

This is not usually a concern, since Oracle Applications database triggers should rarely be disabled. Some triggers (such as Alert events) are disabled before an upgrade and re–enabled at the end of the upgrade.

We strongly advise against the use of database triggers.

PRIMARY KEY

Define a Primary Key for all tables.

Cascade Delete and Foreign Key Constraint

Do not use the Declarative Cascade Delete or the Foreign Key Constraint when defining tables. Cascade Delete does not work across distributed databases, so you should program cascade delete logic everywhere it is needed.

To implement a referential integrity check, create a PL/SQL stored procedure which takes the table unique key(s) as its argument(s) and raises an exception if deleting the row would cause a referential integrity error.

Integrity Checking (See page 9 – 19)

LONG, LONG RAW and RAW Datatypes

Avoid creating tables with the LONG, LONG RAW, or RAW datatypes. Within Oracle Forms, you cannot search using wildcards on any column of these types. Use VARCHAR2(2000) columns instead.

Columns Using a Reserved Word

If a table contains a column named with a PL/SQL or an Oracle Forms reserved word, you must create a view over that table that aliases the offending column to a different name. Since this view does not join to other tables, you can still INSERT, UPDATE, and DELETE through it.

Views

In general, complex blocks are based on views while simple setup blocks are based on tables. The advantages to using views include:

- Network traffic is minimized because all foreign keys are denormalized on the server
- You do not need to code any POST–QUERY logic to populate non–database fields
- You do not need to code PRE–QUERY logic to implement query–by–example for non–database fields

You should also base your Lists of Values (LOVs) on views. This allows you to centralize and share LOV definitions. An LOV view is usually simpler than a block view, since it includes fewer denormalized columns, and contains only valid rows of data.

Example LOV (See page 6 – 14)

Define Views To Improve Performance

Whenever performance is an issue and your table has foreign keys, you should define a view to improve performance. Views allow a single SQL statement to process the foreign keys, reducing parses by the server, and reducing network traffic.

Define Views to Promote Modularity

Any object available in the database promotes modularity and reuse because all client or server side code can access it. Views are extremely desirable because:

- They speed development, as developers can build on logic they already encapsulated
- They modularize code, often meaning that a correction or enhancement can be made in a single location
- They reduce network traffic
- They are often useful for reporting or other activities
- They can be easily and centrally patched at a customer site

When Not to Create A View

Avoid creating views that are used by only one SQL statement. Creating a view that is only used by a single procedure increases maintenance load because both the code containing the SQL statement and the view must be maintained.

ROW_ID Is the First Column

The first column your view should select is the ROWID pseudo–column for the root table, and the view should alias it to ROW_ID. Your view should then include all of the columns in the root table, including the WHO columns, and denormalized foreign key information.

Suggestion: You only need to include the ROWID column if an Oracle Forms block is based on this view. The Oracle Forms field corresponding to the ROW_ID pseudo–column should use the ROW_ID property class.

Change Block Key Mode

In Oracle Forms, you need to change the block Key Mode property to Non–Updatable to turn off Oracle Forms default ROWID references for blocks based on views. Specify the primary keys for your view by setting the item level property Primary Key to True.

For example, a view based on the EMP table has the columns ROW_ID, EMPNO, ENAME, DEPTNO, and DNAME. Set the Key Mode property of block EMP_V to Non–Updatable, and set the Primary Key property of EMPNO to True.

If your block is based on a table, the block Key Mode should be Unique.

Code Triggers for Inserting, Updating, Deleting and Locking

When basing a block on a view, you must code ON–INSERT, ON–UPDATE, ON–DELETE, and ON–LOCK triggers to insert, update, delete, and lock the root table instead of the view.

Coding Table Handlers (See page 4 – 22)

Single Table Views

Single table views do not require triggers for inserting, updating, deleting and locking. Set the block Key Mode to Unique. Single table views do not require a ROW_ID column.

Special Characters

Do not use the CHR() function (used to define a character by its ASCII number) on the server side. This causes problems with server–side platforms that use EBCDIC, such as MVS. You should not need to embed tabs or returns in view definitions.

Sequences

This section discusses standards for creating and using sequences.

Create Single Use Sequences

Use each sequence to supply unique ID values for one column of one table.

Do Not Limit the Range of Your Sequences

Do not create sequences that wrap using the CYCLE option or that have a specified MAXVALUE. The total range of sequences is so great that the upper limits realistically are never encountered.

In general, do not design sequences that wrap or have limited ranges.

Use Number Datatypes to Store Sequence Values

Use a NUMBER datatype to store sequence values within PL/SQL.

If you need to handle a sequence generate a sequence value in your C code, do not assume that a sequence–generated value will fit inside a C long variable. The maximum value for an ascending sequence is 10^27, whereas the maximum value for a C signed long integer is 10^9. If 10^9 is not a reasonable limit for your sequence, you may use a double instead of a long integer. Remember that by using a double for your sequence, you may lose some precision on fractional values. If you do not need to do arithmetic, and simply need to fetch your sequence either to print it or store it back, consider retrieving your sequence in a character string.

Do Not Use the FND_UNIQUE_IDENTIFIER_CONTROL Table

Do not rely on the FND_UNIQUE_IDENTIFIER_CONTROL table to supply sequential values. Use a sequence or the sequential numbering package instead. The FND_UNIQUE_IDENTIFIER_CONTROL table is obsolete and should not have any rows for objects in your product.

Additionally, do not create application–specific versions of the FND table to replace the FND_UNIQUE_IDENTIFIER_CONTROL table.

Table Registration API

You register your custom application tables using a PL/SQL routine in the AD_DD package.

Flexfields and Oracle Alert are the only features or products that depend on this information. Therefore you only need to register those tables (and all of their columns) that will be used with flexfields or Oracle Alert. You can also use the AD_DD API to delete the registrations of tables and columns from Oracle Application Object Library tables should you later modify your tables.

If you alter the table later, then you may need to include revised or new calls to the table registration routines. To alter a registration you should first delete the registration, then reregister the table or column. You should delete the column registration first, then the table registration.

You should include calls to the table registration routines in a PL/SQL script. Though you create your tables in your own application schema, you should run the AD_DD procedures against the APPS schema. You must commit your changes for them to take effect.

The AD_DD API does not check for the existence of the registered table or column in the database schema, but only updates the required AOL tables. You must ensure that the tables and columns registered actually exist and have the same format as that defined using the AD_DD API. You need not register views.

Procedures in the AD_DD Package

```
procedure register_table (p_appl_short_name in varchar2,
                          p_tab_name    in varchar2,
                          p_tab_type    in varchar2,
                          p_next_extent in number default 512,
                          p_pct_free    in number default 10,
                          p_pct_used    in number default 70);

procedure register_column (p_appl_short_name in varchar2,
                          p_tab_name   in varchar2,
                          p_col_name   in varchar2,
                          p_col_seq    in number,
                          p_col_type   in varchar2,
                          p_col_width  in number,
                          p_nullable   in varchar2,
                          p_translate  in varchar2,
```

```
                        p_precision  in number default null,
                        p_scale      in number default null);

procedure delete_table  (p_appl_short_name in varchar2,
                        p_tab_name    in varchar2);

procedure delete_column (p_appl_short_name in varchar2,
                        p_tab_name    in varchar2,
                        p_col_name    in varchar2);
```

p_appl_short_name The application short name of the application that owns the table (usually your custom application).

p_tab_name The name of the table (in uppercase letters).

p_tab_type Use 'T' if it is a transaction table (almost all application tables), or 'S' for a "seed data" table (used only by Oracle Applications products).

p_pct_free The percentage of space in each of the table's blocks reserved for future updates to the table (1–99). The sum of p_pct_free and p_pct_used must be less than 100.

p_pct_used Minimum percentage of used space in each data block of the table (1–99). The sum of p_pct_free and p_pct_used must be less than 100.

p_col_name The name of the column (in uppercase letters).

p_col_seq The sequence number of the column in the table (the order in which the column appears in the table definition).

p_col_type The column type ('NUMBER', 'VARCHAR2', 'DATE', etc.).

p_col_width The column size (a number). Use 9 for DATE columns, 38 for NUMBER columns (unless it has a specific width).

p_nullable Use 'N' if the column is mandatory or 'Y' if the column allows null values.

p_translate Use 'Y' if the column values will be translated for an Oracle Applications product release (used only by Oracle Applications products) or 'N' if the values are not translated (most application columns).

p_next_extent The next extent size, in kilobytes. Do not include the 'K'.

p_precision The total number of digits in a number.

p_scale The number of digits to the right of the decimal point in a number.

Example of Using the AD_DD Package

Here is an example of using the AD_DD package to register a flexfield table and its columns:

```
EXECUTE ad_dd.register_table('FND', 'CUST_FLEX_TEST', 'T',
8, 10, 90);

EXECUTE ad_dd.register_column('FND', 'CUST_FLEX_TEST',
'APPLICATION_ID', 1, 'NUMBER', 38, 'N', 'N');
EXECUTE ad_dd.register_column('FND', 'CUST_FLEX_TEST',
'ID_FLEX_CODE', 2, 'VARCHAR2', 30, 'N', 'N');
EXECUTE ad_dd.register_column('FND', 'CUST_FLEX_TEST',
'LAST_UPDATE_DATE', 3, 'DATE', 9, 'N', 'N');
EXECUTE ad_dd.register_column('FND', 'CUST_FLEX_TEST',
'LAST_UPDATED_BY', 4, 'NUMBER', 38, 'N', 'N');
EXECUTE ad_dd.register_column('FND', 'CUST_FLEX_TEST',
'UNIQUE_ID_COLUMN', 5, 'NUMBER', 38, 'N', 'N');
EXECUTE ad_dd.register_column('FND', 'CUST_FLEX_TEST',
'UNIQUE_ID_COLUMN2', 6, 'NUMBER', 38, 'N', 'N');
EXECUTE ad_dd.register_column('FND', 'CUST_FLEX_TEST',
'SET_DEFINING_COLUMN', 7, 'NUMBER', 38, 'N', 'N');
EXECUTE ad_dd.register_column('FND', 'CUST_FLEX_TEST',
'SUMMARY_FLAG', 8, 'VARCHAR2', 1, 'N', 'N');
EXECUTE ad_dd.register_column('FND', 'CUST_FLEX_TEST',
'ENABLED_FLAG', 9, 'VARCHAR2', 1, 'N', 'N');
EXECUTE ad_dd.register_column('FND', 'CUST_FLEX_TEST',
'START_DATE_ACTIVE', 10, 'DATE', 9, 'N', 'N');
EXECUTE ad_dd.register_column('FND', 'CUST_FLEX_TEST',
'END_DATE_ACTIVE', 11, 'DATE', 9, 'N', 'N');
EXECUTE ad_dd.register_column('FND', 'CUST_FLEX_TEST',
'SEGMENT1', 12, 'VARCHAR2', 60, 'Y', 'N');
EXECUTE ad_dd.register_column('FND', 'CUST_FLEX_TEST',
'SEGMENT2', 13, 'VARCHAR2', 60, 'Y', 'N');
EXECUTE ad_dd.register_column('FND', 'CUST_FLEX_TEST',
'SEGMENT3', 14, 'VARCHAR2', 60, 'Y', 'N');
EXECUTE ad_dd.register_column('FND', 'CUST_FLEX_TEST',
```

```
'SEGMENT4', 15, 'VARCHAR2', 60, 'Y', 'N');
EXECUTE ad_dd.register_column('FND', 'CUST_FLEX_TEST',
'SEGMENT5', 16, 'VARCHAR2', 60, 'Y', 'N');
```

CHAPTER

4

Using PL/SQL in Oracle Applications

This chapter provides you with information you need to build a PL/SQL procedure to use with Oracle Applications. It explains the standards you should follow to develop a PL/SQL procedure, where to locate your code (client or server machine), and how to handle exceptions.

The following topics are covered:

- Overview of Building a PL/SQL Procedure
- PL/SQL Procedure Coding Standards
- Replacements for Oracle Forms Built-ins
- Resources
- Triggers

Overview of Using PL/SQL in Applications

You can use PL/SQL procedures as part of an application that you build around Oracle Applications. By following the coding standards, you can create a PL/SQL procedure that integrates seamlessly with your application and with Oracle Applications.

You use PL/SQL to:

- Develop procedural extensions to your forms and reports quickly and easily
- Modularize your application code to speed development and improve maintainability
- Optimize your application code to reduce client/server network traffic and improve overall performance

You can use PL/SQL, Oracle's procedural language extension to SQL, to develop procedural extensions to custom forms and reports you create with Oracle tools.

For example, to develop a form that follows Oracle Applications standards, you organize your form code into PL/SQL business rule procedures, item handlers, event handlers, and table handlers. You put very little PL/SQL code directly into form triggers because those triggers do not represent a logical model; they are simply event points that Oracle Forms provides for invoking procedural code. If you put most of your code in packaged PL/SQL procedures, and then call those procedures from your triggers, you will have modular form code that is easy to develop and maintain.

You may write any PL/SQL procedure that helps you modularize your form code. For example, an item handler, event handler, or business rule procedure may actually consist of several smaller procedures. Be sure to group these smaller procedures into logical packages so their purpose is clear. (There is no special name for these smaller procedures. They are simply PL/SQL procedures.)

You can also use PL/SQL to develop concurrent programs or stored procedures that are called from concurrent programs. Generally, any concurrent program you would have developed as an immediate concurrent program in past releases of Oracle Applications could be developed as a PL/SQL concurrent program. Or, you may develop the main body of your concurrent program in C, but encapsulate any SQL statements issued by your concurrent program in PL/SQL stored procedures.

PL/SQL Stored Procedures (See page 17 – 3)

Definitions

Server–side

Server–side is a term used to describe PL/SQL procedures that are stored in an Oracle database (on the database server). Procedures and functions stored in the database are also referred to as stored procedures and functions, and may also be referred to as being database server–side procedures.

Client–side

Client–side is a term used to describe PL/SQL procedures that run in programs that are clients of the Oracle database, such as Oracle Forms, Oracle Reports, and libraries. Typically your client–side machine is a PC if you are using the SmartClient (client/server) architecture.

If you are using the Network Computing Architecture (NCA), the term "client–side" in this manual usually refers to the forms server (where the forms reside).

"Client–side" in this manual does not typically refer to the "desktop client", which is usually a PC or other desktop machine running a Web browser or an appletviewer.

General PL/SQL Coding Standards

Always Use Packages

PL/SQL procedures should always be defined within packages. Create a package for each block of a form, or other logical grouping of code.

Package Sizes

A client–side (Oracle Forms) PL/SQL program unit's source code and compiled code together must be less than 64K. (A program unit is a package specification or body or stand–alone procedure.) This implies that the source code for a program unit cannot exceed 10K.

If a package exceeds the 10K limit, you can reduce the size of the package by putting private variables and procedures in one or more "private packages." By standard, only the original package should access variables and procedures in a private package. If an individual procedure exceeds the size limit, you should separate the code into two or more procedures.

When an Oracle Forms PL / SQL procedure exceeds the 64K limit, Oracle Forms raises an error at generate time.

Server–side packages and procedures do not have a size limit, but when Oracle Forms refers to a server–side package or procedure, it creates a local stub, which does have a size limit. The size of a package stub depends on the number of procedures in the package and the number and types of arguments each procedure has. Keep the number of procedures in a package less than 25 to avoid exceeding the 10K limit.

Using Field Names in Client–Side PL/SQL Packages

Always specify field names completely by including the block name (that is, BLOCK.FIELD_NAME instead of just FIELD_NAME). If you specify just the field name, Oracle Forms must scan through the entire list of fields for each block in the form to locate your field and check if its name is ambiguous, potentially degrading your form perfomance. If you include the block name, Oracle Forms searches only the fields in that block and stops when it finds a match. Moreover, if you ever add more blocks, your existing code continues to work since you specified your field names unambiguously.

Field Names in Procedure Parameters

Pass field names to procedures and use COPY to update field values instead of using IN OUT or OUT parameters. This method prevents a field from being marked as changed whether or not you actually modify it in your procedure. Any parameter declared as OUT is always written to when the procedure exits normally.

For example, declare a procedure as `test(my_var VARCHAR2 IN)` and call it as `test('block.field')` instead of declaring the procedure as `test(my_var VARCHAR2 IN OUT)` and calling it as `test(:block.field)`.

Explicitly associate the parameter name and value with => when the parameter list is long to improve readability and ensure that you are not ”off” by a parameter.

Using DEFAULT

Use DEFAULT instead of ":=" when declaring default values for your parameters. DEFAULT is more precise because you are defaulting the values; the calling procedure can override the values.

Conversely, use ":=" instead of DEFAULT when declaring values for your constant variables. Using ":=" is more precise because you are assigning the values, not defaulting them; the values cannot be overridden.

Use Object IDs

Any code that changes multiple properties of an object using the SET_<*OBJECT*>_PROPERTY built–in (or the Oracle Application Object Library equivalent) should use object IDs. First use the appropriate FIND_<*OBJECT*> built–in to get the ID, then pass the ID to the SET_<*OBJECT*>_PROPERTY built–in.

You should also consider storing the ID in a package global so that you retrieve it only once while the form is running.

Handling NULL Value Equivalence

Use caution when handling NULL values in PL/SQL. For example, if `a := NULL` and `b := NULL`, the expression `(a = b)` evaluates to FALSE. In any "=" expression where one of the terms is NULL, the whole expression will resolve to FALSE.

For this reason, to check if a value is equal to NULL, you must use the operator "is" instead. If you're comparing two values where either of the values could be equal to NULL, you should write the expression like this: `((a = b) or ((a is null) and (b is null))`

Global Variables

Oracle Forms and PL/SQL support many different types of global variables. The following table lists the characteristics of each, which enables you to select the type most appropriate for your code.

Definitions:

- Oracle Forms Global: a variable in the "global" pseudo–block of a form
- PL/SQL Package Global: a global defined in the specification of a package

- Oracle Forms Parameter: a variable created within the Oracle Forms Designer as a Parameter

See the *Oracle Forms Reference Manual* for a complete description of these variable types.

Behavior	**Oracle Forms Global**	**PL/SQL Package Global**	**Oracle Forms Parameter**
Can be created at Design time		Y	Y
Can be created at runtime	Y		
Accessible across all forms	Y		
Accessible from attached libraries	Y	(1)	Y
Support specific datatypes	(2)	Y	Y
Have declarative defaults			Y
Can be referenced indirectly	Y		Y
Can be specified on command line			Y
Must be erased to recover memory	Y		
Can be used in any Oracle Forms code	Y		Y

Table 4 – 1 (Page 1 of 1)

(1) A package variable defined in a form is not visible to any attached library; a variable defined in an attached library is visible to the form. (An Oracle Forms Global is visible to an attached library)

(2) Always CHAR(255).

Database Server Side versus Client Side

Performance is a critical aspect of any application. Because network round trips are very costly in a typical client–server environment, minimizing the number of round trips is key to ensuring good performance.

You should decide whether your PL/SQL procedures reside on the server or on the client based on whichever results in the fewest number of network round trips. Here are some guidelines:

- Procedures that call Oracle Forms built–ins (more generally, client built–ins) must reside on the client.

- Procedures that reference fields directly, either as :block.field or via NAME_IN / COPY, must reside on the client. You can avoid referencing fields directly by accepting field values or names as parameters to your PL / SQL procedures, which also improves your code's modularity.
- If a procedure contains three or more SQL statements, or becomes very complicated, the procedure usually belongs on the server.
- Procedures that perform no SQL and that need no database access should reside wherever they are needed.

If a procedure is called from the server, it *must* reside on the server. If a procedure is called from both client and server, it should be defined in both places, unless the procedure is very complicated and double maintenance is too costly. In the latter case, the procedure should reside on the server.

Formatting PL/SQL Code

This section contains recommendations for formatting PL / SQL code.

- Within a package, define private variables first, then private procedures, and finally public procedures.
- Always end procedures and packages by following the "end" statement with the procedure or package name to help delineate procedures.
- Indent code logically. Using increments of two spaces provides an easy way to track your nested cases.
- Indent SQL statements as follows:

Example

```
DECLARE
  CURSOR employees IS
    SELECT empno
    FROM   emp
    WHERE  deptno = 10
           AND ename IN ('WASHINGTON', 'MONROE')
           AND mgr = 2701;
```

- Use ”– –” to start comments so that you can easily comment out large portions of code during debugging with ”/* ... */”.
- Indent comments to align with the code being commented.
- When commenting out code, start the comment delimiter in the leftmost column. When the code is clearly no longer needed, remove it entirely.
- Use uppercase and lowercase to improve the readability of your code (PL/SQL is case–insensitive). As a guideline, use uppercase for reserved words and lowercase for everything else.
- Avoid deeply nested IF–THEN–ELSE condition control. Use IF–THEN–ELSIF instead.

Example of Bad Style

```
IF ... THEN ... ELSE
  IF ... THEN ... ELSE
    IF ... THEN ... ELSE
    END IF
  END IF
END IF;
```

Example of Good Style

```
IF ... THEN ...
ELSIF ... THEN ...
ELSIF ... THEN ...
ELSIF ... THEN ...
ELSE ...
END IF;
```

- Only create nested PL/SQL blocks (BEGIN/END pairs) within a procedure when there is specific exception handling you need to trap.

Exception Handling

Errors in Oracle Forms PL/SQL

If a failure occurs in Oracle Forms PL/SQL and you want to stop further processing, use FND_MESSAGE to display an error message, then RAISE FORM_TRIGGER_FAILURE to stop processing:

```
IF (error_condition) THEN
   fnd_message.set_name(appl_short_name,
        message_name);
   fnd_message.error;
   RAISE FORM_TRIGGER_FAILURE;
END IF;
```

Note that RAISE FORM_TRIGGER_FAILURE causes processing to stop quietly. Since there is no error notification, you must display any messages yourself using FND_MESSAGE before raising the exception.

Message Dictionary APIs for PL/SQL Procedures (See page 12 – 14)

Errors in Stored Procedures

If a failure occurs in a stored procedure and you want to stop further processing, use the package procedures FND_MESSAGE.SET_NAME to set a message, and APP_EXCEPTION.RAISE_EXCEPTION to stop processing:

```
IF (error_condition) THEN
  fnd_message.set_name(appl_short_name,
        message_name);
  APP_EXCEPTION.RAISE_EXCEPTION;
END IF;
```

The calling procedure in the form does not need to do anything to handle this stored procedure error. The code in the ON–ERROR trigger of the form automatically detects the stored procedure error and retrieves and displays the message.

Attention: For performance reasons, server side packages should return a return_code for all expected returns, such as no_rows. Only unexpected exceptions should be processed with an exception handler.

Message Dictionary APIs for PL/SQL Procedures (See page 12 – 14)
Special Triggers in the TEMPLATE form (See page 0–6)
APP_EXCEPTION: Exception Processing APIs (See page 26 – 6)

Testing FORM_SUCCESS, FORM_FAILURE and FORM_FATAL

When testing FORM_SUCCESS, FORM_FAILURE, or FORM_FATAL be aware that their values may be changed by a built–in in another trigger that is fired as a result of your built–in. For example, consider the following code:

```
GO_ITEM('emp.empno');
IF FORM_FAILURE THEN
  RAISE FORM_TRIGGER_FAILURE;
END IF;
```

The GO_ITEM causes other triggers to fire, such as WHEN–NEW–ITEM–INSTANCE. Although the GO_ITEM may fail, the last trigger to fire may succeed, meaning that FORM_FAILURE is false. The following example avoids this problem.

```
GO_ITEM('EMP.EMPNO');
IF :SYSTEM.CURSOR_ITEM != 'EMP.EMPNO' THEN
  –– No need to show an error, because Oracle Forms
  –– must have already reported an error due to
  –– some other condition that caused the GO_ITEM
  –– to fail.
  RAISE FORM_TRIGGER_FAILURE;
END IF;
```

See the *Oracle Forms Reference Manual* for other techniques to trap the failure of each built–in.

Avoid RAISE_APPLICATION_ERROR

Do not use RAISE_APPLICATION_ERROR. It conflicts with the scheme used to process server side exceptions.

Message Dictionary APIs for PL/SQL Procedures (See page 12 – 14)

Using PL/SQL With Date Fields

When you use a PL/SQL variable to hold the value from an Oracle Forms DATE or DATETIME field, you can access that value using the function NAME_IN as shown in the example below:

```
x_date_example := TO_DATE(NAME_IN('block.datetime_field'),
                              'DD-MON-YYYY HH24:MI:SS');
```

The NAME_IN function returns all values as CHAR. Thus when dealing with a DATE field, you must explicitly supply a mask to convert from a DATE format to a CHAR. However, Oracle Forms has an 'internal' representation and a 'displayed' representation for dates. When you use NAME_IN, it is accessing the internal representation. Furthermore, Oracle Forms only uses the following masks when accessing dates with NAME_IN:

DATE fields:	DD–MON–YYYY
DATETIME fields:	DD–MON–YYYY HH24:MI:SS

This mask is used internally only to convert from DATE to CHAR; it is not affected by, nor does it affect, what the user sees. For this reason, there is not an issue concerning what date mask to use if translation is a concern.

If a DATE field has a mask of 'MM/DD/YYYY', causing the user to see something like 2/13/1995, internally you still access it with the mask 'DD–MON–YYYY'. You will typically assign it to a DATE variable, so the internal mask does not cause a concern.

If you intend to assign a DATE field to a CHAR variable and manipulate it as a CHAR, then you may have a translation issue. In that case, you should first assign it to a DATE variable, then assign it to the CHAR variable with a translatable mask such as 'DD/MM/YYYY'.

Copying Between Date Fields

You cannot directly copy a hardcoded date value into a field:

```
copy('01-FEB-1900', 'bar.lamb');
```

The "FEB" part varies across the different languages, so a direct copy is infeasible. Instead, you may call:

```
app_item.copy_date('01-02-1900', 'bar.lamb');
```

This routine does the copy in this way:

```
copy(to_char(to_date('01-01-1900', 'DD-MM-YYYY'),
             'DD-MON-YYYY'), 'bar.lamb');
```

The only format that the NAME_IN and COPY functions accept are DD–MON–YYYY. Cast all date values to that mask, since these functions process everything as if they are CHAR values.

SYSDATE and USER

Instead of the Oracle Forms built–in routines SYSDATE and USER, use the Applications functions:

```
FND_STANDARD.SYSTEM_DATE  return DATE;
FND_STANDARD.USER   return VARCHAR2;
```

These functions behave identically to the built–ins, but are more efficient since they use information already cached elsewhere.

Use these FND_STANDARD functions in Oracle Forms PL/SQL code only; you can use the Oracle Forms built–ins in SQL statements, $$DBDATE$$ defaulting or in stored procedures.

- Minimize references to SYSDATE within client–side PL/SQL. Each reference is translated to a SQL statement and causes a round–trip to the server.
- Time is included in SYSDATE and FND_STANDARD.SYSTEM_DATE by default. Include the time for creation dates and last updated dates. If you do not wish to include the time in the date, you must explicitly truncate it:

  ```
  :BLOCK.DATE_FIELD := TRUNC(FND_STANDARD.SYSTEM_DATE);
  ```

  Truncate the time for start dates and end dates that enable/disable data.
- Use $$DBDATE$$ to default a date value on a new record.

SQL Coding Guidelines

Follow these guidelines for all SQL that you code:

- Use ”select from DUAL” instead of ”select from SYS.DUAL”. Do not use SYSTEM.DUAL.
- All SELECT statements should use an explicit cursor. Implicit SELECT statements actually cause 2 fetches to execute: one to get the data, and one to check for the TOO_MANY_ROWS exception. You can avoid this by FETCHing just a single record from an explicit cursor.
- If you want to SELECT into a procedure parameter, declare the parameter as IN OUT, whether or not you reference the parameter value, unless the parameter is a field.
- A single–row SELECT that returns no rows raises the exception NO_DATA_FOUND. An INSERT, UPDATE, or DELETE that affects no rows does not raise an exception. You need to explicitly check the value of SQL%NOTFOUND if no rows is an error.
- To handle NO_DATA_FOUND exceptions, write an exception handler. Do not code COUNT statements to detect the existence of rows unless that is your only concern.
- When checking the value of a field or PL/SQL variable against a literal, do the check in PL/SQL code, not in a WHERE clause. You may be able to avoid doing the SQL altogether.
- Do not check for errors due to database integrity problems. For example, if a correct database would have a table SYS.DUAL with exactly one row in it, you do not need to check if SYS.DUAL has zero or more than one row or if SYS.DUAL exists.

Triggers in Forms

Follow these general rules for triggers in your forms.

Execution Style

The ’Execution Style’ for all block or field level triggers should either be Override or Before. In general, use style Before, since usually the form–level version of the trigger should also fire. The exception is if you have a flexfield call in the form–level POST–QUERY trigger, but you reset the query status of the block in the block level POST–QUERY.

In that case, the block–level POST–QUERY should use Execution Style After.

Special Triggers in the TEMPLATE form (See page 0–6)

KEY– Trigger Properties

Set the ”Show Keys” property to True for all KEY– triggers you code, except those that you are disabling (which should have ”Show Keys” set to False). Always set the ”Show Keys Description” property to NULL.

WHEN–CREATE–RECORD in Dynamic Query–Only Mode

The WHEN–CREATE–RECORD trigger fires even when the block does not allow inserts. You may need to check if the block allows insert if you have logic in this trigger and your block may dynamically have insert–allowed ”FALSE”:

```
IF GET_ITEM_PROPERTY('<BLOCK>', INSERT_ALLOWED) = FALSE THEN
   null;
ELSE
   <your logic here>;
END IF;
```

Resources

On the PC there is a limit to the number of real widgets available simultaneously (text items and display items are not real Windows widgets, as Oracle Forms creates these items). Every check box, list item, and object group in your form consumes these resources.

If a real widget is on a hidden canvas, the resources it consumes are freed. You can free resources by explicitly hiding a canvas that does not appear on the screen. Also, any canvas set with a display property of FALSE in the Oracle Forms Designer does not consume resources for itself or its widgets until the canvas is visited or the canvas is programmatically displayed.

Remember that Oracle Forms navigates to the first enterable item at startup time, which creates the canvas and all its widgets for the First Navigation Block.

Checking Resource Availability

To check the availability of MS Windows resources before performing some action, use the following utility:

```
if get_application_property(USER_INTERFACE) =
                            'MSWINDOWS' then
   if (FND_UTILITIES.RESOURCES_LOW) then
     FND_MESSAGE.SET_NAME('FND', 'RESOURCES_LOW');
      if (FND_MESSAGE.QUESTION('Do Not Open', 'Open',
                               '', 1) =1) then
        raise FORM_TRIGGER_FAILURE;
     end if;
   end if;
end if;
```

Replacements for Oracle Forms Built–ins

These standards require that certain built–ins be avoided entirely, or ”wrapper” routines be called in their place. For many built–ins, there are multiple methods of invocation. You can call the built–in directly, giving you the standard forms behavior. For some built–ins, there are standard Oracle Applications behaviors, which you invoke by calling APP_STANDARD.EVENT.

Many of these built–ins have a key and a KEY– trigger associated with them. If there is any additional logic which has been added to the KEY– trigger that you want to take advantage of, you can invoke the trigger by using the DO_KEY built–in. This is the same result you would get if the user pressed the associated key.

You should routinely use the DO_KEY built–in. The only reason to bypass the KEY– trigger is if you need to avoid the additional code that would fire.

Do Not Use CALL_FORM

Do not use this Oracle Forms built–in:

CALL_FORM This built–in is incompatible with OPEN_FORM, which is used by Oracle Applications routines.

You should use FND_FUNCTION.EXECUTE instead of either CALL_FORM or OPEN_FORM whenever you need to open a form programatically. Using FND_FUNCTION.EXECUTE allows you to open forms without bypassing Oracle Applications security, and takes care of finding the correct directory path for the form.

Function Security APIs for PL/SQL Procedures (See page 11 – 16)

Oracle Forms Built–In With APPCORE Replacements

These Oracle Forms built–ins have equivalent APPCORE routines that provide additional functionality:

EXIT_FORM The Oracle Applications forms have special exit processing. Do not call EXIT_FORM directly; always call `do_key('EXIT_FORM')`.

To exit the entire Oracle Applications suite, first call:

```
copy('Y','GLOBAL.APPCORE_EXIT_FLAG');
```

Then call:

```
do_key('exit_form');
```

SET_ITEM_PROPERTY Replace with APP_ITEM_PROPERTY.SET_PROPERTY and APP_ITEM_PROPERTY.SET_VISUAL_ATTRIBUTE. These APPCORE routines set the properties in the Oracle Applications standard way and change the propagation behavior. Some properties use the native Oracle Forms SET_ITEM_PROPERTY. For a complete list of properties that APP_ITEM_PROPERTY.SET_PROPERTY covers, see the documentation for that routine.

APP_ITEM_PROPERTY: Individual Property Utilities (See page 26 – 19)

GET_ITEM_PROPERTY Use APP_ITEM_PROPERTY.GET_PROPERTY when getting Oracle Applications specific properties. Use the Oracle Forms built–in when setting or getting other properties.

OPEN_FORM Use FND_FUNCTION.EXECUTE. This routine is necessary for function security.

Both OPEN_FORM and FND_FUNCTION.EXECUTE cause the POST–RECORD and POST–BLOCK triggers to fire.

CLEAR_FORM Use `do_key('clear_form')`. This routine raises the exception FORM_TRIGGER_FAILURE if there is an invalid record.

You may use this built–in without "do_key" to avoid the additional functionality that comes from going through the trigger.

COMMIT Use `do_key('commit_form')`. This routine raises the exception FORM_TRIGGER_FAILURE if there is an invalid record.

You may use this built–in without "do_key" to avoid the additional functionality that comes from going through the trigger.

EDIT_FIELD/ EDIT_ TEXTITEM

Use `do_key('edit_field')`. This routine raises the calendar when the current item is a date.

You may use this built–in without ”do_key” to avoid the additional functionality that comes from going through the trigger.

VALIDATE

Use APP_STANDARD.APP_VALIDATE instead. This routine navigates to any item that causes navigation failure.

You may use this built–in without ”do_key” to avoid the additional functionality that comes from going through the trigger.

Warning: APP_STANDARD.APP_VALIDATE requires that you follow the button coding standards.

APP_STANDARD Package (See page 26 – 25)
Buttons (See page 6 – 10)

Coding Item, Event and Table Handlers

Developers call handlers from triggers to execute all the code necessary to validate an item or to ensure the correct behavior in a particular situation.

Handlers serve to centralize the code so it is easier to read and work with. A typical form has a package for each block, and a package for the form itself. Place code in procedures within these packages and call the procedures (handlers) from the associated triggers. When a handler involves multiple blocks or responds to form–level triggers, place it in the form package.

There are different kinds of procedures for the different kinds of code, such as item handlers, event handlers, and table handlers. Most code resides in these procedures, and other than calls to them, you should keep code in the triggers to a minimum.

Coding Item Handlers

Item handlers are procedures that contain all the logic used for validating a particular item. An item handler package contains all the procedures for validating the items in a block or form.

The packages are usually named after their block or form, while the procedures are named after their particular item. For example, the block EMP includes the items EMPNO, ENAME, and JOB. The corresponding package EMP contains procedures named EMPNO, ENAME, and JOB, making it easy to locate the code associated with a particular item.

An item handler always takes one parameter named EVENT, type VARCHAR2, which is usually the name of the trigger calling the item handler.

Common EVENT Arguments for Item Handlers

The common event points and associated logic are:

PRE–RECORD Reset item attributes for the new record. Typically used for APPCORE routines that enable and disable dependent fields. You can use WHEN–NEW–RECORD–INSTANCE for some cases where you need to use restricted Oracle Forms built–in routines or perform navigation or commits.

INIT Initialize the item.

VALIDATE Validate the item and set dynamic item attributes.

The INIT Event

INIT is short for ”Initialize” and is a directive to the item handler to initialize the item. INIT tells the item handler to examine current conditions in the form and reset its item’s default value and dynamic attributes as necessary. This event is passed by other handlers and is expected by many APPCORE routines.

The most common case is when an item depends on another item. Whenever the master item changes – in WHEN–VALIDATE–ITEM in the master’s item handler – the dependent’s item handler is called with the INIT event.

When a condition for a master item changes, you typically must cascade the event INIT down to other dependent items.

The VALIDATE Event

This pseudo–event is used with many APPCORE routines where the item should be validated. Use this event instead of WHEN–VALIDATE–ITEM, WHEN–CHECKBOX–CHANGED, WHEN–LIST–CHANGED, or WHEN–RADIO–CHANGED (any of which could also be used). You can write your own item handler routines to expect either the VALIDATE event or the trigger names.

Item Handler Format

A typical item handler looks like this:

```
procedure ITEM_NAME(event VARCHAR2) IS
  IF (event = ’WHEN–VALIDATE–ITEM’) THEN
      –– validate the item
  ELSIF (event = ’INIT’) THEN
      –– initialize this dependent item
  ELSIF (event in (’PRE–RECORD’, ’POST–QUERY’)) THEN
      –– etc.
  ELSE fnd_message.debug(’Invalid event passed to item_name: ’ ||
EVENT);
  END IF;
END ITEM_NAME;
```

Suggestion: Remember that writing an item handler is not the whole process; you also must code a trigger for each event that the procedure handles and call the item handler. If what you coded is not happening, the first thing to check is whether you coded the trigger to call your new item handler.

Coding Event Handlers

Event handlers encapsulate logic that pertains to multiple items where it is easier to centralize the code around an event rather than around individual item behavior. You, the developer, determine when an event handler is easier to read than a set of item handlers.

Very complex cross–item behaviors belong in the event handler, while very simple single item behaviors belong in the item handlers. You can call item handlers from event handlers.

For example, you may code an event handler to populate many items on POST–QUERY. Rather than writing item handlers for each of the items, you could encapsulate all of the logic in a single event handler.

Since an event handler handles only one event, it does not need an EVENT parameter. In fact, it should not take any parameters.

Event handlers are named after the triggers, replacing dashes with underscores (for example, the PRE–QUERY event handler is PRE_QUERY).

Common Event Handlers

PRE_QUERY	Populates items with values needed to retrieve the appropriate records.
POST_QUERY	Populates non–base table items.
WHEN_CREATE_RECORD	Populates default values (when using the default value property is insufficient)
WHEN_VALIDATE_RECORD	Validates complex inter–item relationships

Coding Table Handlers

A table handler is a server–side or client–side package that provides an API to a table. Table handlers are used to insert, update, delete, or lock a record, or to check if a record in another table references a record in this table.

Since most of the forms in Oracle Applications are based on views, these table handlers are necessary to handle interactions with the tables underneath the views.

Warning: Change the block Key Mode from the default value "Unique Key" to "Non–Updatable Key" when the block is based on a multi–table view. Specify your primary key items by setting "Primary Key" to True in the items' property sheets.

Table handlers contain some or all of the following procedures:

CHECK_ UNIQUE	Check for duplicate values on unique columns.
CHECK_ REFERENCES	Check for referential integrity
INSERT_ROW	Insert a row in the table
UPDATE_ROW	Update a row in the table
DELETE_ROW	Delete a row from the table
LOCK_ROW	Lock a row in the table

INSERT_ROW, UPDATE_ROW, DELETE_ROW, and LOCK_ROW are commonly used to replace default Oracle Forms transaction processing in the ON–INSERT, ON–UPDATE, ON–DELETE, and ON–LOCK triggers.

In the INSERT_ROW table handler procedure, if a primary key column is allowed to be NULL, remember to add "OR (primary_key IS NULL AND X_col IS NULL)" to the SELECT ROWID statement's WHERE clause.

In the LOCK_ROW table handler procedure, if a column is not allowed to be NULL, remove the "OR (RECINFO.col IS NULL AND X_col IS NULL)" condition from the IF statement.

Also, since Oracle Forms strips trailing spaces from queried field values, normal row locking strips trailing spaces from the database values before comparison. Since the example LOCK_ROW stored procedure does not strip trailing spaces, comparison for this (rare) case always fails. You may use RTRIM to strip trailing spaces if necessary.

Example Client–Side Table Handler (See page 4 – 23)
Example Server–Side Table Handler (See page 4 – 26)

Acting on a Second Table

To perform an action on another table, call that table's appropriate handler procedure rather than performing the action directly.

For example, to perform a cascade DELETE, call the detail table's DELETE_ROWS table handler (which accepts the master primary key as a parameter) instead of performing the DELETE directly in the master table's DELETE_ROW table handler.

Example Client–Side Table Handler

The following is an example of a client–side table handler that provides INSERT_ROW, UPDATE_ROW, DELETE_ROW, and LOCK_ROW procedures for the EMP table. You code the client–side table handler directly into your form.

Package spec you would code for your EMP block

```
PACKAGE EMP IS
  PROCEDURE Insert_Row;
  PROCEDURE Lock_Row;
  PROCEDURE Update_Row;
  PROCEDURE Delete_Row;
END EMP;
```

Package body you would code for your EMP block

```
PACKAGE BODY EMP IS

  PROCEDURE Insert_Row IS
    CURSOR C IS SELECT rowid FROM EMP
                 WHERE empno = :EMP.Empno;
  BEGIN
    INSERT INTO EMP(
              empno,
              ename,
              job,
```

```
             mgr,
             hiredate,
             sal,
             comm,
             deptno
            ) VALUES (
             :EMP.Empno,
             :EMP.Ename,
             :EMP.Job,
             :EMP.Mgr,
             :EMP.Hiredate,
             :EMP.Sal,
             :EMP.Comm,
             :EMP.Deptno
            );
  OPEN C;
  FETCH C INTO :EMP.Row_Id;
  if (C%NOTFOUND) then
    CLOSE C;
    Raise NO_DATA_FOUND;
  end if;
  CLOSE C;
END Insert_Row;

PROCEDURE Lock_Row IS
  Counter NUMBER;
  CURSOR C IS
      SELECT empno,
             ename,
             job,
             mgr,
             hiredate,
             sal,
             comm,
             deptno
      FROM   EMP
      WHERE  rowid = :EMP.Row_Id
      FOR UPDATE of Empno NOWAIT;
  Recinfo C%ROWTYPE;
BEGIN
  Counter := 0;
  LOOP
    BEGIN
      Counter := Counter + 1;
```

```
  OPEN C;
  FETCH C INTO Recinfo;
  if (C%NOTFOUND) then
    CLOSE C;
    FND_MESSAGE.Set_Name('FND',
          'FORM_RECORD_DELETED');
    FND_MESSAGE.Error;
    Raise FORM_TRIGGER_FAILURE;
  end if;
  CLOSE C;
  if (
       (Recinfo.empno =  :EMP.Empno)
     AND (   (Recinfo.ename =  :EMP.Ename)
          OR (    (Recinfo.ename IS NULL)
              AND (:EMP.Ename IS NULL)))
     AND (   (Recinfo.job =  :EMP.Job)
          OR (    (Recinfo.job IS NULL)
              AND (:EMP.Job IS NULL)))
     AND (   (Recinfo.mgr =  :EMP.Mgr)
          OR (    (Recinfo.mgr IS NULL)
              AND (:EMP.Mgr IS NULL)))
     AND (   (Recinfo.hiredate =  :EMP.Hiredate)
          OR (    (Recinfo.hiredate IS NULL)
              AND (:EMP.Hiredate IS NULL)))
     AND (   (Recinfo.sal =  :EMP.Sal)
          OR (    (Recinfo.sal IS NULL)
              AND (:EMP.Sal IS NULL)))
     AND (   (Recinfo.comm =  :EMP.Comm)
          OR (    (Recinfo.comm IS NULL)
              AND (:EMP.Comm IS NULL)))
     AND (Recinfo.deptno =  :EMP.Deptno)
    ) then
    return;
  else
    FND_MESSAGE.Set_Name('FND',
               'FORM_RECORD_CHANGED');
    FND_MESSAGE.Error;
    Raise FORM_TRIGGER_FAILURE;
  end if;
EXCEPTION
  When APP_EXCEPTIONS.RECORD_LOCK_EXCEPTION then
    IF (C% ISOPEN) THEN
       close C;
     END IF;
```

```
        APP_EXCEPTION.Record_Lock_Error(Counter);
      END;
    end LOOP;
  END Lock_Row;

  PROCEDURE Update_Row IS
  BEGIN
    UPDATE EMP
    SET
       empno                           =     :EMP.Empno,
       ename                           =     :EMP.Ename,
       job                             =     :EMP.Job,
       mgr                             =     :EMP.Mgr,
       hiredate                        =     :EMP.Hiredate,
       sal                             =     :EMP.Sal,
       comm                            =     :EMP.Comm,
       deptno                          =     :EMP.Deptno
    WHERE rowid = :EMP.Row_Id;
    if (SQL%NOTFOUND) then
      Raise NO_DATA_FOUND;
    end if;
  END Update_Row;

  PROCEDURE Delete_Row IS
  BEGIN
    DELETE FROM EMP
    WHERE rowid = :EMP.Row_Id;
    if (SQL%NOTFOUND) then
      Raise NO_DATA_FOUND;
    end if;
  END Delete_Row;

END EMP;
```

Example Server–Side Table Handler

The following is an example of a server–side table handler that provides INSERT_ROW, UPDATE_ROW, DELETE_ROW, and LOCK_ROW procedures for the EMP table. Your handler consists of a package in your form and a server–side package in the database. The package in your form calls the server–side package and passes all of the field values as arguments.

Package spec you would code in your form for your EMP block

```
PACKAGE EMP IS
  PROCEDURE Insert_Row;
  PROCEDURE Update_Row;
  PROCEDURE Lock_Row;
  PROCEDURE Delete_Row;
END EMP;
```

Package body you would code in your form for your EMP block

```
PACKAGE BODY EMP IS
PROCEDURE Insert_Row IS
BEGIN
  EMP_PKG.Insert_Row(
      X_Rowid                => :EMP.Row_Id,
      X_Empno                => :EMP.Empno,
      X_Ename                => :EMP.Ename,
      X_Job                  => :EMP.Job,
      X_Mgr                  => :EMP.Mgr,
      X_Hiredate             => :EMP.Hiredate,
      X_Sal                  => :EMP.Sal,
      X_Comm                 => :EMP.Comm,
      X_Deptno               => :EMP.Deptno);
END Insert_Row;

PROCEDURE Update_Row IS
BEGIN
  EMP_PKG.Update_Row(
      X_Rowid                => :EMP.Row_Id,
      X_Empno                => :EMP.Empno,
      X_Ename                => :EMP.Ename,
      X_Job                  => :EMP.Job,
      X_Mgr                  => :EMP.Mgr,
      X_Hiredate             => :EMP.Hiredate,
      X_Sal                  => :EMP.Sal,
      X_Comm                 => :EMP.Comm,
      X_Deptno               => :EMP.Deptno);
END Update_Row;

PROCEDURE Delete_Row IS
BEGIN
  EMP_PKG.Delete_Row(:EMP.Row_Id);
```

```
END Delete_Row;

PROCEDURE Lock_Row IS
  Counter    Number;
BEGIN
  Counter := 0;
  LOOP
    BEGIN
      Counter := Counter + 1;
      EMP_PKG.Lock_Row(
          X_Rowid                => :EMP.Row_Id,
          X_Empno                => :EMP.Empno,
          X_Ename                => :EMP.Ename,
          X_Job                  => :EMP.Job,
          X_Mgr                  => :EMP.Mgr,
          X_Hiredate             => :EMP.Hiredate,
          X_Sal                  => :EMP.Sal,
          X_Comm                 => :EMP.Comm,
          X_Deptno               => :EMP.Deptno);
      return;
    EXCEPTION
      When APP_EXCEPTIONS.RECORD_LOCK_EXCEPTION then
        APP_EXCEPTION.Record_Lock_Error(Counter);
    END;
  end LOOP;
END Lock_Row;
END EMP;
```

Package spec for the server–side table handler (SQL script)

```
SET VERIFY OFF
DEFINE PACKAGE_NAME=”EMP_PKG”
WHENEVER SQLERROR EXIT FAILURE ROLLBACK;
CREATE or REPLACE PACKAGE &PACKAGE_NAME as

/* Put any header information (such as $Header$) here.
It must be written within the package definition so that the
 header information will be available in the package itself.
 This makes it easier to identify package versions during
 upgrades. */

  PROCEDURE Insert_Row(X_Rowid     IN OUT VARCHAR2,
                       X_Empno     NUMBER,
                       X_Ename     VARCHAR2,
```

```
                         X_Job       VARCHAR2,
                         X_Mgr       NUMBER,
                         X_Hiredate  DATE,
                         X_Sal       NUMBER,
                         X_Comm      NUMBER,
                         X_Deptno    NUMBER
                        );

  PROCEDURE Lock_Row(X_Rowid       VARCHAR2,
                     X_Empno       NUMBER,
                     X_Ename       VARCHAR2,
                     X_Job         VARCHAR2,
                     X_Mgr         NUMBER,
                     X_Hiredate    DATE,
                     X_Sal         NUMBER,
                     X_Comm        NUMBER,
                     X_Deptno      NUMBER
                    );

  PROCEDURE Update_Row(X_Rowid     VARCHAR2,
                       X_Empno     NUMBER,
                       X_Ename     VARCHAR2,
                       X_Job       VARCHAR2,
                       X_Mgr       NUMBER,
                       X_Hiredate  DATE,
                       X_Sal       NUMBER,
                       X_Comm      NUMBER,
                       X_Deptno    NUMBER
                      );

  PROCEDURE Delete_Row(X_Rowid VARCHAR2);
END &PACKAGE_NAME;
/
show errors package &PACKAGE_NAME
SELECT to_date('SQLERROR') FROM user_errors
WHERE  name = '&PACKAGE_NAME'
AND    type = 'PACKAGE'
/
commit;
exit;
```

Package body for the server–side table handler (SQL script)

```
SET VERIFY OFF
DEFINE PACKAGE_NAME=”EMP_PKG”
WHENEVER SQLERROR EXIT FAILURE ROLLBACK;
CREATE or REPLACE PACKAGE BODY &PACKAGE_NAME as

/* Put any header information (such as $Header$) here.
 It must be written within the package definition so the
 header information is available in the package itself.
 This makes it easier to identify package versions during
 upgrades. */

  PROCEDURE Insert_Row(X_Rowid     IN OUT VARCHAR2,
                       X_Empno     NUMBER,
                       X_Ename     VARCHAR2,
                       X_Job       VARCHAR2,
                       X_Mgr       NUMBER,
                       X_Hiredate  DATE,
                       X_Sal       NUMBER,
                       X_Comm      NUMBER,
                       X_Deptno    NUMBER
  ) IS
    CURSOR C IS SELECT rowid FROM emp
                 WHERE empno = X_Empno;
   BEGIN
       INSERT INTO emp(
              empno,
              ename,
              job,
              mgr,
              hiredate,
              sal,
              comm,
              deptno
             ) VALUES (
              X_Empno,
              X_Ename,
              X_Job,
              X_Mgr,
              X_Hiredate,
              X_Sal,
              X_Comm,
```

```
             X_Deptno
            );
  OPEN C;
  FETCH C INTO X_Rowid;
  if (C%NOTFOUND) then
    CLOSE C;
    Raise NO_DATA_FOUND;
  end if;
  CLOSE C;
END Insert_Row;

PROCEDURE Lock_Row(X_Rowid          VARCHAR2,
                   X_Empno          NUMBER,
                   X_Ename          VARCHAR2,
                   X_Job            VARCHAR2,
                   X_Mgr            NUMBER,
                   X_Hiredate       DATE,
                   X_Sal            NUMBER,
                   X_Comm           NUMBER,
                   X_Deptno         NUMBER
) IS
  CURSOR C IS
      SELECT *
      FROM   emp
      WHERE  rowid = X_Rowid
      FOR UPDATE of Empno NOWAIT;
  Recinfo C%ROWTYPE;
BEGIN
  OPEN C;
  FETCH C INTO Recinfo;
  if (C%NOTFOUND) then
    CLOSE C;
    FND_MESSAGE.Set_Name('FND', 'FORM_RECORD_DELETED');
    APP_EXCEPTION.Raise_Exception;
  end if;
  CLOSE C;
  if (
             (Recinfo.empno =  X_Empno)
         AND (   (Recinfo.ename =  X_Ename)
              OR (    (Recinfo.ename IS NULL)
                  AND (X_Ename IS NULL)))
         AND (   (Recinfo.job =  X_Job)
              OR (    (Recinfo.job IS NULL)
                  AND (X_Job IS NULL)))
```

```
        AND (   (Recinfo.mgr =  X_Mgr)
             OR (    (Recinfo.mgr IS NULL)
                 AND (X_Mgr IS NULL)))
        AND (   (Recinfo.hiredate =  X_Hiredate)
             OR (    (Recinfo.hiredate IS NULL)
                 AND (X_Hiredate IS NULL)))
        AND (   (Recinfo.sal =  X_Sal)
             OR (    (Recinfo.sal IS NULL)
                 AND (X_Sal IS NULL)))
        AND (   (Recinfo.comm =  X_Comm)
             OR (    (Recinfo.comm IS NULL)
                 AND (X_Comm IS NULL)))
        AND (Recinfo.deptno =  X_Deptno)
    ) then
    return;
  else
    FND_MESSAGE.Set_Name('FND', 'FORM_RECORD_CHANGED');
    APP_EXCEPTION.Raise_Exception;
  end if;
END Lock_Row;

PROCEDURE Update_Row(X_Rowid         VARCHAR2,
                     X_Empno         NUMBER,
                     X_Ename         VARCHAR2,
                     X_Job           VARCHAR2,
                     X_Mgr           NUMBER,
                     X_Hiredate      DATE,
                     X_Sal           NUMBER,
                     X_Comm          NUMBER,
                     X_Deptno        NUMBER
) IS
BEGIN
  UPDATE emp
  SET
     empno                          =     X_Empno,
     ename                          =     X_Ename,
     job                            =     X_Job,
     mgr                            =     X_Mgr,
     hiredate                       =     X_Hiredate,
     sal                            =     X_Sal,
     comm                           =     X_Comm,
     deptno                         =     X_Deptno
  WHERE rowid = X_Rowid;
  if (SQL%NOTFOUND) then
```

```
      Raise NO_DATA_FOUND;
    end if;
  END Update_Row;

  PROCEDURE Delete_Row(X_Rowid VARCHAR2) IS
  BEGIN
    DELETE FROM emp
    WHERE rowid = X_Rowid;
    if (SQL%NOTFOUND) then
      Raise NO_DATA_FOUND;
    end if;
  END Delete_Row;

END &PACKAGE_NAME;
/
show errors package body &PACKAGE_NAME
SELECT to_date('SQLERROR') FROM user_errors
WHERE  name = '&PACKAGE_NAME'
AND    type = 'PACKAGE BODY'
/
commit;
exit;
```

CHAPTER

5

Setting the Properties of Container Objects

This section describes the standard properties and behaviors for Modules, Windows, Canvasses, Blocks, and Regions.

These characteristics may be set in the following ways:

- Inherited through property classes, which cause certain properties to be identical in all forms (such as canvas visual attributes)
- At the discretion of the developer during form design (such as window sizes)
- At runtime, by calling standard library routines (such as window positions)

The following container objects are discussed in this chapter:

- Modules
- Windows, including standards for modal and non–modal windows
- Canvasses, including standards for content and stacked canvasses
- Blocks
- Regions

Modules

Module properties establish an overall framework for the look and feel of each form.

For more information, see the *Oracle Applications User Interface Standards.*

Modules
Oracle Applications User Interface Standards

Property Class

The TEMPLATE form automatically applies the MODULE property class to the module. The settings of this class vary on each GUI platform.

Warning: Do not change any values set by the MODULE property class.

Module Names

Make sure that in each of your forms, the Module Name matches the file name. For example, if a form is called POXPOMPO.fmb, make sure the Module Name (visible in Oracle Forms Designer) is POXPOMPO.

This is especially important if you reference objects from your form. Zoom also relies on the Module Name being correct.

First Navigation Block

Set this property to the name of the first block that users visit when a form is run. Do not set to a WORLD or CONTROL block.

This property also controls where the cursor goes after a CLEAR_FORM, as well as the default 'Action–>Save and Proceed' behavior.

Windows

From the APPSTAND form, windows automatically inherit the proper look and feel of the GUI platform on which they are running, such as characteristics of the frame, title bar fonts, and window manager buttons. This section describes features common to all Oracle Applications windows, as well as behaviors for modal and non–modal windows.

ROOT_WINDOW

The ROOT_WINDOW is a special Oracle Forms window that behaves differently from other windows. Do not use the ROOT_WINDOW, because it interferes with the proper functioning of the Toolbar and other standard Oracle Applications objects.

For more information, see the *Oracle Applications User Interface Standards.*

Windows
Oracle Applications User Interface Standards

Non–Modal Windows

A non–modal window (a ”regular window”) allows the user to interact with any other window, as well as the toolbar and the menu. Non–modal windows are used for the display of most application entities.

For more information, see the *Oracle Applications User Interface Standards.*

Non–Modal Windows
Oracle Applications User Interface Standards

Property Class

Apply the WINDOW property class to all non–modal windows.

View

Always enter the name of the content canvas associated with the window.

Positioning (X, Y)

Non–modal windows can be positioned in a variety of styles. Code all the logic that positions windows in the APP_CUSTOM.OPEN_ WINDOW procedure, and any event that would cause a window to open must call that procedure (for example, pressing a Drilldown Record Indicator, pressing the Open button of a combination block, or pressing a button that leads to a child entity in a different window).

The first window(s) of a form that appears when the form is invoked must also be programmatically positioned.

Positioning Windows Upon Opening (See page 0 – 2)

Title

The *Oracle Applications User Interface Standards* describe how to title your non–modal windows, including rules for showing context information.

Non–Modal Window Title
Oracle Applications User Interface Standards

Some window titles include context information that changes according to the data displayed. Usually, the main entity window title does not change, but titles of detail windows do change to show context information. For these detail windows, you use an APPCORE window titling routine. For all windows, you also set the Title property of the window to the base title you want.

Setting Window Titles Dynamically (See page 0 – 5)

Size (Width, Height)

The maximum allowed window size is 7.8 inches wide and 5 inches high. Any size smaller than this is allowed, down to a minimum of approximately two inches by two inches. If you do not need the maximum size for your window items, you *should* make the window smaller to leave the user with extra space for other windows on the screen.

Closing

You must explicitly code the closing behavior for your windows to ensure that windows do not close while in Enter Query mode, closing the first window of a form closes the entire form, and other standard behaviors. You code the closing behavior in the APP_CUSTOM.CLOSE_WINDOW procedure.

Closing Windows (See page 0 – 3)

Window Opening

If you have logic that must occur when a window opens, place the logic in APP_CUSTOM.OPEN_WINDOW. You must add logic to control block coordination and to position windows.

Master–Detail Relations (See page 7 – 6)

Suggestion: You do not need to explicitly show a window. A GO_BLOCK to a block within a window opens the window automatically.

Modal Windows and Modal Windows with Menu

Modal windows force users to work within a single window, and then to either accept or cancel the changes they have made. Most modal windows in Oracle Applications that use text fields allow limited access to the toolbar and menu. These windows are called Modal windows with Menu.

For more information, see the *Oracle Applications User Interface Standards.*

Modal Windows
Oracle Applications User Interface Standards

Property Class

Apply the WINDOW_DIALOG_WITH_MENU property class to all modal windows that need access to the menu (for List of Values). Use the WINDOW_DIALOG property class to create a modal window with no access to the toolbar or menu.

View

Always enter the name of the content canvas associated with the window.

Position

Modal windows are always opened centered on the screen. They are re–centered each time they are opened.

Include the following call in the code that opens your modal window:

```
app_window.set_window_position('<window_name>','CENTER');
```

Positioning Windows Upon Opening (See page 0 – 2)

Closing

Modal windows cannot be closed with the native GUI window close mechanism. You must explicitly close the window in your code, typically with the following buttons:

OK Closes the window. In some cases, it may perform a commit as well.

Suggestion: A specific verb can be substituted in place of 'OK'. For instance, in a window designed to record additional information before posting, buttons of 'Post' and 'Cancel' are clearer to the user than just 'OK' and 'Cancel'.

Cancel Clears the data without asking for confirmation, and closes the window.

You must move the cursor to a block in a different window before closing the modal window.

Example

Trigger: WHEN–BUTTON–PRESSED on item CANCEL:

```
go_block('LINES');
hide_window('APPROVE_LINES');
```

Processing KEY– Triggers

See Dialog Blocks for information on trapping specific KEY– triggers within a modal window.

Dialog Blocks (See page 5 – 11)

Disabling Specific Menu Entries

If you use the WINDOW_DIALOG_WITH_MENU property class, you must also account for several items on the menu that do not apply to modal windows.

Enable and disable SAVE to control the ’Action–>Save’ and ’Action–>Save and Enter Next’ menu entries. Save is automatically disabled when you call APP_FORM.QUERY_ONLY MODE.

Before entering a modal window, call APP_SPECIAL.ENABLE(’MODAL’, PROPERTY_OFF). When you leave the block, call ENABLE again with PROPERTY_ON. PROPERTY_OFF disables the menu items that are disallowed in a modal block.

APP_SPECIAL: Menu and Toolbar Control
(See page 10 – 9)

Canvasses

This section describes the settings for content and stacked canvasses.

For more information about the use and behavior of canvasses, see the *Oracle Applications User Interface Standards*.

Canvasses
Oracle Applications User Interface Standards

Window

Always enter the name of the window the canvas is shown in.

Content Canvasses

Property Class

You should apply the CANVAS property class to all content canvasses.

Size (Width, Height)

You should size the content canvas the same as the window it is shown in.

Stacked Canvasses

One or more stacked canvasses may be rendered in front of the content canvas of a particular window. If needed, a stacked canvas may fully occupy a window.

See ”Alternative Regions” for a full description of stacked canvas behavior with regions.

Alternative Regions (See page 7 – 12)

For more information, see the *Oracle Applications User Interface Standards*.

Stacked Canvasses
Oracle Applications User Interface Standards

Property Class

Stacked canvasses should use the CANVAS_STACKED property class to enforce the correct behavior.

Display Characteristics

Stacked canvasses should adhere to these display characteristics:

- Only the one stacked canvas that is to be shown when its window is first opened should be set to Displayed.
- Stacked canvasses always have raise on entry set to TRUE.
- Canvasses stacked on top of each other (as in alternative regions) should all be the same size.

The content canvas should be blank in the area that would be covered by the stacked canvasses.

Sequence

When multiple stacked canvasses occupy the same window, and may overlap, the sequence must be set so that the proper canvasses, or portions of canvasses, are displayed.

Whenever possible you should explicitly hide a stacked canvas that is not visible to a user. This reduces the resources that the widgets on it may consume.

Blocks

This section discusses the general settings for all blocks, as well as how to set up blocks for the following situations:

- Context Blocks
- Dialog Blocks
- Blocks With No Base Table
- Multi–Record Blocks
- Single–Record Blocks
- Combination Blocks
- Master–Detail Relations
- Dynamic WHERE Clauses

For more information, see the *Oracle Applications User Interface Standards.*

Blocks
Oracle Applications User Interface Standards

Property Class

Use the BLOCK property class for all non–modal blocks; use BLOCK_DIALOG for blocks displayed within modal windows.

Never override the Visual Attribute property of this class; it varies on each platform.

Key–Mode

If the block is based on a table or a single–table view, set Key–Mode to Unique. If the block is based on a join view, set it to Non–Updatable. Ensure that at least one item in the block is marked as a primary key (set Primary Key at the item level to True for each item that makes up the primary key of the block).

Views (See page 3 – 7)

Delete Allowed

To prevent deletes in a block, set the Delete Allowed property for the block to False (do not code a DELREC trigger to prevent deletes).

Next and Previous Navigation Block

Set these properties for every navigable block. These settings affect the Go–>Next Block and the Go–>Previous Block entries on the pulldown menu, and should not be set to CONTROL or WORLD blocks.

For the first block, set the Previous Navigation block to be itself. For the last block, set the Next Block to be itself.

If the Next Block changes dynamically at runtime, you must still set the property to something logical. You decide the most logical flow of your next and previous blocks.

Context Blocks

Context blocks are shown in detail windows to provide context, and replicate fields that are shown in master windows. To create a context block, make display items that are part of the same block as the master and mirror the context field to the master field.

Dialog Blocks

Dialog blocks are the blocks presented in modal windows. They require the user to interact with them before proceeding to other windows of the application.

Modal Windows (See page 5 – 5)

Processing KEY– Triggers

Most standard Oracle Forms functions, such as Save, Next Block, and Clear All, do not apply in a dialog block. Although the Oracle Applications menu and toolbar may not be accessible, Oracle Forms functions can still be invoked from the keyboard unless you disable them. You should disable all KEY– triggers for the block by coding a KEY–OTHERS trigger that calls APP_EXCEPTION.DISABLED, which will cause a beep when the user attempts a disabled function. You then specifically enable some functions for the block by coding the following additional KEY– triggers:

KEY– Trigger	Code
KEY–OTHERS	app_exception.disabled; (1)
KEY–NEXT–ITEM	next_item;
KEY–PREVIOUS–ITEM	previous_item;
KEY–CLRREC	clear_record
KEY–EDIT	app_standard.event('KEY–EDIT');
KEY–LISTVAL	app_standard.event('KEY–LISTVAL');
KEY–ENTER	enter;
KEY–HELP	app_standard.event('KEY–HELP');
KEY–PRINT	print;

Table 5 – 1 (Page 1 of 1)

(1) This disables every KEY– function in the block that does not have a specific KEY– trigger coded for it.

If the dialog block allows multiple records, then the following KEY– triggers should also be enabled:

KEY– Trigger	Code
KEY–CREREC	create_record;
KEY–NXTREC	next_record;
KEY–PREVREC	previous_record;
KEY–UP	up;
KEY–DOWN	down;

Table 5 – 2 (Page 1 of 1)

Other functions may be enabled if appropriate for the specific dialog block.

In cases where most functions are enabled, just disable those that do not apply by calling APP_EXCEPTION.DISABLED in the KEY– triggers for those specific functions that you want to disable.

Navigation

Navigation to items outside a dialog block must be prevented while the modal window is open. [Tab] must be restricted to fields within that window. The following guidelines prevent the user from navigating out of a dialog block:

- The Navigation Style of the block is usually Same Record. It should never be Change Block.
- The Next and Previous Navigation Blocks should be the same as the block itself.
- Set Next and Previous Navigation Item properties as necessary to keep the user inside the dialog block.

Blocks With No Base Table

You may need to implement blocks that have no base table or view. Use transactional triggers (ON–INSERT, ON–LOCK, etc.) if such a block must process commits.

Do not base the block on a dummy table such as FND_DUAL.

For example, the ”Move Inventory Items” form submits a concurrent request to process the data entered on the screen. Code an ON–INSERT trigger to call the concurrent process submission routine.

Concurrent Processing (See page 15 – 2)

Single–Record Blocks

Single–record blocks allow the user to see as many items of an entity as possible, at the tradeoff of only seeing one record at a time.

Navigation Styles

If the block has no detail blocks, or it has detail blocks but they are in different windows, the Navigation Style should be Same Record; otherwise it is Change Block.

Blocks With Only One Record Available

For blocks with only one record of data, you may want to disable the first record, last record, previous record, and next record options on the Go menu.

To do this, code a block–level WHEN–NEW–RECORD–INSTANCE trigger (Execution Style: Override) with these lines:

```
app_standard.event('WHEN-NEW-RECORD-INSTANCE');
app_special.enable('SINGLE', PROPERTY_OFF);
```

To prevent the user from using a key to perform functions incompatible with one record blocks, code block–level KEY–DOWN, KEY–CREREC, and KEY–NXTREC triggers (Execution Style: Override) containing:

```
app_exception.disabled;
```

APP_SPECIAL: Menu and Toolbar Control (See page 10 – 9)

Multi–Record Blocks

Multi–record blocks allow the user to see as many records of an entity as possible, usually at the tradeoff of seeing fewer attributes of each record simultaneously.

For more information, see the *Oracle Applications User Interface Standards*.

Multi–Record Blocks
Oracle Applications User Interface Standards

You must provide either a current record indicator or a drilldown indicator for each multi–record block, depending on whether the block supports drilldown.

Navigation Style

Set the Navigation Style to Change Record for all multi–record blocks.

Current Record Indicator

If the block does not have any detail blocks (and therefore does not support drilldown), create a current record indicator for the block as follows: Create a text item in the multi–record block. Give the text item the name ’CURRENT_RECORD_INDICATOR’ and apply the property class ’CURRENT_RECORD_INDICATOR’.

Single–clicking on the indicator moves the cursor to the first navigable field of the appropriate record. Do this by creating an item–level WHEN–NEW–ITEM–INSTANCE trigger (Execution Style: Override) on the record indicator item, and issue a `go_item` to the first field of the block. For example:

```
GO_ITEM('lines.order_line_num');
```

For more information, see the *Oracle Applications User Interface Standards.*

Current Record Indicator
Oracle Applications User Interface Standards

Drilldown Indicator

If the multi–record block supports drilldown to one or more detail blocks, create a drilldown indicator as follows: Create a text item in the multi–record block. Name it ’DRILLDOWN_RECORD_INDICATOR’, and apply the property class ’DRILLDOWN_RECORD_INDICATOR.’

Add an item–level WHEN–NEW–ITEM–INSTANCE trigger (Execution Style: Override) to the drilldown indicator item. Call the same logic as the button that corresponds to the drilldown block. For Combination blocks, this should move to the Detail window. In other blocks, if there are one or more child blocks, drilldown moves you to one of them.

Combination Blocks (See page 7 – 8)

You should account for situations where movement to the drilldown block is currently not allowed and the corresponding button is disabled. Check for this condition in the WHEN–NEW–ITEM–INSTANCE trigger before doing the drilldown. If the drilldown is not enabled, issue a call to APP_EXCEPTION.DISABLED and navigate to the first item in the current block.

For more information, see the *Oracle Applications User Interface Standards.*

Current Record Indicator
Oracle Applications User Interface Standards

Combination Blocks

Combination blocks are hybrid formats, where fields are presented in both multi–record (Summary) and single–record (Detail) formats. The Summary and Detail formats are each presented in their own window, but all of the fields of both formats are part of a single block.

☞ **Attention:** Do not confuse the Detail of Summary–Detail with the Detail of Master–Detail.

For more information, see the *Oracle Applications User Interface Standards*.

Combination Blocks
Oracle Applications User Interface Standards

Implementing a Combination Block (See page 7 – 8)

Master–Detail Relations

For more information on the look and feel of master–detail relations, see the *Oracle Applications User Interface Standards*.

Master–Detail Characteristics
Oracle Applications User Interface Standards

Coding Master–Detail Relations (See page 7 – 6)

Prevent Masterless Operations

A user cannot enter or query detail records except in the context of a master record. Always set this property to True.

Prevent Deletion of Detail Records

Because your form should be built using underlying views instead of actual tables, you should not allow the normal Oracle Forms deletion of detail records. Instead, set the Master Deletes property of the relation to Isolated. Then, delete your detail records as part of your Delete_Row procedure in the table handler for the master table.

Other Behaviors

- When a detail block is in a different window than its master, but the detail window is modal, the detail block should only query upon navigation to the block. Set Deferred to True and AutoQuery to True for the relation. Do not code any logic for this relation in the OPEN_WINDOW or CLOSE_WINDOW procedure.

- The first master block of a form does not autoquery unless
  - only a very small number of records will be returned
  - the query will be fast
  - most likely the user will operate on one or more of the queried records

  To autoquery the first block of a form, code the following:

Trigger: WHEN–NEW–FORM–INSTANCE

```
do_key('execute_query');
```

- Do not code anything specific to windows being iconified, even though iconifying a window that contains a master block may make it difficult to operate with a detail block.
- Do not use Master–Detail cascade delete because it is an inefficient operation on the client side. It also generates triggers with hardcoded messages.

Dynamic WHERE Clauses

You may modify the default WHERE clause of a block at runtime for these cases:

- Any query run within the block must adhere to the new criteria
- Complex sub–selects of other SQL are required to query rows requested by a user.

All other cases should just populate values in the PRE–QUERY trigger.

Regions

Regions are groups of fields. Some regions, called Alternative Regions, appear only at selected times and are displayed on stacked canvasses.

Coding Alternative Region Behavior (See page 0 – 12)

For more information about the look and feel of regions, see the *Oracle Applications User Interface Standards*.

Regions
Oracle Applications User Interface Standards

Overflow Regions

Overflow regions show additional fields of a multi–record block in a single–record format immediately below the multi–record fields.

Simply create these fields within the block of interest, and set the Items Displayed property to 1.

CHAPTER

6

Setting the Properties of Widget Objects

This section describes the standard properties for the various form widgets that a user interacts with. It also describes the coding techniques for implementing these widgets.

The following topics are covered:

- Text Items
- Display Items
- Poplists
- Option Groups
- Check boxes
- Buttons
- Lists of Values (LOV)
- Alerts
- Editors
- Flexfields
- Setting Item Properties Dynamically

Text Items

For more information about the general look and feel of widgets, see the *Oracle Applications User Interface Standards*.

General Properties
Oracle Applications User Interface Standards

The following information applies to all text items.

Property Classes

In general, most text items use the TEXT_ITEM property class.

Use the TEXT_ITEM_DISPLAY_ONLY property class on fields that do not allow a user to type, but must support scrolling and or querying. Some date fields use this property class. In cases where the user must tab to display–only fields located on a part of the canvas that is not immediately visible, you may override the NAVIGABLE property inherited from the property class.

Use the TEXT_ITEM_MULTILINE property class on all multiline text items.

Use TEXT_ITEM_DATE for date fields unless the item is display only.

Apply the CREATION_OR_LAST_UPDATE property class to the items containing the WHO date information, CREATION_DATE and LAST_UPDATE_DATE.

Query Length for Text Items

Set the maximum query length to 255 to allow for complex query criteria.

WHEN–VALIDATE–ITEM

This trigger fires when the field value changes. Also, a Required field is not enforced until record–level validation. Therefore you may need to write logic that specifically accounts for a NULL value.

Alignment

To support bidirectional languages such as Arabic, do not use Left or Right alignments (numeric text items can use Right alignments). Use Start and End instead. You may use Center where appropriate.

Generally the property class sets the correct alignment, unless you need to specify Right or Center.

Date Fields

Date fields that the user enters should use the Calendar.

The Calendar (See page 9 – 23)

Data Type

For date fields, use the DATE data type unless the user needs to enter time. Use the DATETIME data type to require the user to enter time.

To default a form field to the current date without the time, use $$DBDATE$$. To default a form field to the current date and time, use $DBDATETIME$$.

Date Field Maximum Length

Create date fields as 11 characters without time, or 20 characters with time.

You do not need to specify a format mask in the item. Oracle Forms defaults the format correctly for each language from the environment variable NLS_DATE_FORMAT.

Oracle Applications currently requires an NLS_DATE_FORMAT setting of DD–MON–RR. Forms date fields that are 11 or 20 characters long will display a four–character year (DD–MON–RRRR) automatically.

Date Field Validation

In general, validate your date fields at the record level rather than at the item level.

Record level validation allows the user to correct errors more easily, especially in a From Date / To Date situation. After entering an incorrect date (last year instead of next year), the user should not need to change first the To Date before correcting the From Date.

Display Items

Display items do not allow any user interaction – they merely display data and never accept cursor focus.

Use a display item for the following situations:

- Null–canvas fields
- Context fields
- Fields that act as titles or prompts

If a field must accept cursor focus, either to allow scrolling or querying, it must be a text item, not a display item.

Text Items (See page 6 – 2)

For more information, see the *Oracle Applications User Interface Standards.*

Display Items
Oracle Applications User Interface Standards

Property Class

If an item is used to hold a dynamic title, use DYNAMIC_TITLE; if an item holds a prompt, use DYNAMIC_PROMPT. Both of these property classes provide a "canvas" colored background (gray). Otherwise, you should apply the DISPLAY_ITEM property class to your display items (provides a white background with no bevel).

Y Position

A display item on a canvas using the DISPLAY_ITEM property class must be positioned .02 inches below the gridline (to account for the lack of the bevel used for enterable text items).

Items using either DYNAMIC_TITLE or DYNAMIC_PROMPT must be .05 inches below the gridline.

These offsets allow the text within them to align properly with other widgets or boilerplate.

Alignment

To support bidirectional languages such as Arabic, do not use Left or Right alignments. Use Start instead of Left, and generally use End in place of Right. Use Right only with numeric fields. You may use Center where appropriate.

Generally, applying the correct property class sets alignment correctly.

Width

Always make sure that the width (length) of the display item is large enough to accommodate translated values without truncating them. Items using either DYNAMIC_TITLE or DYNAMIC_PROMPT inherit a maximum length of 80 (which you should not change). Typically, the largest value you could accommodate in English would be about 60 characters (which,, if expanded about 30 percent, fills an 80–character–wide field).

Poplists

Poplists are used for two distinct purposes in Oracle Applications: to hold data in a small list of possible values, and to set the displayed region for a set of alternative regions.

For information about the look and feel of poplists, see the *Oracle Applications User Interface Standards*.

PopLists
Oracle Applications User Interface Standards

Property Class

Poplists holding data use the LIST property class. Poplists that serve as control elements for alternative regions use the LIST_REGION_CONTROL property class.

Coding Alternative Region Behavior (See page 0 – 12)

Limits

The maximum width of a list element is 30 characters. Your longest value in English for a 30–character–wide poplist should be no longer than 23 characters to account for expansion of values for some languages.

Dynamic List Elements

You may need to populate a list at runtime. If so, be aware of the following issues:

- Never use a list item if you expect more than fifteen elements
- Do not change a list on a per–record basis
- Make sure each populated list has a value that matches the default value. You can dynamically change the default by specifying it as a reference to another field, as opposed to a hardcoded value.

Setting the Value

Always set a poplist based on its value, not its label. The value never gets translated, but the label may. When you set the Default Value property, Oracle Forms will actually accept the label value (for example, Good), but you should always use the hidden value (for example, G) instead.

Option Groups

For information about the look and feel of option groups, see the *Oracle Applications User Interface Standards*.

Option Groups
Oracle Applications User Interface Standards

Property Class

Apply the RADIO_BUTTON property class to each button of an option group. There is no property class to assign to the group itself.

Access Keys

An option group that serves to place the form in a mode (as opposed to holding data) should have Access keys specified for each of the buttons.

Check Boxes

For information about the look and feel of check boxes, see the *Oracle Applications User Interface Standards*.

Check Boxes
Oracle Applications User Interface Standards

Master–Detail Relations (Blocks) (See page 7 – 6)

Property Class

Apply the CHECKBOX property class to each normal check box (used for data entry). Apply the CHECKBOX_COORDINATION property class to each coordination check box.

Coding Window Behavior (See page 0 – 2)

Buttons

Buttons can either be textual or iconic; that is, they display either words or a picture.

Buttons should be items in the block they act upon or appear to be part of (not in a control block). For example, create an ”Accept” button as a non–database object in the block on which it performs the accept. If you navigate to a LINES block from a HEADER block using a ”Lines” button, make the button part of the HEADER block.

For information about the look and feel of buttons, see the *Oracle Applications User Interface Standards*.

Buttons
Oracle Applications User Interface Standards

Property Class

Textual buttons use the BUTTON property class. Iconic buttons use the BUTTON_ICONIC property class and typically appear only in the toolbar or in folder forms.

Warning: Never override the height specified by the BUTTON property class.

Navigable and Mouse Navigate Properties

Single record block buttons are Navigable TRUE. Multi–record block buttons are Navigable FALSE. The exception is Clear buttons, which should always use Navigable FALSE. This is to prevent users from accidentally clearing records when they expect to fire the default button.

All buttons are Mouse Navigate False.

Iconic Buttons and Keyboard Only Operation

Iconic buttons cannot be operated from the keyboard. If your form is intended to used for heads–down data entry (keyboard only), this implies the functionality they add must either be non–essential or have a a secondary invocation method, such as the menu.

Enter–Query Mode

Most buttons do not apply in Enter–Query mode. Users cannot leave the current record while in this mode. You do not need to explicitly disable a button while in Enter–Query mode; instead set the trigger property ”Fire in Enter–Query mode” for the WHEN–BUTTON–PRESSED trigger to FALSE.

Call APP_STANDARD.APP_VALIDATE

Buttons should call APP_STANDARD.APP_VALIDATE and pass a scope before performing their action. This ensures that your records are valid before performing an action, and that the button acts on the expected block.

Lists of Values (LOVs)

Use Lists of Values to provide validation on a text item when you expect to have more than fifteen values.

For more information, see the *Oracle Applications User Interface Standards.*

List of Values (LOV)
Oracle Applications User Interface Standards

Property Class

Apply the LOV property class to all LOVs.

Suggestion: You may override the LOV Type and Auto Refresh property as needed.

Base LOVs on Views

You should base your LOVs on views. This denormalizes any foreign key references and provides the same performance advantages as basing your blocks on views.

An LOV view is usually simpler than a form view, since it does not include all denormalized columns. The LOV view does need to join to foreign key tables to get meanings associated with list and radio group values, whereas in a form view the meanings are hardcoded into the boilerplate text or the widget.

Views (See page 3 – 7)

When Not To Use a View

If the view is very simple or, conversely, overly–complicated because the bind variables are not in the SELECT list, then you may code the entire SQL needed by the LOV directly into the form.

Rules

- The title of an LOV is the name of the object in the LOV, and is plural.
- The prompt of the first column is related to, or matches identically, the prompt of the item that invoked it.
- The width of each column should be wide enough to show most values (just like the width of fields on a canvas). Make the LOV

wide enough to show all included columns, up to a maximum of 7.8”.

- Always specify the default value by the value, not the label. This ensures that the default is translated correctly.
- Use your judgement when deciding which columns to bring over for the LOV. Sometime you only need to bring over a primary key and its display name, if the rest of the data would take too long to fetch. After the row is selected, use the WHEN–VALIDATE–ITEM trigger to bring over any other necessary columns. VARCHAR(2000) columns should not be part of an LOV.

Show Only Valid Values

A LOV should show only those rows that currently can be selected, unless the LOV is in a Find Window (Find Window LOV’s show all rows that have ever been valid).

EXCEPTION: Validation can be performed after–the–fact if any of the following apply:

- The validation clause cannot be written in SQL.
- The validation clause is too costly to evaluate in SQL.
- The reason for exclusion from the list is obscure to the user.

In such cases, after the value is selected, show an error message indicating exactly why the value could not be selected.

Row–LOV

For more information on how to code Row–LOVs in response to ”Query–>Find,” see:

Query Find Windows (See page 8 – 2)

Assigning Values in POST–QUERY

If your item has a List of Values, the ”LOV for Validation” property is set to True, and you assign a value to the field in the POST–QUERY trigger, the item is marked as changed because the LOV fires. To avoid this complication, set the RECORD_STATUS back to QUERY at the end of the POST–QUERY trigger.

LOV Behaviors

You may alter the properties on your LOV to create the following behavior:

Auto Refresh

If the values displayed by the LOV are static during a session and the number of rows is not excessive, turn Auto Refresh off (False) to cache the LOV values for the session. Caching the values avoids database hits and network round trips for subsequent invocations of the LOV, and eliminating unnecessary round trips is a key factor in producing a product that can run on a wide area network. However, the caching consumes memory that is not recovered until the form is closed.

Long List

If an LOV may show more than one hundred rows, then the user must be prompted to reduce the list of valid values first (Long List: True).

Never have Long List set to True, and Auto Refresh set to False, on an LOV. This combination would cause only the reduced set of rows to be cached if the user enters something in the reduction criteria window. With Auto Refresh off, there is no way of returning to the full set of rows. Typically it is not wise to cache an LOV that returns more than 100 rows.

Example LOV

The EMP table contains the following columns: EMPNO, ENAME, JOB, MGR, HIREDATE, SAL, COMM AND DEPTNO. DEPTNO is a foreign key to the table DEPT, which contains the columns DEPTNO, DNAME, and LOC.

A form view of the EMP table would contain all columns in EMP, denormalize EMP.DEPTNO, and include the column DEPT.DNAME, as well. It might also include DEPT.LOCATION and other DEPT columns, and contain records for all past and present employees:

```
CREATE VIEW EMP_V AS
SELECT EMP.EMPNO, EMP.ENAME, EMP.JOB, EMP.MGR,
  EMP.HIREDATE, EMP.SAL, EMP.COMM,
  EMP.DEPTNO, DEPT.DNAME, DEPT.LOCATION
FROM EMP, DEPT
WHERE DEPT.DEPTNO = EMP.DEPTNO;
```

By contrast, an LOV view of EMP would only include columns EMP.EMPNO and EMP.ENAME. DEPT information would be included only if necessary to help select an employee.

Decoding Y/N Values

For Y / N values, decode ”Y” to ”*” and ”N” to null to avoid a join to FND_LOOKUPS.

Example A table T has columns ID, NAME, and ENABLED_FLAG. ENABLED_FLAG contains Y / N values. Create your view as follows:

```
CREATE VIEW T_V AS
   SELECT ID, NAME,
   DECODE(ENABLED_FLAG, 'Y', '*', NULL)
   FROM T;
```

Dependent Fields

An LOV on a dependent field should use the value in the master field to reduce the list.

For example, if NAME is dependent on TYPE, the entry LOV for NAME’s WHERE clause would include the condition:

```
WHERE TYPE = :MY_BLOCK.TYPE
```

LOVs in ENTER–QUERY Mode

LOVs in ENTER–QUERY mode should be used sparingly, as Query Find is the preferred method for a user to locate records.

Query Find Windows (See page 8 – 2)

You should only code them where they dramatically improve the usability of ENTER–QUERY mode, and you expect this mode to be used regularly despite Query Find.

An LOV in ENTER–QUERY mode should display all values that the user can query, not just currently valid values.

> EXAMPLE: An LOV for vendors in a purchase order form in enter–query mode shows all vendors that could ever be placed on a PO, not just the set of vendors that currently are allowed to be placed on a PO.

Do not reuse the entry LOV in ENTER_QUERY mode unless it provides the correct set of data for both modes.

☞ **Attention:** WHEN–VALIDATE–ITEM does not fire in ENTER–QUERY mode. Therefore, you cannot depend on the WHEN–VALIDATE–ITEM trigger to clear hidden fields when selecting from an ENTER–QUERY LOV.

Implementation

To enable LOVs in ENTER–QUERY mode on an item, create an item–level KEY–LISTVAL trigger as follows:

Trigger: KEY–LISTVAL

```
IF (:SYSTEM.MODE != 'ENTER-QUERY') THEN LIST_VALUES;
ELSE SHOW_LOV('query lov');
END IF;
```

Return into the LOV Item Only

When implementing LOVs in ENTER–QUERY mode, do not return values into any field other than the field from which the LOV is invoked. If the LOV selects into a hidden field, there is no way to clear the hidden field. Clearing or typing over the displayed field will not clear the hidden field. Users must select another value from the LOV or cancel their query and start over.

Alerts

Oracle Applications does not use the native Oracle Forms alert object. The Oracle Application Object Library Message Dictionary feature is used instead, as it provides full translation capabilities and handles text larger than 80 characters.

Message Dictionary APIs for PL/SQL Procedures (See page 12 – 14)

Editors

Do not write special code for the editor. Rely on native Oracle Forms behavior.

Flexfields

For more information on visual standards for descriptive flexfields, see the *Oracle Applications User Interface Standards*.

Descriptive Flexfields
Oracle Applications User Interface Standards

For more information on visual standards for key flexfields, see the *Oracle Applications User Interface Standards*.

Key Flexfields
Oracle Applications User Interface Standards

For information on building flexfields into your tables, see the *Oracle Application Object Library Reference Manual*, Release 10.

There is a PL/SQL API for building flexfields into your forms.

Flexfield APIs (See page 14 – 25)

Usage

All entities should provide a descriptive flexfield to allow customization.

Avoid using the same descriptive flexfield definition in more than one form. Because a customer can reference any field in their flexfield definition, they may reference a field that exists in one form but not the others.

Key flexfields should use the ”ENABLE_LIST_LAMP” LOV, with the Use LOV for Validation property set to False. Descriptive flexfields do not use an LOV.

Setting Item Properties Dynamically

This section describes item properties and how they are used in Oracle Applications.

There are item properties and item property behaviors specific to Oracle Applications, in addition to those available with native Oracle Forms. Specifically, Oracle Applications has modified the behavior of the ENABLED property and added two new properties, ENTERABLE and ALTERABLE, to the native Oracle Forms functionality. You can set these properties dynamically at runtime.

Use APP_ITEM_PROPERTY.SET_PROPERTY

In your PL/SQL code, *do not* use the Oracle Forms built–in SET_ITEM_PROPERTY to set the following field properties:

- DISPLAYED
- ENABLED
- ENTERABLE
- ALTERABLE
- INSERT_ALLOWED
- UPDATEABLE
- NAVIGABLE
- REQUIRED
- ICON_NAME

Instead, use the APPCORE library package procedure APP_ITEM_PROPERTY.SET_PROPERTY for the following reasons:

- Using SET_ITEM_PROPERTY sometimes affects properties other than the ones you set. For instance, setting ENABLED off also turns off NAVIGABLE as a side–effect. If you then turn ENABLED back on, NAVIGABLE will remain off. If you use APP_ITEM_PROPERTY, when you turn ENABLED back on, NAVIGABLE is turned on as well.
- APP_ITEM_PROPERTY remaps some properties to other values that are consistent with the *Oracle Applications User Interface Standards,* like changing visual attributes.
- There are some additional properties that APP_ITEM_ PROPERTY provides that native Oracle Forms does not. You

should use APP_ITEM_PROPERTY to make sure that these additional properties behave as you expect.

Hint Attribute

Do not use the "Hint Attribute" with any of your items.

List of Item Properties

Below is a list of properties that can be set dynamically at runtime and have a specified behavior as described in the *Oracle Applications User Interface Standards*. For each property, there is a description of how it behaves and rules for when to use it.

DISPLAYED

Call APP_ITEM_PROPERTY.SET_PROPERTY(item_id, DISPLAYED, PROPERTY_OFF) to turn DISPLAYED off.

Setting	DISPLAYED	=	OFF
sets	ENABLED	=	OFF
sets	NAVIGABLE	=	OFF
sets	REQUIRED	=	OFF
sets	UPDATEABLE	=	OFF
sets	UPDATE_NULL	=	OFF
sets	QUERYABLE	=	OFF

Call APP_ITEM_PROPERTY.SET_PROPERTY(item_id, DISPLAYED, PROPERTY_ON) to turn DISPLAYED on.

Setting	DISPLAYED	=	ON
sets	ENABLED	=	ON

Setting	DISPLAYED	=	ON
sets	NAVIGABLE	=	ON
sets	UPDATEABLE	=	ON
sets	UPDATE_NULL	=	OFF
sets	INSERT_ALLOWED	=	ON
sets	QUERYABLE	=	ON

ENABLED

The Oracle Applications ENABLED property (not to be confused with the Oracle Forms ENABLED property) is used to disable items that do not currently apply.

For text and list items, ENABLED is a combination of the NAVIGABLE, INSERTABLE, and UPDATE_ALLOWED properties (dithered text is hard to read, and the Oracle Forms ENABLED property dithers the text in the field). The user can click into a text or list item, but cannot type. This is visually indicated by a gray background in the field. This visual attribute will apply to every record in the block. If you are disabling the field on only the current record, use ENTERABLE. The only difference between ENTERABLE and ENABLED for text and list items is that the visual attribute is applied to the current record, and all records, respectively.

For items other than text and list items, setting ENABLED sets the native Oracle Forms ENABLED property. Any item set with the Oracle Forms ENABLED property off cannot be navigated to using the mouse or the keyboard, and has dithered text (every other pixel removed).

Use the ENABLED property to customize the applicable items of a form. Set this property only in PRE–FORM, before the user begins a transaction. Do not set this property on a record level as it affects all records in the block; use the ENTERABLE property instead.

The behavior described here for ENABLED only occurs when you use APP_ITEM_PROPERTY at runtime; setting this property in the Oracle Forms Designer does not produce the same result.

Call APP_ITEM_PROPERTY.SET_PROPERTY(item_id, ENABLED, PROPERTY_OFF) to turn ENABLED off.

Setting	ENABLED	=	OFF
sets	NAVIGABLE	=	OFF
sets	UPDATEABLE	=	OFF
sets	INSERT_ALLOWED	=	OFF

ENTERABLE

ENTERABLE is an Oracle Applications item property used to control an item that is conditionally enterable, depending on other values in the record. It controls whether an item can be changed or navigated to. It uses a visual attribute applied at the record level to distinguish when an item is not enterable.

The user can click into a non–enterable item but cannot type any values. A non–enterable text or list item has black text on a gray background to indicate that it is non–enterable. Items other than text or list items do not get a visual attribute applied, and therefore do not visually indicate whether they are enterable.

Note that the APP_FIELD.SET_DEPENDENT_FIELD call turns the ENTERABLE property on and off for the dependent field.

Call APP_ITEM_PROPERTY.SET_PROPERTY(item_id, ENTERABLE, PROPERTY_ON) to turn ENTERABLE on.

Setting	ENTERABLE	=	ON
sets	NAVIGABLE	=	ON
sets	UPDATEABLE	=	ON
sets	UPDATE_NULL	=	OFF
sets	INSERT_ALLOWED	=	ON

ALTERABLE

ALTERABLE is an Oracle Applications item property used for display–only information when the user may need to scroll through or query on the information. It controls whether an item can be changed, regardless of whether the record is new or committed.

If an item is non–alterable, the user can enter and scroll through the item and query on the item but cannot type any values. There is no indicator that an item is non–alterable.

Call APP_ITEM_PROPERTY.SET_PROPERTY(item_id, ALTERABLE, PROPERTY_ON) to turn ALTERABLE on.

Setting	ALTERABLE	=	ON
sets	UPDATEABLE	=	ON
sets	UPDATE_NULL	=	OFF
sets	INSERT_ALLOWED	=	ON

NAVIGABLE

NAVIGABLE controls whether the cursor moves into an item when the user presses [TAB]. If the user clicks on a non–navigable item, the cursor moves into the item. There is no indicator that an item is non–navigable.

This property is used for mutually exclusive fields (by the APPCORE library procedure APP_FIELD.SET_EXCLUSIVE_FIELD). The null field is non–navigable, but the user can click into the field and enter a value, causing the field to become navigable and the other field to become null and non–navigable.

REQUIRED

Only use the REQUIRED property with text items and poplists that require the user to input data.

When setting this property in Oracle Forms Designer, override the Visual Attribute to be DATA_REQUIRED. This Visual Attribute is for future use.

Use APP_ITEM_PROPERTY.SET_PROPERTY to set this property dynamically since it is affected by other Oracle Applications item properties.

UPDATEABLE

UPDATEABLE controls whether an item in a queried or committed record can be changed.

Call APP_ITEM_PROPERTY.SET_PROPERTY(item_id, UPDATEABLE, PROPERTY_ON) to turn UPDATEABLE on.

Setting	UPDATEABLE	=	ON
sets	UPDATE_NULL	=	OFF

UPDATE_NULL

UPDATE_NULL allows the user to update the item if and only if the item is currently null.

UPDATE_NULL acts as the inverse of UPDATEABLE; they are mutually exclusive properties. Turning one on turns the other one off.

Call APP_ITEM_PROPERTY.SET_PROPERTY(item_id, UPDATE_NULL, PROPERTY_ON) to turn UPDATE_NULL on.

Setting	UPDATE_NULL	=	ON
sets	UPDATEABLE	=	OFF

INSERT_ALLOWED

INSERT_ALLOWED controls whether an item in a new, uncommitted record can be changed.

MOUSE_NAVIGATE

MOUSE_NAVIGATE controls whether clicking on a button, check box, list or radio group causes navigation to the item. If the user clicks on a mouse–navigable item, the cursor moves into the item. If the user clicks on a non–mouse–navigable item, the item is pressed but the cursor remains in the current item and navigation triggers do not fire.

In general, MOUSE_NAVIGATE should be off.

Buttons (See page 6 – 10)

ICON_NAME

Use APP_ITEM_PROPERTY.SET_PROPERTY to set ICON_NAME because the Oracle Applications call appends the icon name to the .dll on Windows, saving future icon load time.

Setting Visual Attributes Programatically

Unlike most Oracle Applications visual attributes that are applied as part of a property class or are applied automatically by APPCORE routines, the following visual attributes must be applied programatically by the developer.

DATA_DRILLDOWN

If you dynamically allow the drilldown capability in your block, you should apply the DATA_DRILLDOWN visual attribute programmatically.

DATA_SPECIAL

DATA_SPECIAL applies the color red on white to a field that needs special emphasis because it contains a value that violates a business rule or requires the user's close attention. For example, negative values in financial forms should be red on white. This visual attribute is ordinarily only applied at runtime.

Warning: Any use of color coding should augment an indicator that functions in a monochrome environment.

DATA_REQUIRED

Oracle Applications do not use the DATA_REQUIRED visual attribute.

CHAPTER

7

Controlling Window, Block and Region Behavior

This section describes coding standard behaviors for windows and alternative regions.

The following topics are covered:

- Controlling Records in a Window
- Duplicating Records
- Renumbering All Records in a Window
- Passing Instructions to a Form
- Navigation Between Forms
- Controlling Window Behavior
- Controlling Block Behavior
- Coding Master–Detail Relations
- Implementing a Combination Block
- Coding Alternative Region Behavior

Controlling Window Behavior

Controlling window behavior includes coding logic that positions windows upon opening, controlling which windows close under various conditions, providing context–sensitive titles for detail windows, and so on. If you have master–detail relationships between blocks in separate windows, you must also code logic for that situation.

Coding Master–Detail Relations (See page 7 – 6)

Positioning Windows Upon Opening

Example The Purchase Order header window contains a button labeled ’Lines’ that leads to the LINES block in a different window.

Step 1 Add or modify the following triggers:

Trigger: PRE–FORM

```
app_window.set_window_position('HEADER','FIRST_WINDOW');
```

Trigger: WHEN–BUTTON–PRESSED on the LINES button

```
app_custom.open_window('LINES');
```

Step 2 Modify APP_CUSTOM.OPEN_WINDOW as follows:

```
IF wnd = 'LINES' THEN
   APP_WINDOW.SET_WINDOW_POSITION('LINES',
                              'CASCADE','HEADER');
   go_block('LINES');
END IF;
```

The styles available are:

- CASCADE: Child window overlaps the parent window, offset to the right and down by 0.3” from the current position of the parent window. Usually used for detail windows.
- RIGHT, BELOW: Child window opens to the right of, or below, the parent window without obscuring it.
- OVERLAP: Detail window overlaps the parent window, aligned with its left edge, but offset down by 0.3”.
- CENTER: Window opens centered relative to another window. Usually used for modal windows.

- FIRST_WINDOW: Position the window immediately below the toolbar. Usually used for the main entity window.

Closing Windows

The window close events for all non–modal windows (but no modal windows) get passed to APP_CUSTOM.CLOSE_WINDOW. The default code provided in the TEMPLATE form does the following:

- If the form is in enter–query mode, APP_CUSTOM calls

  ```
  APP_EXCEPTION.DISABLED
  ```

- Otherwise, if the cursor is currently in the window to be closed, APP_CUSTOM issues a do_key('PREVIOUS_BLOCK') to attempt to move the cursor out of the current window
- Finally, APP_CUSTOM hides the window with a call to HIDE_WINDOW('<window_name>').

You need to modify this procedure to account for any other behaviors you require. Specifically, modify it to handle block coordination issues and detail windows.

Remember that you must move the cursor out of the window before closing it, otherwise the window reopens automatically.

To close the first window of a form, which is equivalent to 'Action–>Close Form' call APP_WINDOW.CLOSE_FIRST_WINDOW.

Example In a form with windows 'Header,' 'Lines,' and 'Shipments,' where Lines is a detail of Header, and Shipments is a detail of Lines, the logic to close the windows is as follows:

```
PROCEDURE close_window (wnd VARCHAR2) IS
   IF wnd = 'HEADER' THEN
      --
      -- Exit the form
      --
      app_window.close_first_window;
   ELSIF wnd = 'LINES' THEN
      --
      -- Close detail windows (Shipments)
      --
      app_custom.close_window('SHIPMENTS');
      --
      -- If cursor is in this window,
```

```
      -- move it to the HEADER block
      --
      IF (wnd = GET_VIEW_PROPERTY(GET_ITEM_PROPERTY(
          :SYSTEM.CURSOR_ITEM,ITEM_CANVAS),
          WINDOW_NAME)) THEN
         GO_BLOCK('HEADER');
      END IF;
   ELSIF wnd = 'SHIPMENTS' THEN
      --
      -- If cursor is in this window,
      -- move it to the LINES block
      --
      IF (wnd = GET_VIEW_PROPERTY(GET_ITEM_PROPERTY(
           :SYSTEM.CURSOR_ITEM, ITEM_CANVAS),
           WINDOW_NAME)) THEN
         GO_BLOCK('LINES');
      END IF;
   END IF;
   --
   -- THIS CODE MUST REMAIN HERE.  It ensures
   -- the cursor is not in the window that will
   -- be closed by moving it to the previous block.
   --
   IF (wnd = GET_VIEW_PROPERTY(GET_ITEM_PROPERTY(
        :SYSTEM.CURSOR_ITEM, ITEM_CANVAS),
         WINDOW_NAME)) THEN
      DO_KEY('PREVIOUS_BLOCK');
   END IF;
   --
   -- Now actually close the designated window
   --
   HIDE_WINDOW(wnd);
END close_window;
```

Master–Detail Relations (See page 7 – 6)

Warning: You must leave the default clause that attempts to move the cursor and close the window name passed to this procedure.

Setting Window Titles Dynamically

Dynamic Title Example

In the Enter Journal form, show the current Set of Books and Journal name in the Journal Lines window.

Step 1 Set the Lines window title to 'Journal Lines' in the Oracle Forms Designer.

Step 2 Code the PRE–RECORD trigger of the Journal block:

```
app_window.set_title('LINES', name_in('Journal.SOB'),
                     :journal.name);
```

Step 3 Code the WHEN–VALIDATE–ITEM trigger of the journal.names field:

```
app_window.set_title('LINES', name_in('Journal.SOB'),
                     :journal.name);
```

If you need to change the base title of a window, call SET_WINDOW_PROPERTY(...TITLE...). Any future calls to APP_WINDOW.SET_TITLE preserve your new base title.

Warning: Do not include the characters "(" or ":" in any of your window titles. These characters get added by the APPCORE window titling routine when you need to dynamically change the title to show context. Your base window titles should never include these characters. If you use a hyphen ("–"), do not surround it with spaces. In other words, do not both precede and follow your hyphen with spaces.

Coding Master–Detail Relations

Coordination Between Windows

When a detail block is in a different window than its master, and each window is non–modal, then the detail block must provide a mechanism for the user to toggle between immediate and deferred coordination. This allows a user to keep a block visible, but control the performance cost of coordinating detail records when the master record is changed.

When a detail block is not visible, its coordination should always be deferred. Use the procedure APP_WINDOW.SET_COORDINATION to coordinate master–detail blocks in different windows.

APP_WINDOW: Window Utilities (See page 26 – 27)

The sample code below uses the following objects:

- Master block ’ORDERS’, in window ’ORDERS’
- Detail Block ’LINES’, in window ’LINES’
- Relation ’ORDERS_LINES’
- Coordination check box ’CONTROL.ORDERS_LINES’
- Button to navigate to the LINES block ’CONTROL.LINES’

Step 1 Create a button to navigate to the detail block.

Step 2 Create a coordination check box in a control block in the detail window to specify the user’s preference of immediate or deferred coordination when the window is open. The check box should have the CHECKBOX_COORDINATION property class, which provides a value of ”IMMEDIATE” when checked and ”DEFERRED” when unchecked. The check box value should default to checked (IMMEDIATE).

Step 3 Create your item handler procedures as follows:

```
PACKAGE BODY control IS

        PROCEDURE lines(EVENT VARCHAR2) IS
        BEGIN
           IF (EVENT = ’WHEN–BUTTON–PRESSED’) THEN
              app_custom.open_window(’LINES’);
```

```
      END IF;
   END lines;

   PROCEDURE orders_lines(EVENT VARCHAR2) IS
   BEGIN
      IF (EVENT = 'WHEN-CHECKBOX-CHANGED') THEN
         APP_WINDOW.SET_COORDINATION(EVENT,
                                     :control.orders_lines,
                                     'ORDERS_LINES');
      END IF;
    END orders_lines;

END control;
```

Step 4 Customize the APP_CUSTOM template package as follows:

In the OPEN_WINDOW procedure, add:

```
IF (WND = 'LINES') THEN
   APP_WINDOW.SET_COORDINATION('OPEN-WINDOW',
                               :control.orders_lines,
                               'ORDERS_LINES');
   GO_BLOCK('LINES');
END IF;
```

In the CLOSE_WINDOW procedure, add:

```
IF (WND = 'LINES') THEN
  APP_WINDOW.SET_COORDINATION('WHEN-WINDOW-CLOSED',
                              :control.orders_lines,
                              'ORDERS_LINES');
END IF;
```

Step 5 Call your field and event handler procedures in:

Trigger: WHEN–BUTTON–PRESSED on control.lines:

```
control.lines('WHEN-BUTTON-PRESSED');
```

Trigger: KEY–NXTBLK on ORDER:

```
control.lines('WHEN-BUTTON-PRESSED');
```

Trigger: WHEN–CHECKBOX–CHANGED on control.order_lines:

```
control.orders_lines('WHEN-CHECKBOX-CHANGED');
```

Implementing a Combination Block

Each item in a block can have its own Items Displayed property, so you can have a single block in which some items are single–record ('Detail') and some are multi–record ('Summary'). When you implement a combination block, most items appear twice, so coordination of the values in these items must be managed. The Mirror Item property does this automatically. You control which portion of the block is navigated to in different situations using a field called 'Switcher'. The Switcher field is the first navigable item in the block. When the cursor enters the Switcher, it is immediately moved to the first item in either the Detail or Summary portion of the block.

Step 1 Setting up the combination block

Create two windows and canvasses to hold the different portions of your block. Use the non–mirror items in your block for the Summary portion. Duplicate the items to create the Detail portion. The Detail portion of your combination block should be sequenced first. Thus, when the user does not fill in a required item and tries to commit the block, Oracle Forms positions the cursor in that item in the Detail block.

Step 2 Setting the item properties

For the mirror items, change the item names to reflect the real item that they are mirroring (for example, name the mirror item of "status" to "status_mir"). Set the Mirror Item property, and make sure the Base Table property is set to TRUE (if the mirrored items are a base table item).

Set the block–level Records Displayed property for your Summary portion. This will get picked up by the items so long as you do not explicitly set the Items Displayed property. So that your Detail portion items do not get the same value, explicitly set their Items Displayed property to 1.

To prevent the user from tabbing out of the Detail and into the Summary, set the Previous Navigation Item property for the first Detail item, and the Next Navigation Item property for the last Detail item.

To enforce the standard multi–record block navigation behavior of Change Record, call APP_COMBO.KEY_PREV_ITEM in the KEY–PREV–ITEM (Fire in ENTER–QUERY mode: False) trigger of the first navigable item of the Summary portion, and call `next_record` in the KEY–NEXT–ITEM trigger (Fire in ENTER–QUERY mode: False) of the last navigable item of the Summary portion.

APP_COMBO: Combination Block API (See page 26 – 2)

If you are converting an existing block into a combination block, do not forget to change references in any existing triggers to recognize that there are now two instances of every field.

Step 3 The Drilldown Record Indicator

Add a Drilldown Record Indicator that does an `execute_trigger('SUMMARY_DETAIL')`.

Step 4 The Record Count Parameter

Create a parameter to store the record count for the portion of the block you are currently in. Name the parameter <block>_RECORD_COUNT, where <block> is the name of the combination block. The APPCORE code depends on this naming standard. This information is used to determine which portion of the block to navigate to. The parameter should have a datatype of NUMBER and a default value of 2, so that the cursor is initially in the Summary portion. (If you want the cursor to start in the Detail portion, set the default value to 1).

Create a block level WHEN–NEW–ITEM–INSTANCE trigger (Execution Style: Before) that contains the following code:

```
:PARAMETER.<block>_RECORD_COUNT :=
GET_ITEM_PROPERTY(:SYSTEM.CURSOR_ITEM,
                      RECORDS_DISPLAYED);
```

Step 5 The Switcher

Create a text item and assign it the Property Class 'SWITCHER'. It needs to be the lowest sequenced item in the block. Place it at (0,0) on the Toolbar canvas (The Switcher belongs on the Toolbar canvas because whatever canvas it is on paints.). Create an item–level WHEN–NEW–ITEM–INSTANCE trigger (Execution Style: Override) that contains the following code:

```
IF(:PARAMETER.<block>_RECORD_COUNT > 1) THEN
  GO_ITEM('<first Summary field>');
ELSE
  APP_WINDOW.SET_WINDOW_POSITION('<Detail window>',
                          'OVERLAP',
                         '<Summary window>');
```

```
    GO_ITEM('<first Detail field>');
  END IF;
```

Step 6 The Summary / Detail Menu Item

Create a block–level SUMMARY_DETAIL trigger (Execution Style: Override) that contains the following code:

```
IF GET_ITEM_PROPERTY(:SYSTEM.CURSOR_ITEM,
                     RECORDS_DISPLAYED) > 1 THEN
   :PARAMETER.<block>_RECORD_COUNT := 1;
ELSE
   :PARAMETER.<block>_RECORD_COUNT := 2;
END IF;
GO_ITEM('<block>.Switcher');
```

This code changes the value in the RECORDS_DISPLAYED parameter so that the Switcher sends the cursor into the opposite portion of the block. It will fire whenever the user chooses "Go –> Summary / Detail."

Create a block–level PRE–BLOCK trigger (Execution Style: Override) that contains the following code:

```
APP_SPECIAL.ENABLE('SUMMARY_DETAIL', PROPERTY_ON);
```

Finally, create a form–level PRE–BLOCK trigger (Execution Style: Override) that contains the code:

```
APP_SPECIAL.ENABLE('SUMMARY_DETAIL', PROPERTY_OFF);
```

If all blocks are combination blocks, you can turn on SUMMARY_DETAIL at the form–level and ignore the PRE–BLOCK trigger. If most blocks are combination blocks, you can turn SUMMARY_DETAIL on at the form–level, and disable it at the block–level for those blocks that are not combination blocks.

Step 7 Initial navigation and window operations

If your combination block is the first block in the form, position the two windows in the PRE–FORM trigger with the following calls:

```
APP_WINDOW.SET_WINDOW_POSITION('<Summary window>',
                                'FIRST_WINDOW');
APP_WINDOW.SET_WINDOW_POSITION('<Detail window>',
                                'OVERLAP',
                                '<Summary window>');
```

Usually, the Summary is entered first, but there are cases where it is dynamically determined that the Detail should be entered first. If you need to dynamically decide this, set the parameter <block>_RECORD_COUNT in the PRE–FORM trigger (1 to send it to the Detail, 2 to send it to the Summary).

Coding Alternative Region Behavior

Alternative Regions

A block with multiple regions that cannot be rendered simultaneously uses a series of stacked canvasses to display each region, one at a time, within a single region boundary. These stacked regions are called ”Alternative Regions”.

For more information, see the *Oracle Applications User Interface Standards.*

Alternative Regions
Oracle Applications User Interface Standards

Each alternative region has a poplist control element containing all possible regions for that block.

Behavior of the Alternative Region Poplist

Alternative region poplists should behave according to the following standards:

- The Alternative Region poplist should have the Queryable attribute set to TRUE, so that it can be used while the block is in ENTER–QUERY mode.
- KEY–MENU invokes an LOV that allows the user to select from the same list of choices as in the control poplist. The title of this LOV is ”Regions.” You need this function for keyboard compatibility because the control poplist is not otherwise accessible except via the mouse.

Example: Coding an Alternative Region

Block LINES has some fields on a content canvas ORDER. The last of these fields is ITEM.

LINES has alternative regions on canvasses LINES_PRICE and LINES_ITEM. The regions are accessible only if LINES.ITEM is not null. The first item of LINES_PRICE is LIST_PRICE. The first item of LINES_ITEM is DESCRIPTION.

Step 1 Create a poplist in a control block to select the current region. The poplist should be queryable and non–navigable. The poplist should display a friendly name for each region with a corresponding value equal to the region's canvas name.

The block CONTROL has a queryable, non–navigable poplist named LINES_REGIONS (block name plus _REGIONS) that contains the following values, with the internal value following the displayed value: Price Information (LINES_PRICE), Item Information (LINES_ITEM).

Step 2 Visit the CONTROL block:

At form startup, you must visit the block containing the control poplist to instantiate it:

- Create a text item called DUMMY as the first item in the CONTROL block. Make the text item Displayed, Enabled and Navigable, Position 0,0, WD=0, HT=0, and +–Length=1. Place it on the first canvas to be displayed.
- In WHEN–NEW–FORM–INSTANCE, make two GO_BLOCK() calls, one to the CONTROL block and another to the First Navigation Block.
- Make sure you do similar GO_BLOCK calls in the code where you handle KEY–CLRFRM.

Step 3 Setting the First Displayed Region:

Within Oracle Forms Designer, designate the first stacked canvas of the set of alternative regions to show as displayed; make all other canvasses in the set not displayed to improve startup performance.

You must sequence the stacked canvasses carefully by ordering the canvasses within the list in the Oracle Forms Object Navigator (the first stacked canvas in the list is the first stacked canvas displayed). In addition, you must sequence your items to have the correct order when a user tabs through the fields on the alternative regions.

Suggestion: When stacked canvasses are referenced, their sequence may be unpredictable. In this case, issue a SHOW_VIEW at form startup, or whenever the window is first displayed, to force the proper canvas to render.

Make sure your stacked canvas views are all exactly the same size and occupy exactly the same space on the content canvas.

Step 4 Create your item handler procedures to control which region displays as in the following example. Remember, in our example form, we want to disallow access to the regions unless the field LINES.ITEM is not null:

```
PACKAGE BODY control IS

  g_canvas_name  VARCHAR2(30) := null;

  PROCEDURE lines_regions(event varchar2) IS
    target_canvas_name VARCHAR2(30);
    curr_canvas_name VARCHAR2(30) :=
                    get_item_property(:system.cursor_item,
                                      ITEM_CANVAS);
  BEGIN
    IF (event = 'WHEN-NEW-ITEM-INSTANCE') THEN
      -- Check if the poplist and the canvas are out of synch
      -- to prevent flashing if they are not.
      IF  ((curr_canvas_name in ('LINES_PRICE', 'LINES_ITEM')) AND
           (curr_canvas_name != :control.lines_regions)) THEN
        :control.lines_regions := curr_canvas_name;
        g_canvas_name := curr_canvas_name;
      END IF;
    ELSIF (event = 'WHEN-LIST-CHANGED') THEN
      target_canvas_name := :control.lines_regions;

      -- The following is optional code to disallow access
      -- to certain regions under certain conditions
      -- Check that the region is accessible.  Always allow access
      -- during queries.
      IF (:SYSTEM.MODE = 'ENTER-QUERY') THEN
        null;
      ELSE
        IF (:lines.item is null) THEN
          FND_MESSAGE.SET_NAME('OE', 'OE_ENTER_ITEM_FIRST');
          FND_MESSAGE.ERROR;
          :control.lines_regions := g_canvas_name;
          RAISE FORM_TRIGGER_FAILURE;
        END IF;
   -- End of optional code

      END IF;
      -- Display the region.  If in the same block, go to the
      -- first item in the region.
        IF curr_canvas_name in ('LINES_PRICE', 'LINES_ITEM') THEN
          hide_view(curr_canvas_name);
        END IF;
        show_view(target_canvas_name);
        IF (:system.cursor_block = 'LINES') THEN
          IF (target_canvas_name = 'LINES_PRICE') THEN
            -- Go to the first item in the canvas LINES_PRICE
```

```
            go_item('lines.list_price');
          ELSIF (target_canvas_name = 'LINES_ITEM') THEN
            -- Go to the first item in the canvas LINES_ITEM
            go_item('lines.description');
          END IF;
        END IF;
        g_canvas_name := target_canvas_name;
    ELSE
        fnd_message.debug('Invalid event passed to
              control.lines_regions');
    END IF;
  END lines_regions;
END control;
```

After the user displays the LOV via KEY–MENU and chooses a value from the list, the WHEN–LIST–CHANGED handler switches the regions.

Step 5 Call the following triggers:

Trigger: Block–level WHEN–NEW–ITEM–INSTANCE on the LINES block:

```
CONTROL.LINES_REGIONS('WHEN-NEW-ITEM-INSTANCE');
```

Trigger: Block–level KEY–MENU on the LINES block (Execution Style: Override):

```
IF APP_REGION.ALT_REGIONS('CONTROL.LINES_REGIONS') THEN
  CONTROL.LINES_REGIONS('WHEN-LIST-CHANGED');
END IF;
```

Trigger: Item–level WHEN–LIST–CHANGED on CONTROL.LINES_REGIONS.

```
CONTROL.LINES_REGIONS('WHEN-LIST-CHANGED');
```

These triggers should fire in ENTER–QUERY mode.

Controlling Records in a Window

This section discusses

- Duplicating Records
- Renumbering All Records in a Window

Duplicating Records

Why Duplicate Record is Disabled by Default

By default, duplicate record is disabled at the form level. There are several reasons for this:

- The special column ROW_ID is duplicated and must be manually exempted if it exists
- The record is marked as valid even through the items may contain time–sensitive data that is no longer valid
- Defaults are overridden
- In many blocks, Duplicate Record makes no sense (modal dialogs, find blocks, etc.)

For any block where you want to enable Duplicate Record, you must write code. You must process unique keys, possibly reapply defaults, and confirm that copied data is still valid. None of this is done by default, and this can lead to errors or data corruption.

In general, duplicate all item values, even if the item value must be unique. The user may wish to create a unique value very similar to the previous value.

Do not override a default if

- The item cannot be modified by the user
- Or, the item must contain a specific value for a new record
- Or, the item is a sequential number and the default is the correct value most of the time

Example

A block *order* has items *order_number* and *order_date* which are defaulted from the sequence *order_S* and from SYSDATE respectively, and which cannot be modified by the user. The item *status* should contain 'Open' for a new order, but the user can change the Status to ’Book’ at any time to book the order.

Step 1 Create your event handler procedures as follows:

```
PACKAGE BODY order IS
   PROCEDURE KEY_DUPREC IS
   CURSOR new_order_number IS SELECT order_S.nextval
                              FROM sys.dual;
  BEGIN
    DUPLICATE_RECORD;
    open new_order_number;
    fetch new_order_number into :order.order_number;
    close new_order_number;
    :order.status : = ’Open’;
    :order.order_date := FND_STANDARD.SYSTEM_DATE;
    :order.row_id := null;
  END KEY_DUPREC;
END order;
```

Step 2 Call your event handler procedures in:

Trigger: KEY–DUPREC on order:

```
order.KEY_DUPREC;
```

Renumbering All Records in a Window

To renumber an item sequentially for all records on the block, create a user–named trigger to increment the sequence variable and set the sequence item. Use the procedure APP_RECORD.FOR_ALL_RECORDS to fire the trigger once for each record.

To number an item sequentially as records are created, create a variable or item to contain the sequence number. Create a WHEN–CREATE–RECORD trigger to increment the sequence variable and default the sequence item. However, if you want to renumber all the records in a window, use the procedure APP_RECORD.FOR_ALL_RECORDS.

If you are renumbering your items after a query or commit, you may wish to reset the record status so that the record is not marked as changed.

Example

A block *lines* has item *line_number*. When a record is deleted, *line_number* must be renumbered.

Step 1 Create your item handler procedures as follows:

```
PACKAGE BODY lines IS

  line_number_seq number := 0;

  PROCEDURE delete_row IS
  BEGIN
    line_number_seq := 0;
    APP_RECORD.FOR_ALL_RECORDS('reseq_line_number');
  END delete_row;

END lines;
```

Step 2 Create a user–defined trigger RESEQ_LINE_NUMBER as follows:

```
lines.line_number_seq := lines.line_number_seq + 1;
:lines.line_number := lines.line_number_seq;
```

Step 3 Call your item handler procedures in:

Trigger: KEY–DELETE:

```
lines.line_number('KEY-DELETE');
```

Warning: Be aware of the consequences of this type of processing. Specifically, consider the following points:

If a very large number of records are queried in a block, looping through them one at a time can be very slow.

Not all the records that could be in the block may be in the current query set if you allow the user to enter the query.

If you are changing a value that is part of a unique key, you may get errors at commit time because the record currently being committed may conflict with another already in the database, even though that record has also been changed in the block.

Navigation Between Forms

This section discusses controlling navigation between forms.

Passing Instructions to a Form

To pass information when navigating from one form to another when both forms are already open, use the WHEN–FORM–NAVIGATE trigger. You do not code this trigger directly; instead pass the information through global variables.

To use this trigger, populate a global variable called GLOBAL.WHEN_FORM_NAVIGATE with the name of a user–named trigger. When a form is navigated to, this trigger fires.

The WHEN–FORM–NAVIGATE trigger fires upon programmatically navigating to a form using the GO_FORM built–in. Accordingly, this trigger is referenced into all forms.

Querying an Item

It often makes sense to navigate to a form and query on a specific item. For example, suppose you have an Order Entry form ORDERS and a Catalog form CATALOGS. You want to navigate from the ORDERS form to CATALOGS and query up a specific part number.

- In the ORDERS form, create a global variable called GLOBAL.PART_NUMBER, and populate it with the value you want to query.
- In the ORDERS form, create a global variable called GLOBAL.WHEN_FORM_NAVIGATE. Populate this variable with the string ”QUERY_PART_NUMBER”.
- Create a user–named trigger in the CATALOGS form, ”QUERY_PART_NUMBER”. In this trigger, enter query mode by calling EXECUTE_QUERY.
- Create a PRE–QUERY trigger in the CATALOGS form that calls `copy(’GLOBAL.PART_NUMBER, ’PARTS_BLOCK.PART_NUMBER’).` Then call `copy(’’,’GLOBAL.PART_NUMBER’).` When there is a value in GLOBAL.PART_NUMBER, it becomes part of the query criteria.

CHAPTER

8

Enabling Query Behavior

This section describes how you build Row–LOVs and Find windows and enable Query Find functionality for your form.

The following topics are covered:

- Implementing Row–LOVs
- Implementing Find Windows

Overview of Query Find

There are two implementations for Query Find. One implementation shows a Row–LOV that shows the available rows and allows you to choose one. The other implementation opens a Find window, which shows you the fields the user is likely to want to use for selecting data.

Use only one implementation for a given block. All queryable blocks within your form should support Query Find. The *Oracle Applications User Interface Standards* describe what situations are better served by the two implementations.

APP_FIND: Query Find APIs (See page 26 – 14)

Query Find
Oracle Applications User Interface Standards

Raising Query Find on Form Startup

If you want a Row–LOV or Find window to raise immediately upon entering the form, at the end of your WHEN–NEW–FORM–INSTANCE trigger, call:

```
EXECUTE_TRIGGER('QUERY_FIND');
```

This will simulate the user invoking the function while in the first block of the form.

Implementing Row–LOV

To implement a Row–LOV, create an LOV that selects the primary key of the row the user wants into a form parameter, and then copy that value into the primary key field in the results block right before executing a query.

This example uses the DEPT block, which is based on the DEPT table, and consists of the three columns DEPTNO, DNAME and LOC. This table contains a row for each department in a company.

Step 1 Create a Parameter for Your Primary Key

Create a form parameter(s) to hold the primary key(s) for the LOV. If the Row–LOV is for a detail block, you do not need a parameter for the foreign key to the master block (the join column(s)), as you should include that column in the WHERE clause of your record group in a later step. Set the datatype and length appropriately.

For example, for the DEPT block, create a parameter called DEPTNO_QF.

Step 2 Create an LOV

Create an LOV that includes the columns your user needs to identify the desired row. If the Row–LOV is for a detail block, you should include the foreign key to the master block (the join column(s)) in the WHERE clause of your record group. Return the primary key for the row into the parameter.

For our example, create an LOV, DEPT_QF, that contains the columns DEPTNO and DNAME. Set the return item for DEPTNO into parameter DEPTNO_QF. Although the user sees DNAME , it is not returned into any field.

Step 3 Create a PRE–QUERY Trigger

Create a block–level PRE–QUERY trigger (Execution Style: Before) that contains:

```
IF :parameter.G_query_find = 'TRUE' THEN
  <Primary Key> := :parameter.<Your parameter>;
  :parameter.G_query_find := 'FALSE';
END IF;
```

For multi–part keys, you need multiple assignments for the primary key.

The parameter G_query_find exists in the TEMPLATE form.

For the Dept example, your PRE–QUERY trigger contains:

```
IF :parameter.G_query_find = 'TRUE' THEN
  :DEPT.DEPTNO := :parameter.DEPTNO_QF
  :parameter.G_query_find := 'FALSE';
END IF;
```

Step 4 Create a QUERY_FIND Trigger

Finally, create a block–level user–named trigger QUERY_FIND on the results block (Execution Style: Override) that contains:

```
APP_FIND.QUERY_FIND('<Your LOV Name>');
```

For DEPT:

```
APP_FIND.QUERY_FIND('DEPT_QF');
```

Implementing Find Windows

To implement a Find window, create an additional window that contains the fields a user is most likely to search by when they initiate the search and copy all the item values from that block into the results block just before executing a query.

In this example, there is a block based on the EMP table. This is referred to as the results block. The primary key for this table is EMPNO. This block also contains the date field HIREDATE. The Find window is designed to locate records by EMPNO or a range of HIREDATES.

For more information on the look and feel of Find windows, see the *Oracle Applications User Interface Standards*.

Find Windows
Field Ranges (From / To Field Pairs)
Oracle Applications User Interface Standards

Flexfields in Find windows require special treatment.

Using Key Flexfields in Find Windows (See page 14 – 63)

To code a Find window, follow the steps below.

Step 1 Copy the QUERY_FIND Object Group from APPSTAND

Copy the QUERY_FIND object group from the APPSTAND form to your form. It contains a window, a block and a canvas from which to start building your Find window.

After you copy it, delete the object group. This leaves the window, canvas and block, but allows you to copy the object group again if you need another Find window.

Warning: DO NOT REFERENCE THIS OBJECT GROUP; you need to customize it.

Step 2 Rename the Block, Canvas and Window

Rename the Find Block, Canvas, and Window. Set the queryable property of the block to FALSE.

For this example, rename the block, canvas and window to EMP_QF, EMP_QF_CANVAS, and EMP_QF_WINDOW, respectively.

Step 3 Edit the NEW Button's Trigger

Edit the WHEN–BUTTON–PRESSED trigger for the NEW button in the Find window block so that it passes the Results block name as the argument. This information allows Oracle Applications to navigate to your block and place you on a new record. This button is included because when you first enter a form, the Find window may automatically come up; users who want to immediately start entering a new record can press this button.

```
app_find.new('<Your results blockname here>');
```

becomes

```
app_find.new('EMP');
```

Step 4 Edit the FIND Button's Trigger

Edit the WHEN–BUTTON–PRESSED trigger for the FIND button so that it passes the Results block name. This information allows Oracle Applications to navigate to your block and execute a query.

```
app_find.find('<Your results blockname here>');
```

becomes

```
app_find.find('EMP')
```

If you need to do further validation of items in the Find window, place your code before the call to APP_FIND.FIND. Specifically, you should validate that any low / high range fields are correct. You may also give a warning if no criteria has been entered at all, or if the criteria entered may take a very long time to process.

Step 5 Set Navigation Block Properties

Set the Previous Navigation Block property of the Find block to be the results block. This allows the user to leave the Find window without executing a query.

From the results block, next and previous block only move up and down the hierarchy of objects; they never take you to the Find window.

Step 6 Edit the KEY–NXTBLK Trigger

Edit the KEY–NXTBLK trigger on the Find block so that it has the exact same functionality as the FIND button. If the user selects ”Go–>Next Block,” the behavior should mimic pressing the FIND button.

Step 7 Change the Find Window Title

Change the title of the Find window.

The EMP example uses ”Find Employees”.

Step 8 Create Necessary Items

Create the items that the user can query on in the Find window block. You may find it convenient to copy items from the Results block to the Find window block.

Follow these guidelines for items in the Find window:

- Set the Required property to FALSE
- Set the default value to NULL
- If you copied the items from the Results block, ensure that your new items all have Base Table Item set to False, and remove all triggers associated with them (especially validation triggers). If for some reason you decide you need to keep a particular trigger, remember to change the fields it references to point to the Find block.
- Typically, an item in the Find window block has an LOV associated with it, because users should usually be able to select

exactly one valid value for the item. The LOV should show all values that have ever been valid, not just those values that are currently valid. Date fields may use the Calendar and the related KEY–LISTVAL trigger.

- If you have an item that has a displayed value and an associated ID field, the Find window block should have both as well. The ID field should be used to drive the query to improve performance.
- Items that are check boxes or option groups in the Results block should be poplists in the Find window block. When they are NULL, no restriction is imposed on the query.

Step 9 Fit the Find Window to Your Form

Adjust your Find window for your specific case: resize the window, position, fields, and so on.

Step 10 Create a PRE–QUERY Trigger

Create a block–level Pre–Query trigger in the Results block (Execution Style: Before) that copies query criteria from the Find window block to the Results block (where the query actually occurs).

You can use the Oracle Forms COPY built–in to copy character data. For other data types, you can assign the values directly using :=, but this method does not allow the user to use wildcards. However, most of your Find window items use LOVs to provide a unique value, so wildcards would not be necessary.

```
IF :parameter.G_query_find = 'TRUE' THEN
   COPY (<find Window field>,'<results field>');
   :parameter.G_query_find := 'FALSE';
END IF;
```

A commonly used 'special criteria' example is to query on ranges of numbers, dates, or characters. The APP_FIND.QUERY_RANGE procedure is defined to take care of the query logic for you. Pass in the low and high values as the first two arguments, and the name of the database field actually being queried on as the third argument.

In our EMP example,

```
IF :parameter.G_query_find = 'TRUE' THEN
   COPY(:EMP_QF.EMPNO, 'EMP.EMPNO');
   APP_FIND.QUERY_RANGE(:EMP_QF.Hiredate_from,
                        :EMP_QF.Hiredate_to,
```

```
                     'EMP.Hiredate');
   :parameter.G_query_find := 'FALSE';
END IF;
```

- Your base table field query length (in the Results block) must be long enough to contain the query criteria. If it is not, you get an error that the value is too long for your field. All fields should have a minimum query length of 255.
- If you have radio groups, list items, or check boxes based on database fields in your Results block, you should only copy those values from the Find window if they are not NULL.

Step 11 Create a QUERY_FIND Trigger

Create a block–level user–named trigger "QUERY_FIND" (Execution Style: Override) on the Results block that contains:

```
APP_FIND.QUERY_FIND('<results block window>',
           '<Find window>',
           '<Find window block>');
```

In our EMP example:

```
APP_FIND.QUERY_FIND('EMP_WINDOW', 'EMP_QF_WINDOW',
          'EMP_QF');
```

CHAPTER

9

Coding Item Behavior

This section describes the following topics:

- Item Relations
- Defaults
- Integrity Checking
- The Calendar
- CALENDAR: Calendar Package

Item Relations

There are many behaviors in complex forms that must be enforced dynamically at runtime, either to adhere to the field–level validation model of Oracle Applications, or to enforce specific business rules.

- Dependent Items
- Conditionally Dependent Items
- Multiple Dependent Items
- Two Master Items and One Dependent Items
- Cascading Dependence
- Mutually Exclusive Items
- Mutually Inclusive Items
- Mutually Inclusive Items with Dependents
- Conditionally Mandatory Items

You should model your form's item and event handlers after these examples.

Disabled Items and WHEN–VALIDATE–ITEM Trigger

In most of the item relations you are dynamically disabling and enabling items. For your disabled items, note these Oracle Forms coding issues:

- WHEN–VALIDATE–ITEM always fires the first time a user TABs through each field on a brand new record, even if they do not make a change. Internally Oracle Forms notes that the value changes from unknown to null, therefore it fires WHEN–VALIDATE–ITEM. Also, WHEN–VALIDATE–ITEM fires when a user changes a field from a non–null value to null.

  Furthermore, a user can leave a required field null at any time; it is only trapped at record level. Therefore, all WHEN–VALIDATE–ITEM triggers must account for the value of the field being null, and act accordingly. Since you cannot distinguish between the user changing the value to null, or Oracle Forms setting the value to null the first time, both must behave as if the user changed the value.

- Most of the time, a disabled item has a null value. Since you account for nulls because of the previous issue, this is not a problem. In those rare cases that a disabled field has a value, and that value got set while it was disabled and the field has not been validated yet, you may need to add logic to WHEN–VALIDATE–ITEM to do nothing.

Dependent Items

To create a text item, check box, or poplist that is enabled only when a master item is populated, use the procedure APP_FIELD.SET_DEPENDENT_FIELD. This routine enforces the following behaviors:

- The dependent item is either cleared or made invalid when the master item changes.
- If the master item is NULL or the condition is FALSE, the dependent item is disabled.

Create the item handler procedures as shown below and then call the procedures from the specified triggers.

Attention: These routines do not apply to display–only text items. To conditionally grey out display–only text items, use the routine APP_ITEM_PROPERTY.SET_VISUAL_ATTRIBUTE.

APP_ITEM_PROPERTY: Property Utilities (See page 26 – 19)

In this example, a block *order* has items *item_type* and *item_name*. Item_name is dependent on item_type, thus item_name is enabled only when item_type is NOT NULL.

Step 1 Create your item handler procedures as follows:

```
PACKAGE BODY ORDER IS

  PROCEDURE ITEM_TYPE(EVENT VARCHAR2) IS
  BEGIN
    IF (EVENT = 'WHEN-VALIDATE-ITEM') THEN
      --- Any validation logic goes here.
      ITEM_NAME('INIT');
    ELSE
       fnd_message.debug('Invalid event passed to
       ORDER.ITEM_TYPE: ' || EVENT);
    END IF;
  END ITEM_TYPE;
```

```
PROCEDURE ITEM_NAME(EVENT VARCHAR2) IS
BEGIN
  IF ((EVENT = 'PRE-RECORD') OR
      (EVENT = 'INIT')) THEN
    APP_FIELD.SET_DEPENDENT_FIELD(EVENT,
                              'ORDER.ITEM_TYPE',
                              'ORDER.ITEM_NAME');
  ELSE
     fnd_message.debug('Invalid event passed to
     ORDER.ITEM_NAME: ' || EVENT);
  END IF;
END ITEM_NAME;
END ORDER;
```

Step 2 Call your item handler procedures in:

Trigger: WHEN–VALIDATE–ITEM on item_type:

```
order.item_type('WHEN-VALIDATE-ITEM');
```

Trigger: PRE–RECORD on order (Fire in Enter Query Mode: False):

```
order.item_name('PRE-RECORD');
```

Step 3 If your master and dependent items are in a multi–row block, or if they are items in a single–row block that is a detail of a master block, you must call SET_DEPENDENT_FIELD for the POST–QUERY event as well.

```
PROCEDURE ITEM_NAME(EVENT VARCHAR2) IS
        BEGIN
          IF ((EVENT = 'PRE-RECORD') OR
              (EVENT = 'INIT') OR
              (EVENT = 'POST-QUERY')) THEN
             APP_FIELD.SET_DEPENDENT_FIELD(EVENT,
                                      'ORDER.ITEM_TYPE',
                                      'ORDER.ITEM_NAME');
          ELSE
             fnd_message.debug('Invalid event passed to
             ORDER.ITEM_NAME: ' || EVENT);
          END IF;
        END ITEM_NAME;
```

Add another call to your item handler procedure in:

Trigger: POST–QUERY

```
ORDER.ITEM_NAME('POST-QUERY');
```

Attention: In a multi–record block, if the dependent item is the last item in the record, the cursor navigates to the next record when tabbing from the master. To work around this behavior, code a KEY–NEXT–ITEM trigger that does a VALIDATE(Item_scope) and then a NEXT_ITEM.

Attention: If the dependent item is a required list or option group, set the 'invalidate' parameter in the call to APP_FIELD.SET_DEPENDENT_FIELD to TRUE. When this flag is TRUE, the dependent item is marked as invalid rather than cleared.

Conditionally Dependent Item

A conditionally dependent item is enabled or disabled depending on the particular value of the master item. In this example, the block *order* has items *item_type* and *item_size*. Item_size is enabled only when item_type is 'SHOES'.

Step 1 Create your item handler procedures as follows. Note that this item handler is very similar to the simple master / dependent situation, but you specify the condition instead of the name of the master item.

```
PACKAGE BODY order IS

 PROCEDURE ITEM_TYPE(EVENT VARCHAR2) IS
 BEGIN
   IF (EVENT = 'WHEN-VALIDATE-ITEM') THEN
     size('INIT');
   ELSE
      fnd_message.debug('Invalid event passed to
        ORDER.ITEM_TYPE: ' || EVENT);
   END IF;
 END item_type;

 PROCEDURE size(EVENT VARCHAR2) IS
 BEGIN
   IF ((EVENT = 'PRE-RECORD') OR
       (EVENT = 'INIT')) THEN
     APP_FIELD.SET_DEPENDENT_FIELD(EVENT,
                                  (:order.item_type = 'SHOES'),
                                   'ORDER.SIZE');
   ELSE
      fnd_message.debug('Invalid event passed to
        ORDER.SIZE: ' || EVENT);
   END IF;
```

```
  END size;
END order;
```

Step 2 Call your item handler procedures in:

Trigger: PRE–RECORD on order (Fire in Enter Query Mode: False):

```
order.item_size('PRE-RECORD');
```

Trigger: WHEN–VALIDATE–ITEM on item_type:

```
order.item_type('WHEN-VALIDATE-ITEM');
```

Multiple Dependent Items

There are cases where multiple items are dependent on a single master item. For example, only certain item_types can specify a color and size. Therefore, the color and size fields are dependent on the master field item_type, and they are enabled only when item_type is RAINCOAT.

Step 1 Create your item handler procedures as follows:

```
PACKAGE BODY order IS

     PROCEDURE item_type(EVENT VARCHAR2) IS
     BEGIN
       IF (EVENT = 'WHEN-VALIDATE-ITEM') THEN
        color('INIT');
        size('INIT');
       ELSE
          fnd_message.debug('Invalid event passed to
            ORDER.ITEM_TYPE: ' || EVENT);
       END IF;
     END item_type;

     PROCEDURE color(EVENT VARCHAR2) IS
     BEGIN
       IF (EVENT = 'PRE-RECORD') OR
          (EVENT = 'INIT') THEN
           APP_FIELD.SET_DEPENDENT_FIELD(EVENT,
                                 (:order.item_type = 'RAINCOAT'),
                                  'ORDER.COLOR');
       ELSE
          fnd_message.debug('Invalid event passed to
            ORDER.COLOR: ' || EVENT);
       END IF;
     END color;
```

```
     PROCEDURE size(EVENT VARCHAR2) IS
     BEGIN
       IF (EVENT = 'PRE-RECORD') OR
          (EVENT = 'INIT') THEN
         APP_FIELD.SET_DEPENDENT_FIELD(EVENT,
                                 (:order.item_type = 'RAINCOAT'),
                                  'ORDER.SIZE');
       ELSE
          fnd_message.debug('Invalid event passed to
            ORDER.SIZE: ' || EVENT);
       END IF;
     END size;

END order;
```

Step 2 Call your item handler procedures in:

Trigger: WHEN–VALIDATE–ITEM on order.item_type:

```
order.item_type('WHEN-VALIDATE-ITEM');
```

Trigger: PRE–RECORD (Fire in Enter Query Mode: False):

```
order.color('PRE-RECORD');
order.size('PRE-RECORD');
```

Two Master Items and One Dependent Item

There may also be cases where an item is dependent on two master items. Suppose that different sizes of sweaters come in different colors. You cannot fill in the color of the sweater until you have filled in both item_type and size. The validation of block.dependent is controlled by the content of both master_1 and master_2.

Step 1 Create your item handler procedures as follows:

```
PACKAGE BODY order IS

         PROCEDURE item_type(EVENT VARCHAR2) IS
         BEGIN
           IF (EVENT = 'WHEN-VALIDATE-ITEM') THEN
             color('INIT'):
       ELSE
          fnd_message.debug('Invalid event passed to
            ORDER.ITEM_TYPE: ' || EVENT);
           END IF;
         END item_type;
```

```
    PROCEDURE size(EVENT VARCHAR2) IS
    BEGIN
      IF (EVENT = 'WHEN-VALIDATE-ITEM') THEN
        color('INIT');
  ELSE
     fnd_message.debug('Invalid event passed to
       ORDER.SIZE: ' || EVENT);
      END IF;
    END size;

    PROCEDURE color(EVENT VARCHAR2) IS
    BEGIN
      IF (EVENT = 'PRE-RECORD') OR
         (EVENT = 'INIT') THEN
         APP_FIELD.SET_DEPENDENT_FIELD(EVENT,
               ((:order.item_type IS NOT NULL) AND
                (:order.size IS NOT NULL)),
                 'ORDER.COLOR');
      ELSE
       fnd_message.debug('Invalid event passed to
       ORDER.COLOR: ' || EVENT);
      END IF;
    END color;

END order;
```

Step 2 Call your item handler procedures in:

Trigger: WHEN–VALIDATE–ITEM on order.item_type:

```
order.item_type('WHEN-VALIDATE-ITEM');
```

Trigger: WHEN–VALIDATE–ITEM on order.size:

```
order.size('WHEN-VALIDATE-ITEM');
```

Trigger: PRE–RECORD (Fire in Enter Query Mode: False):

```
order.color('PRE-RECORD');
```

Cascading Dependence

With cascading dependence, *item_3* depends on *item_2*, which in turn depends on *item_1*. Usually all items are in the same block.

For example, the block *order* contains the items *vendor*, *site*, and *contact*.

The list of valid sites depends on the current vendor.

- Whenever vendor is changed, site is cleared.
- Whenever vendor is null, site is disabled.

The list of valid contacts depends on the current site.

- Whenever site is changed, contact is cleared.
- Whenever site is null, contact is disabled.

To code the correct behavior for these dependent items, follow these steps.

Step 1 Create your item handler procedures as follows:

```
PACKAGE BODY order IS

    PROCEDURE vendor(EVENT VARCHAR2) IS
    BEGIN
       IF (EVENT = 'WHEN-VALIDATE-ITEM') THEN
          SITE('INIT');
       ELSE
          fnd_message.debug('Invalid event passed to
            ORDER.VENDOR: ' || EVENT);
       END IF;
    END VENDOR;

    PROCEDURE SITE(EVENT VARCHAR2) IS
    BEGIN
       IF (EVENT = 'WHEN-VALIDATE-ITEM') THEN
          CONTACT('INIT');
       ELSIF (EVENT = 'PRE-RECORD') OR
             (EVENT = 'INIT') THEN
          APP_FIELD.SET_DEPENDENT_FIELD(EVENT,
                                        'ORDER.VENDOR',
                                        'ORDER.SITE');
          CONTACT(EVENT);
       ELSE
          fnd_message.debug('Invalid event passed to
            ORDER.SITE: ' || EVENT);
       END IF;
    END SITE;

    PROCEDURE CONTACT(EVENT VARCHAR2) IS
    BEGIN
       IF (EVENT = 'PRE-RECORD') OR
          (EVENT = 'INIT') THEN
          APP_FIELD.SET_DEPENDENT_FIELD(EVENT,
                                       'ORDER.SITE',
                                       'ORDER.CONTACT');
       ELSE
```

```
          fnd_message.debug('Invalid event passed to
            ORDER.CONTACT: ' || EVENT);
      END IF;
   END CONTACT;

END ORDER;
```

Step 2 Call your item handler procedures in:

Trigger: WHEN–VALIDATE–ITEM on vendor:

```
order.vendor('WHEN-VALIDATE-ITEM');
```

Trigger: WHEN–VALIDATE–ITEM on site:

```
order.site('WHEN-VALIDATE-ITEM');
```

Trigger: PRE–RECORD on order (Fire in Enter Query Mode: False):

```
order.site('PRE-RECORD');
order.contact('PRE-RECORD');
```

Remember that the following chain of events occurs whenever the VENDOR field is validated:

- VENDOR is validated, which calls SITE ('INIT').
- SITE ('INIT') causes the state of SITE to change and calls CONTACT ('INIT').
- CONTACT ('INIT') causes the state of CONTACT to change.

Mutually Exclusive Items

Use the procedure APP_FIELD.SET_EXCLUSIVE_FIELD to code two items where only one item is valid at a time.

The key to coding an item handler procedure for mutually exclusive items is to realize that mutually exclusive items are logically one item. Whenever one of a pair of mutually exclusive items is dependent on or depended upon by another item, they both are. Their relationship to other items is always identical. Therefore, code a single item handler procedure for the single logical item.

If both mutually exclusive items are NULL, then both items are navigable. If one item is populated, then the other item is unnavigable (you can still click there), and any value in that item is cleared.

If one item *must* be not null, set the REQUIRED property of both items to be on in the Oracle Forms Designer. If both items may be null, set the REQUIRED property of both items to be off. APP_FIELD.SET_EXCLUSIVE_FIELD reads the initial REQUIRED property and dynamically manages the REQUIRED properties of both items.

You can also use the procedure APP_FIELD.SET_EXCLUSIVE_FIELD for a set of three mutually exclusive items. For more than three items, you must write your own custom logic.

☞ **Attention:** Mutually exclusive check boxes and required lists require mouse operations.

For example, a block *lines* has mutually exclusive items *credit* and *debit*.

Step 1 Create your item handler procedures as follows:

```
PACKAGE BODY lines IS

   PROCEDURE credit_debit(EVENT VARCHAR2) IS
   BEGIN
      IF ((EVENT = 'WHEN-VALIDATE-ITEM') OR
          (EVENT = 'PRE-RECORD')) THEN
         APP_FIELD.SET_EXCLUSIVE_FIELD(EVENT,
                                       'LINES.CREDIT',
                                       'LINES.DEBIT');
      ELSIF (EVENT = 'WHEN-CREATE-RECORD') THEN
         SET_ITEM_PROPERTY('lines.credit', ITEM_IS_VALID,
                           PROPERTY_TRUE);
         SET_ITEM_PROPERTY('lines.debit', ITEM_IS_VALID,
                           PROPERTY_TRUE);
      ELSE
        fnd_message.debug('Invalid event passed to
        Lines.credit_debit: ' || EVENT);
      END IF;
   END credit_debit;

END lines;
```

Step 2 Call your item handler procedures in:

Trigger: WHEN–VALIDATE–ITEM on credit:

```
lines.credit_debit('WHEN-VALIDATE-ITEM');
```

Trigger: WHEN–VALIDATE–ITEM on debit:

```
lines.credit_debit('WHEN-VALIDATE-ITEM');
```

Trigger: PRE–RECORD on lines (Fire in Enter Query Mode: False):

```
lines.credit_debit('PRE-RECORD');
```

Trigger: WHEN–CREATE–RECORD on lines:

```
lines.credit_debit('WHEN-CREATE-RECORD');
```

You only need the WHEN–CREATE–RECORD trigger if the resulting one of your mutually–exclusive fields is required. This trigger initially sets all the mutually–exclusive fields of the set to be required. The fields are then reset appropriately once a user enters a value in one of them.

Mutually Inclusive Items

Use APP_FIELD.SET_INCLUSIVE_FIELD to code a set of items where, if any of the items is not null, all of the items are required.

The item values may be entered in any order. If all of the items are null, then the items are optional.

You can use the procedure APP_FIELD.SET_INCLUSIVE_FIELD for up to five mutually inclusive items. For more than five items, you must write your own custom logic.

This example shows a block *payment_info* with mutually inclusive items *payment_type* and *amount*.

Step 1 Create your item handler procedures as follows:

```
PACKAGE BODY payment_info IS

   PROCEDURE payment_type_amount(EVENT VARCHAR2) IS
   BEGIN
      IF ((EVENT = 'WHEN-VALIDATE-ITEM') OR
          (EVENT = 'PRE-RECORD')) THEN
         APP_FIELD.SET_INCLUSIVE_FIELD(EVENT,
                                 'PAYMENT_INFO.PAYMENT_TYPE',
                                 'PAYMENT_INFO.AMOUNT');
      ELSE
         fnd_message.debug('Invalid event to
           payment_info.payment_type_ amount: ' || EVENT);
      END IF;
   END payment_type_amount;

END payment_info;
```

Step 2 Call your item handler procedures in:

Trigger: WHEN–VALIDATE–ITEM on payment_info.payment_type:

```
payment_info.payment_type_amount('WHEN-VALIDATE-ITEM');
```

Trigger: WHEN–VALIDATE–ITEM on payment_info.amount:

```
payment_info.payment_type_amount('WHEN-VALIDATE-ITEM');
```

Trigger: PRE–RECORD on payment_info (Fire in Enter Query Mode: False):

```
payment_info.payment_type_amount('PRE-RECORD');
```

Mutually Inclusive Items with Dependent Items

There are cases where items are dependent on master items, where the master items are mutually inclusive.

Item Relations (See page 9 – 2)

This example shows a block *payment_info* with mutually inclusive items *payment_type* and *amount*, just as in the previous example. The block also contains two regions, one for check information and one for credit card information. Check Information has a single item, *check_number.* Credit Card Information has five items: *credit_type, card_holder, number, expiration_date,* and *approval_code.*

Payment Type can be Cash, Check, or Credit.

- When Payment Type is Check, the Check Information region is enabled.
- When Payment Type is Credit, the Credit Card Information region is enabled.

Step 1 Create your item handler procedures as follows:

```
PACKAGE BODY payment_info IS

   PROCEDURE payment_type_amount(EVENT VARCHAR2) IS
   BEGIN
      IF (EVENT = 'WHEN-VALIDATE-ITEM') THEN
         APP_FIELD.SET_INCLUSIVE_FIELD(EVENT,
                                  'PAYMENT_INFO.PAYMENT_TYPE',
                                  'PAYMENT_INFO.AMOUNT');
         IF (:SYSTEM.CURSOR_ITEM =
             'payment_info.payment_type') THEN
```

```
         check_info('INIT');
         credit_info('INIT');
      END IF;

   ELSIF (EVENT = 'PRE-RECORD') THEN
      APP_FIELD.SET_INCLUSIVE_FIELD(EVENT,
                              'PAYMENT_INFO.PAYMENT_TYPE',
                              'PAYMENT_INFO.AMOUNT');
   ELSE
      fnd_message.debug('Invalid event in
        payment_info.payment_type_amount: ' || EVENT);
   END IF;
END payment_type_amount;


PROCEDURE check_info IS
BEGIN
   IF ((EVENT = 'PRE-RECORD') OR
       (EVENT = 'INIT')) THEN
      APP_FIELD.SET_DEPENDENT_FIELD(EVENT,
        (:payment_info.payment_type = 'Check'),
        'PAYMENT_INFO.CHECK_NUMBER');
   ELSE
      fnd_message.debug('Invalid event in
         payment_info.check_info: ' || EVENT);
   END IF;
END check_info;

PROCEDURE credit_info IS
   CONDITION BOOLEAN;
BEGIN
   IF ((EVENT = 'PRE-RECORD') OR
       (EVENT = 'INIT')) THEN
      CONDITION := (:payment_info.payment_type = 'Credit');
      APP_FIELD.SET_DEPENDENT_FIELD(EVENT,
                               CONDITION,
                              'PAYMENT_INFO.CREDIT_TYPE');
      APP_FIELD.SET_DEPENDENT_FIELD(EVENT,
                               CONDITION,
                               'PAYMENT_INFO.NUMBER');
      APP_FIELD.SET_DEPENDENT_FIELD(EVENT,
                               CONDITION,
                               'PAYMENT_INFO.CARD_HOLDER');
      APP_FIELD.SET_DEPENDENT_FIELD(EVENT,
                               CONDITION,
                              'PAYMENT_INFO.EXPIRATION_DATE');
      APP_FIELD.SET_DEPENDENT_FIELD(EVENT,
                              CONDITION,
                              'PAYMENT_INFO.APPROVAL_CODE');
```

```
        ELSE
          fnd_message.debug('Invalid event in
               payment_info.credit_info: ' || EVENT);
        END IF;
    END credit_info;

END payment_info;
```

Step 2 Call your item handler procedures in:

Trigger: WHEN–VALIDATE–ITEM on payment_info.payment_type:

```
payment_info.payment_type_amount('WHEN–VALIDATE–ITEM');
```

Trigger: WHEN–VALIDATE–ITEM on payment_info.amount:

```
payment_info.payment_type_amount('WHEN–VALIDATE–ITEM');
```

Trigger: PRE–RECORD on payment_info (Fire in Enter Query Mode: False):

```
payment_info.payment_type_amount('PRE–RECORD');
payment_info.check_info('PRE–RECORD');
payment_info.credit_info('PRE–RECORD');
```

Conditionally Mandatory Items

Use the procedure APP_FIELD.SET_REQUIRED_FIELD to code an item that is only mandatory when a certain condition is met. If the condition is FALSE, the dependent item is optional. Any value in the dependent item is not cleared. If an item is both conditionally required and dependent, call APP_FIELD.SET_DEPENDENT_FIELD before calling APP_FIELD.SET_REQUIRED_FIELD.

An example demonstrates using APP_FIELD.SET_REQUIRED_FIELD.

A block *purchase_order* has items *total* and *vp_approval*. Vp_approval is required when total is more than $10,000. (Note: *quantity* * *unit_price* = *total*).

Step 1 Create your item handler procedures as follows:

```
PACKAGE BODY purchase_order IS

    PROCEDURE vp_approval(EVENT VARCHAR2) IS
    BEGIN
       IF ((EVENT = 'PRE-RECORD') OR
           (EVENT = 'INIT')) THEN
          APP_FIELD.SET_REQUIRED_FIELD(EVENT,
                                (:purchase_order.total > 10000),
                                 'PURCHASE_ORDER.VP_APPROVAL');
       ELSE
          fnd_message.debug('Invalid event in
            purchase_order.vp_approval: ' || EVENT);
       END IF;
    END vp_approval;

    PROCEDURE total(EVENT VARCHAR2) IS
    BEGIN
       IF (EVENT = 'INIT') THEN
          :purchase_order.total := :purchase_order.quantity *
                                   :purchase_order.unit_price;
          vp_approval('INIT');
       ELSE
          fnd_message.debug('Invalid event in purchase_order.total:
' || EVENT);
    END total;

    PROCEDURE quantity(EVENT VARCHAR2) IS
    BEGIN
       IF (EVENT = 'WHEN-VALIDATE-ITEM') THEN
          total('INIT');
       ELSE
          fnd_message.debug('Invalid event in
            purchase_order.quantity: ' || EVENT);
       END IF;
    END quantity;

    PROCEDURE unit_price(EVENT VARCHAR2) IS
    BEGIN
       IF (EVENT = 'WHEN-VALIDATE-ITEM') THEN
          total('INIT');
       ELSE
          fnd_message.debug('Invalid event in
            purchase_order.unit_price: ' || EVENT);
       END IF;
    END unit_price;

END purchase_order;
```

Step 2 Call your item handler procedures in:

Trigger: PRE–RECORD on purchase_order (Fire in Enter Query Mode: False):

```
purchase_order.vp_approval('PRE–RECORD');
```

Trigger: WHEN–VALIDATE–ITEM on quantity:

```
purchase_order.quantity('WHEN–VALIDATE–ITEM');
```

Trigger: WHEN–VALIDATE–ITEM on unit_price:

```
purchase_order.unit_price('WHEN–VALIDATE–ITEM');
```

Defaults

Defaults on a New Record

To default values when the user first creates a new record, use the Default values property in the Oracle Forms Designer. For more complex defaulting behavior, follow the example below.

Step 1 Create your event handler procedure as follows:

```
PACKAGE block IS

    PROCEDURE WHEN_CREATE_RECORD IS
    BEGIN
       :block.item1 := default_value1;
       :block.item2 := default_value2;
           ...
    END WHEN_CREATE_RECORD;

END block;
```

Step 2 Call your event handler procedures in:

Trigger: WHEN–CREATE–RECORD:

```
block.WHEN_CREATE_RECORD;
```

Applying Defaults While Entering a Record

When you want to set a default for an item whose validation depends on another item (for example, to apply the default when the master value changes), set the default values in the dependent item's INIT event.

Integrity Checking

This section discusses how to handle:

- Uniqueness Checks
- Referential Integrity Checks

Uniqueness Check

To do a uniqueness check for a key, use a select statement that is invoked by the WHEN–VALIDATE–ITEM event.

Note that a uniqueness check done in WHEN–VALIDATE–ITEM does not catch duplicates residing in uncommitted rows (for instance, a user enters uncommitted, duplicate rows in a detail block). The database constraints will catch this situation, as well as the situation where an identical key is committed by someone else between the time that the when–validate–item fired and your record is committed. For this reason, you do not need to write a uniqueness check in PRE–UPDATE or PRE–INSERT.

- If there is a single unique key field, always call the CHECK_UNIQUE package from WHEN–VALIDATE–ITEM for that field.
- If the unique combination is comprised of multiple fields, call the CHECK_UNIQUE package from the WHEN–VALIDATE–RECORD trigger.

Example:

```
PROCEDURE CHECK_UNIQUE(X_ROWID VARCHAR2,
          pkey1 type1, pkey2 type2, ...) IS
  DUMMY NUMBER;
BEGIN
   SELECT COUNT(1)
      INTO DUMMY
      FROM table
   WHERE pkeycol1 = pkey1
    AND pkeycol2 = pkey2
      ...
    AND ((X_ROWID IS NULL) OR (ROWID != X_ROWID));
   IF (DUMMY >= 1) then
       FND_MESSAGE.SET_NAME('prod', 'message_name');
      APP_EXCEPTION.RAISE_EXCEPTION;
```

```
      END IF;
   END CHECK_UNIQUE;
```

Create your item handler procedure as follows:

```
   PACKAGE BODY block IS
      PROCEDURE item(EVENT VARCHAR2) IS
      BEGIN
         IF (EVENT = 'WHEN-VALIDATE-ITEM') THEN
           table_PKG.CHECK_UNIQUE(:block.row_id,
                     :block.pkey1, :block.pkey2, ...);
         ELSE
            message('Invalid event in block.item');
         END IF
      END item;
   END block;
```

Referential Integrity Check

When deleting a record, you must be concerned about the existence of other records that may be referencing that record. For example, if an item has already been placed on a Purchase Order, what should occur when you attempt to delete the item? Three possible answers are:

- Don't allow the item to be deleted.
- Also delete the Purchase Order.
- Allow the item to be deleted, and null out the reference to it on the Purchase Order.

Most the time, the first solution is both the most practical and sensible. To do this, create a procedure that detects these referenced cases, and raise an exception.

Giving Warning Before Deleting Details

To give a warning when detail records will be deleted, create CHECK_REFERENCES as a function which returns FALSE if detail records exist (CHECK_REFERENCES should still raise an exception if deleting the row would cause a referential integrity error).

If a table contains subtypes, you must decide whether you need one CHECK_REFERENCES procedure or one CHECK_REFERENCES procedure per subtype.

If the subtypes share most of the foreign key references with some subtype–specific foreign key references, then create just one CHECK_REFERENCES procedure with the first parameter a subtype discriminator.

If the subtypes are orthogonal, then create a CHECK_subtype_REFERENCES procedure for each subtype.

Example Referential Integrity Check

Step 1 Create your table handler procedures as follows:

```
CREATE OR REPLACE PACKAGE BODY table_PKG AS

  PROCEDURE CHECK_REFERENCES(pkey1 type1, pkey2 type2, ...)
IS
    MESSAGE_NAME VARCHAR2(80);
    DUMMY        credit;
  BEGIN
    MESSAGE_NAME := 'message_name1';
    SELECT 1 INTO DUMMY FROM DUAL WHERE NOT EXISTS
      (SELECT 1 FROM referencing_table1
       WHERE ref_key1 = pkey1
         AND ref_key2 = pkey2
         ...
      );
    MESSAGE_NAME := 'message_name2';
    SELECT 1 INTO DUMMY FROM DUAL WHERE NOT EXISTS
      (SELECT 1 FROM referencing_table2
       WHERE ref_key1 = pkey1
         AND ref_key2 = pkey2
         ...
      );
    ...

  EXCEPTION
    WHEN NO_DATA_FOUND THEN
      FND_MESSAGE.SET_NAME('prod', MESSAGE_NAME);
      APP_EXCEPTION.RAISE_EXCEPTION;
  END CHECK_REFERENCES;

END table_PKG;
```

Step 2 Create your event handler procedures as follows:

```
PACKAGE BODY block IS
  PROCEDURE key_delete IS
  BEGIN
```

```
    --
    -- First make sure its possible to delete this record.
    -- An exception will be raised if its not.
    --
    table_PKG.CHECK_REFRENCES(pkey1, pkey2, ...);
    --
    -- Since it is possible to delete the row, ask the
    -- user if they really want to,
    -- and delete it if they respond with 'OK'.
    --
    app_record.delete_row;
  END key_delete;
END block;
```

Step 3 Call the event handler:

Trigger: KEY–DELETE:

```
block.dey_delete;
```

Suggestion: You should do similar steps again with the ON–DELETE trigger. It is possible that between the time a user requested the delete, and actually saved the transaction, a record was entered elsewhere that will cause referential integrity problems. Remember that KEY–DELETE fires in response to the user initiating a delete, but it does not actually perform the delete; it just flags the record to be deleted and clears it from the screen. The ON–DELETE trigger fires at commit time and actually performs the delete.

The Calendar

The Calendar is a standard object that allows selection of date and time values from a Calendar. It also allows the developer to specify validation rules ensuring that only valid dates can be selected. Both the List and Edit functions should invoke the Calendar on any date field.

For each date field within a form, you should provide the code necessary for the user to call the Calendar feature. However, the calendar is not a replacement for validating the data in the field.

The Calendar is automatically included in the TEMPLATE form.

For more information on the user interface standards for the Calendar, see the *Oracle Applications User Interface Standards*.

Calendar
Oracle Applications User Interface Standards

LOV for Date and Datetime Fields

Date and datetime fields should enable the List lamp. When the user invokes List on these fields, the form opens the Calendar window.

Date fields should use the ENABLE_LIST_LAMP LOV, which is included in the TEMPLATE form. This setting enables the menu and Toolbar List of Values entries for your date fields. Set 'Use LOV for Validation' to FALSE on fields that use this LOV. If you leave 'Use LOV for Validation' set to TRUE, you will see an LOV that has no columns.

Required Calls

Each date field within a form needs to have the following code:

Trigger: KEY–LISTVAL:

```
calendar.show([first_date]);
```

By default, the Calendar shows the month of the value in the date field (if a value exists) upon first opening. If no specific date is supplied, the Calendar shows the current month.

Do not pass the current field into CALENDAR.SHOW as a parameter, as this forces validation of the field. The current field is used a default. Generally, the code in KEY–LISTVAL should be:

```
calendar.show;
```

☞ **Attention:** Never pass the value of the current date field as the argument to CALENDAR.SHOW. Because the calendar actually disables all Oracle Forms validation momentarily, if the user has entered an invalid date then immediately invokes the calendar, a PL/SQL error occurs. SHOW automatically handles this case when no argument is passed to it.

The KEY–LISTVAL trigger must have execution style ’Override’, and should not fire in enter–query mode.

CALENDAR: Calendar Package (See page 9 – 28)

Display Only Mode

The entire calendar can be run in a display–only mode, where it is used to show one or more dates as Selected, rather than allowing the user to select a particular date. For example, it can be used to show all dates on which an employee was absent.

In this mode, characteristics of the field the cursor is on are ignored. All the user can do is change the month and year shown, and press ’OK’ to close the window (no value is ever written back to the form).

To invoke this mode, the following calls are required in addition to those listed above:

Trigger: KEY–LISTVAL:

```
calendar.setup('DISPLAY');
calendar.setup('TITLE', null, null,
               '<translated text for window title>');
```

Additional CALENDAR.SETUP calls are required after these 2 calls to establish those dates that should be shown as selected.

Advanced Calendar Options

You can incorporate optional features into your Calendar call. If you use any of the optional calls, they must be placed *before* the mandatory `calendar.show` call.

The following examples customize the Calendar to show or disable specific dates.

Disable Weekends in the Calendar Window

To disable weekends (where the weekend is defined as Saturday and Sunday):

```
calendar.setup('WEEKEND');
```

Disable Specific Date Ranges

To disable specific date ranges, where the dates are either hardcoded or references to other fields on the form:

```
calendar.setup(<30 char identifying name>, <low_date>,
                 <high_date>);
```

This call can be repeated as many times as needed. A null LOW_DATE is treated as the beginning of time; a null HIGH_DATE is treated as the end of time.

Disable Specific Date Ranges From a Table

To disable specific date ranges, where the dates are contained in a table:

```
calendar.setup(<30 char identifying name>, null, null,
<SQL>);
```

This call may only be made once per field, but may return multiple rows. A null LOW_DATE is treated as the beginning of time; a null HIGH_DATE is treated as the end of time. Use NVL in your SQL statement if this is not the desired behavior.

Restrictions from several tables can be performed by using UNION SQL statements. The selected columns *must* be aliased to LOW_DATE and HIGH_DATE.

Suggestion: Ordering on the LOW_DATE column may improve performance. Restricting the dates returned to a small range near the anticipated selected value also improves performance.

Calling the Calendar from non–DATE fields

If you need to be able to activate the Calendar from a field that is not explicitly declared as a DATE or DATETIME field (such as a CHAR text item that serves multiple purposes depending on context), write the Calendar calls as normal. The Calendar acts as if invoked from a DATE field, and when the user selects a value the date is written back to the field in the format 'DD–MON–YYYY'.

Then user–named trigger ’CALENDAR_WROTE_DATE’ fires. Create that trigger at the item level, and add any code you need to process the value (typically you need to apply a mask to it).

Calendar Examples

Example – Weekdays Only

In this example, you want to open the Calendar to show either the date currently displayed in the DATE item, or the current month if no date is displayed. Additionally, you want to disable weekends (Saturdays and Sundays).

Trigger: KEY–LISTVAL:

```
calendar.setup('WEEKEND');
calendar.show;
```

Example – Only Include Workdays

In a form with the field SHIP_BY_DATE, you want to open the Calendar and customize it to:

- Disable all holidays defined in the ORG_HOLIDAYS table
- Disable weekends
- Show the month corresponding to the date in field ’LINES.NEED_BY_DATE’ when the Calendar is opened

The code to implement this is:

Trigger: KEY–LISTVAL:

```
calendar.setup('WEEKEND');
calendar.setup('Manufacturing Holidays', null, null,
                 'select action_date LOW_DATE,
                 action_date HIGH_DATE '||
                 'from org_holidays where
                 date_type = ''HOLIDAY''');
calendar.show(:lines.need_by_date);
```

Example – Range of Days Enabled

In a form with a field NEED_BY_DATE, you want the Calendar to show the month corresponding to the date in the field

LINES.CREATED_DATE + 30 days. You also want to disable all dates before and including :LINES.CREATED_DATE.

The code to implement this is:

Trigger: KEY–LISTVAL:

```
calendar.setup('After created date', null,
                  lines.created_date);
calendar.show(:lines.need_by_date + 30);
```

Example – Display Only Calendar

A form uses a button called "Holidays" to show all Manufacturing holidays. The current month displays initially, and the calendar finds the selected dates in the ORG_DATES table.

The code to implement this is:

Trigger: WHEN–BUTTON–PRESSED on HOLIDAYS:

```
calendar.setup('TITLE', null, null,
          '<translated text for "Manufacturing Holidays">');
calendar.setup('Manufacturing Holidays', null, null,
     'select action_date LOW_DATE, action_date HIGH_DATE '||
     'from  org_dates where date_type = ''HOLIDAY''');
calendar.show;
```

CALENDAR: Calendar Package

For standards and examples of coding calendars into your forms, see:

The Calendar (See page 9 – 23)

CALENDAR.SHOW

Syntax:

```
PROCEDURE  show (first_date date default null);
```

Description

This call shows the calendar. Do not pass the current field value into show; this value is used by default.

CALENDAR.SETUP

Syntax:

```
PROCEDURE  setup (new_type       varchar2,
                  low_date       date     DEFAULT null,
                  high_date      date     DEFAULT null,
                  sql_string     varchar2 DEFAULT null);
```

CALENDAR.EVENT

Syntax:

```
PROCEDURE  event (event varchar2);
```

CHAPTER

10

Controlling the Toolbar and the Default Menu

This chapter provides you with information you need to modify the Oracle Applications default menu (also known as the pulldown menus) and the toolbar.

The following topics are covered:

- Pulldown Menus and the Toolbar
- APP_SPECIAL: Menu and Toolbar Control

Pulldown Menus and the Toolbar

The Oracle Applications pulldown menus (the default menu) allow the user to invoke standard Oracle Forms functions, such as 'Clear Record' and 'List of Values' as well as application–specific functions.

For detailed listings of each pulldown menu, see the *Oracle Applications User Interface Standards*.

Pulldown Menus
Oracle Applications User Interface Standards

For a detailed list of each iconic button on the Toolbar, see the *Oracle Applications User Interface Standards*.

Toolbar
Oracle Applications User Interface Standards

Both the menu and the Toolbar are included in the TEMPLATE form. Entries on the menu and the Toolbar are disabled and enabled automatically based on the current context.

Menu and Toolbar Entries

Your menu and toolbar should react consistently to changes within your forms. Unless specified otherwise, the following behaviors come automatically from the form–level triggers embedded in your forms. The triggers that control the behavior appear with each entry (if applicable).

Menu and Toolbar Entries

In order as the toolbar lists them:

- Navigate To...(WHEN–NEW–RECORD–INSTANCE)

  Enabled except in called forms.

- Zoom (WHEN–NEW–BLOCK–INSTANCE)

  Enabled if the customer defines a zoom for the current block

- Save

  Always enabled.

- Save and Proceed

  Always enabled.

- Print...

  Always enabled.

- Clear Form

  Always enabled

- Summary / Detail

  Disabled by default; developer can enable / disable as required.

- Find... (WHEN–NEW–RECORD–INSTANCE)

  Enabled if the block allows querying and is not already in enter–query mode.

- New Record (WHEN–NEW–BLOCK–INSTANCE)

  Enabled if the block allows inserts.

- Delete Record (WHEN–NEW–RECORD–INSTANCE)

  Enabled if the block allows deletes.

- Clear Record

  Always enabled.

- Folder Tools

  Enabled if the cursor is in a folder block; developer must provide code in a combination folder block.

- Translations

  Disabled by default; developer can enable / disable as required.

- Attachments (WHEN–NEW–RECORD–INSTANCE and WHEN–NEW–BLOCK–INSTANCE)

  Icon set, and entry enabled based on the existence of attachment definitions and actual attachments.

- List of Values (WHEN–NEW–ITEM–INSTANCE)

  Enabled when the current item is a text item and has an LOV associated with it, and is either a date field or is changeable by the user (based on insert / update allowed properties of the block and field).

- Edit Field... (WHEN–NEW–ITEM–INSTANCE)

  Enabled when the current item is a text item.

- Window Help

Always enabled.

Menu Only Entries

In order of the menu, from Action to Help:

- Action, Exit Oracle Applications (WHEN–NEW–RECORD–INSTANCE)

  Enabled if not in enter–query mode.

- Edit, Cut / Copy / Paste

  These menu items are processed by Oracle Forms / Toolkit.

- Edit, Clear Field (WHEN–NEW–ITEM–INSTANCE)

  Enabled when the current item is a text item.

- Edit, Duplicate Field Above (WHEN–NEW–ITEM–INSTANCE)

  Enabled if the current record number is > 1.

- Edit, Duplicate Record Above (WHEN–NEW–RECORD–INSTANCE)

  Enabled if the current record number is > 1 and the record status is 'NEW'.

- Query, Find All (WHEN–NEW–RECORD–INSTANCE)

  Enabled if the block allows querying, and not already in enter–query mode.

- Query, Enter (WHEN–NEW–BLOCK–INSTANCE)

  Enabled if the block allows querying.

- Query, Run (WHEN–NEW–BLOCK–INSTANCE)

  Enabled if the block allows querying.

- Query, Cancel (WHEN–NEW–RECORD–INSTANCE)

  Enabled if in enter–query mode.

- Query, Show Last criteria (WHEN–NEW–RECORD–INSTANCE)

  Enabled if in enter–query mode.

- Query, Count Matching Records (WHEN–NEW–RECORD–INSTANCE)

  Enabled if the block allows querying.

- Query, Get More Records (WHEN–NEW–RECORD–INSTANCE)

Enabled if the block allows querying, and not already in enter–query mode.

- Go, Previous Record (WHEN–NEW–RECORD–INSTANCE)

  Enabled if the current record number is > 1.

- Go, Next Block (WHEN–NEW–BLOCK–INSTANCE)

  Enabled if the Next Navigation Block does not match the current block.

- Go, Previous Block (WHEN–NEW–BLOCK–INSTANCE)

  Enabled if the Previous Navigation Block does not match the current block.

- Go, First Record

  Enabled if the current record number is > 1.

- Help, About This Record

  Enabled if the current block has a base table or view. Disable this menu option if the underlying table has no WHO columns.

Dynamic Menu Control

You can use the APP_SPECIAL.ENABLE procedure to dynamically control menu items, if the behavior you need is not provided automatically. First, determine if the default menu control handles the menu item in question, and ensure that there really is a need to override the default behaviors.

If the menu item is not controlled by the default menu control, use any trigger (typically PRE–BLOCK or WHEN–NEW–BLOCK–INSTANCE), adding the code:

```
app_special.enable('the menu item', PROPERTY_OFF);
```

Turn the menu item back on when you leave (typically POST–BLOCK) by calling:

```
app_special.enable('the menu item', PROPERTY_ON);
```

Include the full name of the menu item in this call, for example:

```
app_special.enable('EDIT.CLEAR_FIELD', PROPERTY_OFF);
```

You can determine the full names of the menu items by copying FNDMENU from the `apps10/au10/res` area and opening the copy to examine the menu items.

If the menu item is controlled by the default menu control, use the trigger listed (either WHEN–NEW–BLOCK–INSTANCE, WHEN–NEW–RECORD–INSTANCE, or WHEN–NEW–ITEM–INSTANCE). Make the trigger style "Override" and add the following code:

```
app_standard.event('TRIGGER_NAME');
app_special.enable('Menu_item', PROPERTY_OFF);
```

The item will be correctly enabled in other blocks by the default menu control, so it is not necessary to enable it when leaving the block, record, or item.

APP_SPECIAL: Menu and Toolbar Control (See page 10 – 9)

Correcting Variations

The most common sources of variation from these behaviors are:

- The existence of a trigger at the item or block level which is not style 'Before', thus preventing the form–level trigger from also firing. Each entry above lists the event it must receive to be set properly (WHEN–NEW–BLOCK–INSTANCE, WHEN–NEW–RECORD–INSTANCE or WHEN–NEW–ITEM–INSTANCE).
- Incorrect settings for the block–level properties Next Navigation Block, Previous Navigation Block, and Query Allowed.
- Date and descriptive flexfield fields lacking the ENABLE_LIST_LAMP LOV.

If at any time you need to change a property and need to force a refresh of the menu (because the appropriate WHEN– trigger will not fire after the change you made), call APP_STANDARD.SYNCHRONIZE.

Blocks Where Only One Record Is Possible

You may want to disable some menu options for blocks in which only one record is possible. The Single Record Blocks section discusses when and how to do this.

Single Record Blocks (See page 5 – 13)

Save and Proceed

By default, this function performs a Save, then moves to the First Navigation Block of the form and proceeds to the next record. You can override this behavior.

Replace the code within the form–level ACCEPT trigger, or create a block–level ACCEPT trigger with execution style 'Override' that calls any of the following:

- APP_STANDARD.EVENT('ACCEPT') to get the default behavior
- APP_STANDARD.EVENT('ACCEPT:0') to get the default behavior, except that the cursor does not change blocks
- APP_STANDARD.EVENT('ACCEPT:<blockname>') to get default behavior except the cursor moves to the specified block.

Synchronizing

The Toolbar and menu are automatically updated by the standard form–level WHEN–NEW–RECORD–INSTANCE, WHEN–NEW–BLOCK–INSTANCE, and WHEN–NEW–ITEM–INSTANCE triggers. If you code these triggers at the item or block level, be sure to set execution style to 'Before.' If you change a property of a block or an item, the menu and Toolbar do not reflect this change until the appropriate trigger fires.

For example, if you turn the block property 'Insert Allowed' off on a block WHILE THE CURSOR IS ALREADY IN THAT BLOCK, you must explicitly call the routine below to synchronize the menu and the Toolbar:

```
APP_STANDARD.SYNCHRONIZE;
```

APP_STANDARD Package (See page 26 – 25)

Application–Specific Entries: The Special Menu

You can customize the menu to display application–specific values. The menu supports up to fifteen entries under the 'Special' entry.

APP_SPECIAL: Menu and Toolbar Control (See page 10 – 9)

Example Special Menu Entry

Suppose you have a special function called 'Book Order' that you want to add to the menu and the Toolbar. To add 'Book Order' as the first entry on the Special menu and as an icon on the Toolbar, such that they are only available in the 'Header' block of a form, do the following:

Step 1 Modify the form level PRE–FORM trigger:

```
PRE–FORM
app_special.instantiate('SPECIAL1', '&Book Order', 'bkord');
```

If you plan to translate your form, you should use Message Dictionary, a parameter, or a static record group cell to store the Special Menu entry. You then retrieve the value (which is translated when the application is translated) into a variable and pass the variable to the APP_SPECIAL routine. For example:

```
app_special.instantiate('SPECIAL1', my_menu_entry, 'bkord');
```

Step 2 Add a form–level PRE–BLOCK trigger:

```
PRE–BLOCK
app_special.enable('SPECIAL1',PROPERTY_OFF);
```

Step 3 Add a block level PRE–BLOCK trigger to the block in which you want to enable your special menu entries:

```
PRE–BLOCK in HEADER block
app_special.enable('SPECIAL1',PROPERTY_ON);
```

Step 4 Add a block level SPECIAL1 user–named trigger that contains code to actually perform your 'Book Order' function. It executes when the user chooses this menu entry.

Disabling the Special Menu

To disable all special menu entries (for example, when entering query–mode), call `APP_SPECIAL.ENABLE('SPECIAL', PROPERTY_OFF);`

APP_SPECIAL: Menu and Toolbar Control

Use the APP_SPECIAL package to enable and customize buttons on the toolbar.

Application–Specific Entries: The Special Menu
(See page 10 – 7)

APP_SPECIAL.INSTANTIATE

Summary

```
procedure  APP_SPECIAL.INSTANTIATE(
           option_name        varchar2,
           hint               varchar2 default null,
           icon               varchar2 default null,
           initially_enabled  boolean  default true,
           separator          varchar2 default null);
```

Description

This call constructs the special menu according to your specifications. Call this function in the PRE–FORM trigger, after the call to APP_STANDARD.EVENT('PRE–FORM').

Arguments (input)

option_name SPECIAL1 to SPECIAL15. Indicate the slot on the special menu in which you want to put your function. SPECIAL1 is at the top of the menu, SPECIAL15 at the bottom. When you instantiate any menu entry, the top level menu is enabled.

hint Your menu item label. Pass a translated string (if your form is to be translated, you should define a message in Message Dictionary, retrieve the message first, and pass the retrieved message string to APP_SPECIAL). Include an '&' in the string to define which character becomes the shortcut key for that item (this is the same as the behavior in the Oracle Forms Designer. For example, '&Book Orders').

icon If you include an iconic button on the toolbar for the function, give the name of the icon. This only applies for SPECIAL1, SPECIAL2,and SPECIAL3.

If there is no corresponding toolbar button, pass NULL.

Application–Specific Entries: The Special Menu (See page 10 – 7)

initially_enabled A boolean value that lets you set the initial status of the menu item. If you do not want to enable the item when your application starts, pass FALSE.

separator This parameter is not yet supported.

Example 1

```
APP_SPECIAL.INSTANTIATE('SPECIAL3','&Book Order', 'POBKORD',
TRUE);
```

APP_SPECIAL.ENABLE

Summary

```
procedure  APP_SPECIAL.ENABLE(
            option_name varchar2,
            state number);
```

Description

This call controls the enabling and disabling of the items in the menu, including the Special menu (and their corresponding toolbar buttons), allowing you to customize your menus for each block.

Menu and Toolbar Entries (See page 10 – 2)

If a special function is available for most of the blocks in a form, create a form level PRE–BLOCK trigger that ENABLEs the function. For any block where this is not a valid function, code a block level PRE–BLOCK trigger with Execution Style set to 'Override' that DISABLEs the function.

Enable and disable SAVE to control the 'Action–>Save' and 'Action–>Save and Enter Next' menu entries. Save is automatically disabled when you call APP_FORM.QUERY_ONLY MODE.

Before entering a modal window, call APP_SPECIAL. ENABLE('MODAL', PROPERTY_OFF). When you leave the block, call ENABLE again with PROPERTY_ON. PROPERTY_OFF disables the menu items that are disallowed in a modal block.

You can control the availability of the ATTACHMENTS, TRANSLATION, SUMMARY / DETAIL, and SELECT_ALL menu entries.

Use the SINGLE option to disable the first record, last record, previous record, and next record options on the Go menu in a block with only one available record.

Single Record Blocks (See page 5 – 13)

Use the ABOUT option to disable the Help–>About This Record menu option.

If the function is not available for most blocks in a form, reduce the total number of triggers you need to code by DISABLing it in the form level PRE–BLOCK and ENABLing it in the block level PRE–BLOCK.

If the function is available in every block in the form, ENABLE it in the PRE–FORM trigger.

Arguments (input)

	option_name	The name of the option to be enabled. Possible values ABOUT, ATTACHMENTS, MODAL, SAVE, SELECT_ALL, SINGLE, SPECIAL1, ...through SPECIAL15, SPECIAL, SUMMARY / DETAIL, TRANSLATION, or the full name of any menu item. Calling SPECIAL disables all Special Menu items.
	state	Either PROPERTY_ON or PROPERTY_OFF

Example 1

```
APP_SPECIAL.ENABLE('SPECIAL3',PROPERTY_ON);
```

CHAPTER

11

Menus and Function Security

This chapter provides you with information you need to implement Function Security and application menus in the Navigator. This chapter also provides you with Function Security APIs you can use in your client–side PL/SQL procedures.

The following topics are covered:

- Overview of Menus and Function Security
- Function Security Standards
- Function Security APIs for PL/SQL Procedures

Overview of Menus and Function Security

Function security lets you restrict application functionality to authorized users.

Application developers register functions when they develop forms. A System Administrator administers function security by creating responsibilities that include or exclude particular functions.

Basic Function Security

- Group the forms and functionality of an application into logical menu structures that will appear in the Navigator
- Assign a menu to one or more responsibilities
- Assign one or more responsibilities to one or more users

Advanced Function Security

- Oracle Applications GUI–based architecture aggregates several related business functions into a single form
- Not all users should have access to every business function in a form
- Oracle Applications provides the ability to identify pieces of application logic as *functions*
- Functions can be secured on a responsibility basis (that is, included or excluded from a responsibility)

Terms

Function

A function is a part of an application's functionality that is registered under a unique name for the purpose of assigning it to, or excluding it from, a responsibility.

There are two types of functions: form functions, and non–form functions. For clarity, we refer to a form function as a *form*, and a non–form function as a *subfunction*, even though both are just instances of functions in the database.

Form (Form Function)

A form function (*form*) invokes an Oracle Forms form. Form functions have the unique property that you may navigate to them using the Navigator window.

Subfunction (Non–Form Function)

A non–form function (*subfunction*) is a securable subset of a form's functionality: in other words, a function executed from within a form.

A developer can write a form to test the availability of a particular subfunction, and then take some action based on whether the subfunction is available in the current responsibility.

Subfunctions are frequently associated with buttons or other graphical elements on forms. For example, when a subfunction is enabled, the corresponding button is enabled.

However, a subfunction may be tested and executed at any time during a form's operation, and it need not have an explicit user interface impact. For example, if a subfunction corresponds to a form procedure not associated with a graphical element, its availability is not obvious to the form's user.

Menu

A menu is a hierarchical arrangement of functions and menus of functions that appears in the Navigator. Each responsibility has a menu assigned to it.

The Oracle Applications default menu appears as the pulldown menu across the top of a window and is not generally controlled using Function Security.

Pulldown Menus and the Toolbar (See page 10 – 2)

Menu Entry

A menu entry is a menu component that identifies a function or a menu of functions. In some cases, both a function and a menu of functions correspond to the same menu entry. For example, both a form and its menu of subfunctions can occupy the same menu entry.

Responsibility

A responsibility defines an application user's current privileges while working with Oracle Applications. When an application user signs on, they select a responsibility that grants certain privileges, specifically:

- The functions that the user may access. Functions are determined by the menu assigned to the responsibility.
- The concurrent programs, such as reports, that the user may run (request security group).
- The application database accounts that forms, concurrent programs, and reports connect to (data group).

Forms and Subfunctions

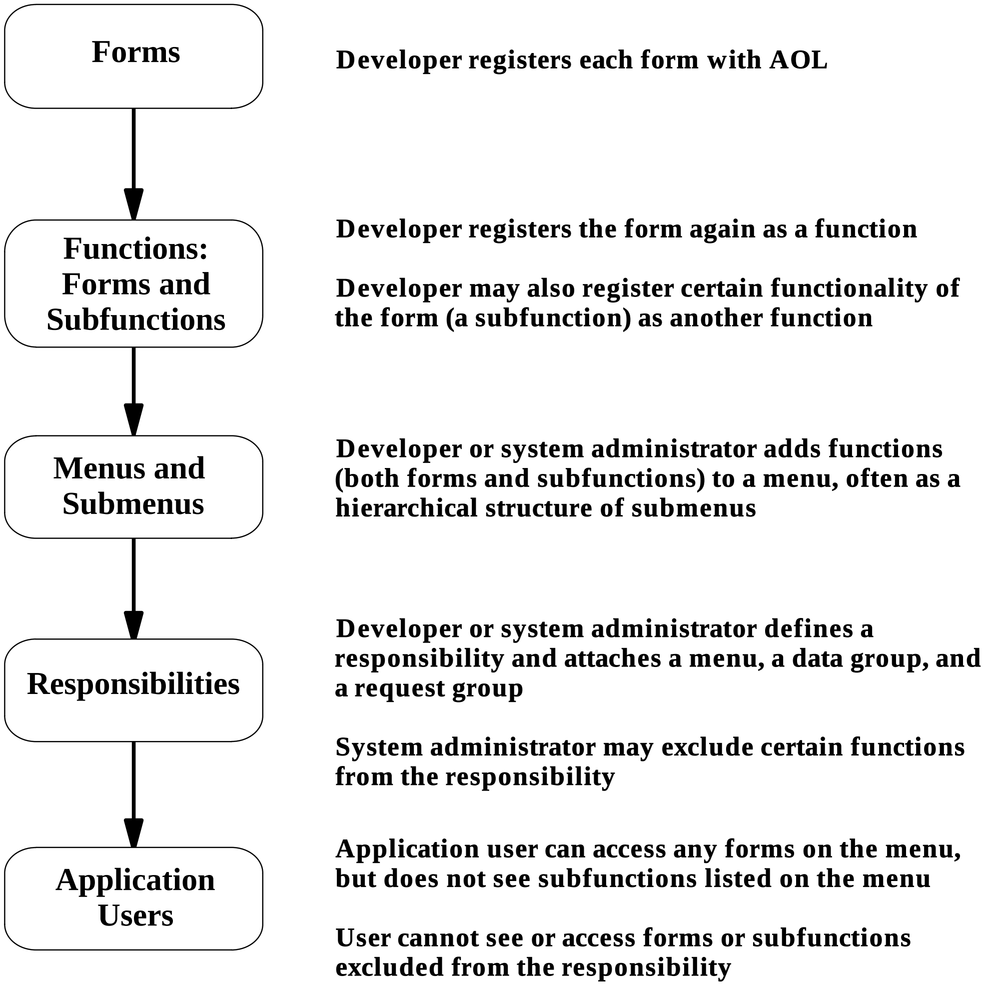
A form is a special class of function that differs from a subfunction in two ways:

- Forms appear in the Navigate window and can be navigated to. Subfunctions do not appear in the Navigate window and cannot be navigated to.
- Forms can exist on their own. Subfunctions can only be called by logic embodied within a form; they cannot exist on their own.

A form as a whole, including all of its program logic, is always designated as a function. Subsets of a form's program logic can optionally be designated as subfunctions if there is a need to secure those subsets.

For example, suppose that a form contains three windows. The entire form is designated as a function that can be secured (included or excluded from a responsibility.) Each of the form's three windows can be also be designated as functions (subfunctions), which means they can be individually secured. Thus, while different responsibilities may include this form, certain of the form's windows may not be accessible from each of those responsibilities, depending on how function security rules are applied.

How Function Security Works

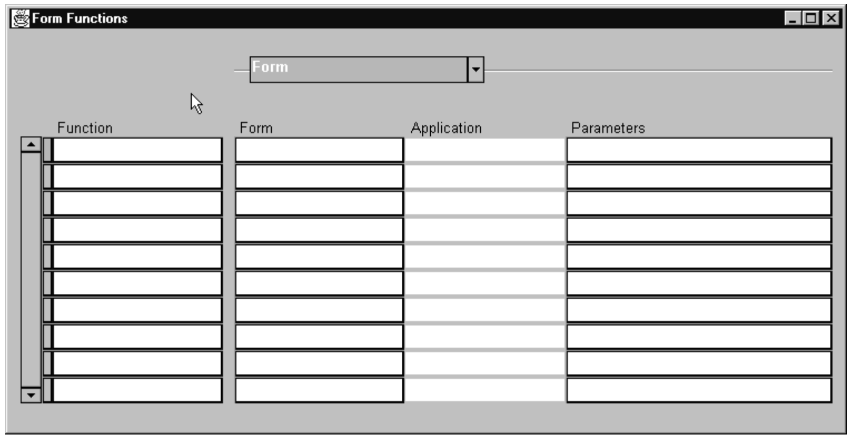


Developers Register Functions

Developers can require parts of their Oracle Forms code to look up a unique *function name*, and then take some action based on whether the function is available in the current responsibility.

Developers register functions. They can also register parameters that pass values to a function. For example, a form may support data entry only when a function parameter is passed to it.

Register your form functions and subfunctions on the Form Functions window.



Developers Create Menus

Typically, developers define a menu including all the functions available in an application (that is, all the forms and their securable subfunctions). For some applications, developers may define additional menus that restrict the application's functionality by omitting specific forms and subfunctions.

When developers define menus of functions, they typically group the subfunctions of a form on a subfunction menu they associate with the form.

When you create a menu, you typically include each form, each subfunction, and each submenu on a separate line of the menu. Generally, each form and each submenu should have a prompt so it will show up as a separate item in the Navigator window.

Attention: Usually you should *not* include a prompt for subfunctions, as you usually do not want them to be visible to the user on the Navigator. This also applies for form functions that you may open using the CUSTOM library and Zoom, but that you do not want the user to navigate to explicitly (that is, you should include the form function on the menu so it will be available to the responsibility, but you should not give it a prompt).

Coding Zoom (See page 25 – 6)

System Administrators Exclude Functions

Each Oracle Applications product is delivered with one or more predefined menu hierarchies. System Administrators can assign a predefined menu hierarchy to a responsibility. To tailor a responsibility, System Administrators exclude functions or menus of functions from that responsibility using exclusion rules.

When a menu is excluded, all of its menu entries, that is, all the functions and menus of functions that it selects, are excluded.

When you exclude a function from a responsibility, all occurrences of that function throughout the responsibility's menu structure are excluded.

Available Functions Depend on the Current Responsibility

When a user first selects or changes their responsibility, a list of functions obtained from the responsibility's menu structure is cached in memory.

Functions a System Administrator has excluded from the current responsibility are marked as unavailable.

Form functions in the function hierarchy (that is, the menu hierarchy) are displayed in the Navigate window. Available subfunctions are accessed by working with the application's forms.

Using Form Functions

To call a form from the Navigate window menu, you define a form function that consists of your form with any arguments you need to pass. You then define a menu that calls your form function.

You should use FND_FUNCTION.EXECUTE instead of OPEN_FORM whenever you need to open a form programatically. Using FND_FUNCTION.EXECUTE allows you to open forms without bypassing Oracle Applications security, and takes care of finding the correct directory path for the form.

Query–Only Forms

When you define a Form Function in the Function Security system (that is, for the Navigator menu), you can add the string:

```
QUERY_ONLY=YES
```

to the form's Arguments string to cause the form to be called in query–only mode. The FND_FUNCTION.EXECUTE procedure (which is also used by the Oracle Application Object Library Navigator) sets the QUERY_ONLY flag that sets all database blocks to non–insertable, non–updatable, and non–deletable.

To dynamically determine when to call a form in query–only mode, add the string to the other_params argument of the call to FND_FUNCTION.EXECUTE.

Disable or remove all functionality that does not apply when the form is run in Query–Only mode, such as 'New' buttons in Find Windows. Entries on the menu (other than Special) are handled automatically. Turn off any logic that defaults values into new records when the form is in Query–Only mode (this logic is usually called from the WHEN–CREATE–RECORD triggers). Check for this mode by checking the parameter `query_only`:

```
IF name_in('parameter.query_only') != 'YES' THEN
   <defaulting logic here>
END IF;
```

☞ **Attention:** Use query–only forms only when the user does not have update privileges on the form; not when the primary purpose of the form is viewing values.

Do not limit a form to query only because the primary need is viewing values. If the user has update privileges on a form, you should not create a separate query–only form function, even when calling the form from a special menu for the purpose of querying information. Forcing users to use the Navigator to reopen a form in an updatable mode if they need to make a change is a clear violation of our user interface standards.

There may be rare cases where technical limitations force you to limit the user to query mode on particular calls to a form. In general, however, update privileges should remain constant within a responsibility, regardless of how the user accesses the form (from a branch of the Navigator menu, or from a special menu in another form).

Form Window Name Changes

Some forms (such as the Submit Requests form) accept arguments that change the form window name. With the Submit Requests form, you use the parameter TITLE to specify the name of a message in the Message Dictionary. That message becomes the window title.

The syntax to use is:

```
TITLE=”<appl_short_name>:<message_name>”
```

If the message REP_ROUTING contained (in English) the text ”Report Routing”, you use the argument

```
TITLE=”MFG:REP_ROUTING”
```

to open the Submit Request window with the title Report Routing.

See the *Oracle Applications System Administrator’s Guide* for more information on customizing the Submit Requests form.

Help Target Changes

When a user selects the help button in Oracle Applications, the applications try to open the correct help file at a help target consisting of the form name and the window name: *form_name.window_name.* You can override the form name portion of this target by passing the parameter

HELP_TARGET=”*Alternate_Form_name”*

You can pass the HELP_TARGET parameter either when calling the form function using FND_FUNCTION.EXECUTE (using the other_params argument) or when you define the function in the Functions window.

See the *Oracle Applications System Administrator’s Guide* for more information on help targets in Oracle Applications.

Function Security Standards

The section contains the function security standards followed by Oracle Corporation developers.

General Function and Menu Standards

The Oracle Applications menu structure includes two types of items: functions, and menus of functions. Generally, functions are either forms (form functions) or subfunctions within those forms (non–form functions).

There may be some cases of functions that are neither forms nor subfunctions, but those cases should be rare, and thus are not addressed by these standards.

A ”full access” responsibility with a menu that includes all the functions in an application is predefined for each Oracle Applications product and shipped to customers. This menu includes one link to each of the product's forms.

Menus are Object–based

A standard Oracle Applications menu structure is object–based (as opposed to the type of action taken on an object). It has as many levels of categorical grouping as necessary until eventually getting to a menu entry for a single object, such as Purchase Orders. All Purchase Order forms are grouped together, including transaction forms, maintenance forms, inquiry forms, and any other form that works with Purchase Orders.

Menu Categories

At the top level of a menu, two general categories should always exist: Setup and Report. The setup forms are grouped separately, since they are primarily used at installation time, and after that would be ”in the way” if they were mixed with other forms. The report forms are grouped separately for users whose sole task is to run reports. Report forms are easy to secure using such a structure; moreover, reports frequently do not group purely by single object. Thus, all reports should be grouped under the Report top–level menu entry, not under other category areas or other branches.

Here is a simplified example of a product's top–level menu, with the Purchase Orders entry decomposed to a second menu level:

```
Purchase Orders
    Purchase Orders            (<–Purchase Orders Gateway)
    View Expiring Purchase Orders
    Mass Delete Purchase Orders
Quotes
Suppliers
Setup
Reports
```

Reports versus Processes

If you create separate instances of the Submit Requests form to launch specific processes (programs) or groups of processes, place that form function under the appropriate object–based name in your menu. (A process is a program that manipulates data, rather than a report that only sorts or displays the data.)

In the above example, the "Mass Delete Purchase Orders" menu entry might open a specialized Submit Request window to launch the Mass Delete Purchase Order standard request submission program. Since this process deletes data, it appears under the Purchase Order menu entry rather than the Reports menu entry.

Multi–row and Single–row Displays

When you have both a multi–row and a single row display of data in a form (usually in a combination block), title the multi–row window and associated menu entry using the plural of the entity name followed by "Summary", for example: "Purchase Orders Summary". Title the single–row window (and the associated menu entry, if there is one) with the plural of the entity name, for example: "Purchase Orders". If you have only a single–row version of a form, the form name and associated menu entry are simply the plural of the entity name, for example: "Purchase Orders".

Form Function Standards

Function Names and User Function Names

The user function name (which is defined using the Form Functions form, and which is the selection value in the LOV on the Menus form) is simply the form name, for example: "Purchase Orders". It is important to follow these user function naming standards, since end users see user function names in the Navigator window's Top Ten List.

Create function names (not user function names) as: <APPLICATION_SHORTNAME>_<FORMNAME>_<MODE>. <MODE> is optional; use it if there are several functions that reference the same form. If you create a function that references a form in another product, use your products shortname as the application shortname. For example, WIP_FNDRSRUN_AUTOCREATE.

Never begin a user function name with a number, such as "2–Tier Pricing Structure", since the number would conflict visually with the Top Ten List in the Navigator window. Menu entry prompts should not have numbers anywhere in them, because the numbers would conflict with the keyboard accelerators for the Top Ten List in the Navigator.

When the same form is used for multiple functions, differing only in the parameters passed to it, make the user function name the logical name for the function, for example, "View Purchase Orders". Internally, use a function like "PO_POXPOMPO_VIEW", for example, if you want to show this is the version of the form used only for viewing purchase orders. Do not use separator characters other than underscores.

Subfunction Standards

Hide Unavailable Functions

If a subfunction determines whether a button, field, menu choice or other form item is available, code the subfunction to hide the item if the user does not have access to that function. Hide anything not enabled while the user is in the form (as opposed to item that are enabled / disabled based on actions taken in the form).

Subfunction Menus

A form may have subfunctions within it whose availability to the user is determined by responsibility. To accomplish this, a menu of these subfunctions is placed in the menu hierarchy below the level of the form. A menu of subfunctions always assumes the name of the form entry with "_MENU" appended, for example: "PO_POXPOMPO_MENU". The user menu name should be the <form name>: Subfunctions, for example: "Purchase Orders: Subfunctions".

Subfunctions are tied directly to forms in the shipped menu to make it easier for the System Administrator to understand which subfunctions are part of which forms. In other words, there is one hierarchy combining the menu structure with the security structure, as opposed to separate menu and security structures following different hierarchies.

Subfunction Names

All subfunctions for each form are predefined by the developer, and are named <form>_<subfunction>, for example: ”PO_POXPOMPO_ DELETE”. The user function name should be <form name>: <subfunction>, for example: ”Purchase Orders: Delete”. This naming standard is important because it enables the System Administrator to find all the available functions to include or exclude from a responsibility by using Autoreduction in the LOV in the Responsibilities form. For example, the System Administrator can enter ”Purchase Orders”, and then see the Purchase Orders form itself, the subfunctions menu(s) for that form, and all the restrictable subfunctions. Without this naming standard, it would be difficult to find all the subfunctions of a form.

Grouping Subfunctions into Categories

Where there are many restrictable subfunctions for a particular form, and those subfunctions group well into categories (Approvals, for example), group the subfunctions according to their category, creating for example, ”PO_POXPOMPO_APPROVALS_MENU”, linking all the approval subfunctions below it. Grouping all Approval subfunctions into a single category allows the System Administrator to restrict access to all Approval subfunctions with one menu exclusion for that responsibility.

Grouping subfunctions by category should be done only when multiple subfunction categories exist, but not when all subfunctions of a form belong to a single category. The user names for these subfunction menus and the subfunctions under them follows the standard described above for subfunctions, for example: ”Purchase Orders: Approvals”, ”Purchase Orders: Approvals: Batch Approval”. Note that the word ”Menu” is not included in the subfunction menu names to help clarify that while subfunctions are stored like menus, they are not really menus in the user presentation. Instead, plurality indicates multiple subfunctions, as in ”Approvals” instead of ”Approval”.

Forms Appearing Multiple Times in a Menu

To add a form somewhere else in the menu, the System Administrator links the form into the additional location. There is no need to create a second copy of the subfunction menu since only one is applicable per form. However, the System Administrator is free to copy what existed elsewhere, linking both the subfunction menu and form onto the new location. (The results would be the same.) It is not possible to have the same form appear with access to different subfunctions in different places on the menu.

Some forms appear several times in a menu under different function names (for example, the QuickCodes form or the Submit Request form). Do not combine subfunctions for these forms into subfunction categories. Each subfunction should exist as a separate menu entry on the form's _menu rather than on a lower level subfunction menu.

For this special case, the standard ensures that System Administrators explicitly exclude by subfunction rather than by menu. Since the form window names may change, it is not always obvious that the form appears more than once in a menu. If System Administrators try to exclude a subfunction menu, they may not realize that the menu includes another copy of that subfunction menu under a separate occurrence of the form.

Including a subfunction anywhere in a menu permits the use of that subfunction wherever it is called in the menu. Excluding a subfunction for a responsibility restricts the use of that subfunction throughout the menu.

Function Security Reports

Use the function security reports to document the structure of your Navigator menus. You can use these reports as hardcopy to document your customized menu structures before upgrading your Oracle Applications software.

The function security reports consist of the Function Security Functions Report, the Function Security Menu Report, and the Function Security Navigator Report.

These reports are available through the Function Security Menu Reports request set in the System Administrator responsibility. For each report, specify the responsibility whose function security you want to review.

Function Security Function Report

Specify a responsibility when submitting the report. The report output lists the functions accessible by the specified responsibility.

The report does not include items excluded by function security rules.

Function Security Menu Report

Specify a responsibility when submitting the report. The report output lists the complete menu of the responsibility, including all submenus and functions.

The report indicates any excluded menu items with the rule that excluded it.

Function Security Navigator Report

Specify a responsibility when submitting the report. The report output lists the menu as it appears in the navigator for the responsibility specified.

This report does not include items excluded by function security rules, or non–form functions that do not appear in the Navigator.

Function Security APIs for PL/SQL Procedures

This section describes function security APIs you can use in your client–side PL / SQL procedures.

FND_FUNCTION.TEST and FND_FUNCTION_QUERY indicate whether a particular function is currently accessible. You can construct your code to test the availability of a particular function, and then take some action based on whether the function is available or not.

You can use FND_FUNCTION.EXECUTE to execute a particular form function.

FND_FUNCTION.TEST

Summary

```
function FND_FUNCTION.TEST

  (function_name IN varchar2) return boolean;
```

Description Tests whether a particular function is currently accessible. Typically you would test for a function's availability at form startup (for example, to prevent certain buttons from being displayed or certain windows from being accessible).

Arguments (input)

function_name The name of the function to test.

Example

```
IF (FND_FUNCTION.TEST('DEM_DEMXXEOR_PRINT_ORDER')) THEN
 /* Put Print Order on the Special menu */
    app_special.instantiate('SPECIAL1','&Print Order');
ELSE
 /* hide the corresponding button on the form
    (and the special menu is not instantiated) */
    app_item_property.set_property('orders.print_order',
                                   DISPLAYED, PROPERTY_OFF);
END IF;
```

FND_FUNCTION.QUERY

Summary

```
procedure FND_FUNCTION.QUERY

          (function_name   IN varchar2,
           accessible      OUT varchar2,
           function_type   OUT varchar2,
```

```
              form_path       OUT varchar2,
              arguments       OUT varchar2);
```

Description Checks whether a particular function is currently accessible, and if so, returns information about the function in function_type, form_path, and arguments. If the function is not accessible, function_type, form_path, and arguments are set to empty strings.

Arguments (input)

function_name	The name of the function to check.

Arguments (output)

accessible	Set to 'Y 'or 'N' to indicate whether the function can be accessed by the current responsibility.
function_type	The type of the function as specified in the Form Functions form.
form_path	The file system path to the form (or an empty string if there is no form associated with this function.)
arguments	The list of arguments specified for this function.

FND_FUNCTION.EXECUTE

Summary

```
procedure FND_FUNCTION.EXECUTE
             (function_name   IN varchar2,
              open_flag       IN varchar2,
              session_flag    IN varchar2  default NULL,
              other_params    IN varchar2  default NULL
              activate        IN varchar2  default 'ACTIVATE');
```

Description Executes the specified form function. Only executes functions that have a form attached. Displays a message to the end user if the function is not accessible.

Both the POST–RECORD and the POST–BLOCK triggers fire when you open another form. If this is a concern, set a flag before calling FND_FUNCTION.EXECUTE, and check this flag in your POST–RECORD and POST–BLOCK triggers. This may no longer be necessary in Oracle Forms version 4.5.7, but does no harm.

Make sure that the function is defined with Oracle Application Object Library. Also, the function must be somewhere on the menu for the responsibility, though the form does not need to be accessible from the menu in the Navigator (do this by adding the function to the menu but leaving the prompt blank). Otherwise, the user will get a message saying that function is not available.

You should use FND_FUNCTION.EXECUTE instead of OPEN_FORM whenever you need to open a form programatically. Using FND_FUNCTION.EXECUTE allows you to open forms without bypassing Oracle Applications security, and takes care of finding the correct directory path for the form.

Arguments (input)

function_name — The developer name of the form function to execute.

open_flag — 'Y' indicates that OPEN_FORM should be used; 'N' indicates that NEW_FORM should be used. You should always pass 'Y' for open_flag, which means to execute the function using the Oracle Forms OPEN_FORM built–in rather than the NEW_FORM built–in.

session_flag — Passing 'Y' for session_flag (the default) opens your form in a new database session, while 'N' opens the form in the same session as the existing form.

Opening a form in a new database session causes the form to have an independent commit cycle.

other_params — An additional parameter string that is appended to any parameters defined for the function in the Form Functions form. You can use other_params to set some parameters dynamically. It can take any number of parameters.

☞ **Attention:** If there are multiple additional parameters, the values passed to those parameters must have double quotes around them. For example, a form accepts two pieces of context information to perform a query when the form is accessed from a particular window. The concatenated string to pass should have the following syntax:

```
FND_FUNCTION.EXECUTE(
    FUNCTION_NAME=> function_name,
    OPEN_FLAG=>'Y', SESSION_FLAG=>'Y',
    OTHER_PARAMS=>
   'CONTEXT1="'||:block.context1 || ' "
    CONTEXT2=" || :block.context2 || ' " ');
```

activate_flag — Either ACTIVATE or NO_ACTIVATE (default is ACTIVATE). This flag determines whether the focus goes to the new form (ACTIVATE) or remains in the calling form (NO_ACTIVATE).

Example

```
FND_FUNCTION.EXECUTE(FUNCTION_NAME=>'DEM_DEMXXEOR',
    OPEN_FLAG=>'Y', SESSION_FLAG=>'Y',
    OTHER_PARAMS=>
   'ORDER_ID="'||param_to_pass1||
   '" CUSTOMER_NAME="'||param_to_pass2||'"');
```

FND_FUNCTION.USER_FUNCTION_NAME

Summary

```
function FND_FUNCTION.USER_FUNCTION_NAME
        (function_name   IN varchar2)
        return varchar2;
```

Description Returns the user function name.

Arguments (input)

function_name The developer name of the function.

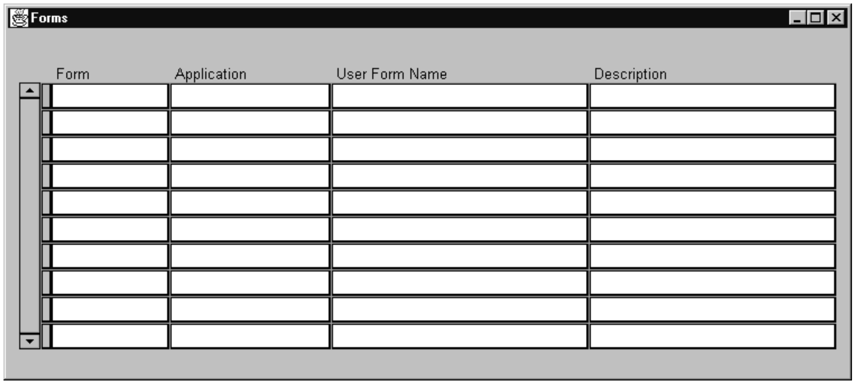
FND_FUNCTION.CURRENT_FORM_FUNCTION

Summary

```
function FND_FUNCTION.CURRENT_FORM_FUNCTION return varchar2;
```

Description Returns the function name with which the current form was called.

Forms Window



Register an application form with Oracle Applications.

You must register a form before you can call it from a menu or a responsibility.

Prerequisites

- Register your application with Oracle Application Object Library using the Applications window

Forms Block

The combination of application name and form name uniquely identifies your form.

Form

Enter the filename of your form (without an extension). Your form filename must be all uppercase, and its .fmx file must be located in your application directory structure.

Application

This is the application that owns your form. You can define an application by using the Applications window.

Applications: page 2 – 6

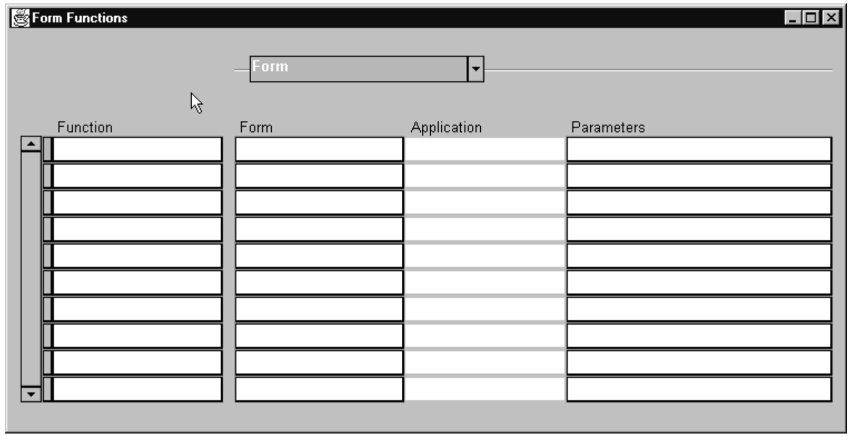
Oracle Applications looks for your form in the appropriate language directory of your forms directory, based on the application owning your form.

For example, if you are using American English on a Unix platform, Oracle Applications expects to find your form files in the directory /<Your application top directory>/forms/US.

User Form Name

This is the form name you see when selecting a form using the Functions window.

Form Functions Window



Define new functions. A function is a part of an application's functionality that is registered under a unique name for the purpose of assigning it to, or excluding it from, a responsibility.

There are two types of functions: form functions, and non–form functions.

For clarity, we refer to a form function as a *form*, and a non–form function as a *subfunction*, even though both are just instances of functions in the database.

Form Functions Block

Function

Users do not see this unique function name. However, you may use this name when calling your function programmatically. You should follow the naming conventions for functions.

User Function Name

Enter a unique name that describes your function. You see this name when assigning functions to menus. This name appears in the Top Ten List of the Navigator window.

Type

Type is a free-form description of the function's use. A function's type is passed back when a developer tests the availability of a function. The developer can write code that takes an action based on the function's type.

By convention, Oracle Applications form functions are registered with a type of FORM. A few, specialized functions that determine common form behaviors are registered with a type of UTIL.

Even if you do not register a form function with a type of FORM, Oracle Applications treats it as a form if you specify a valid Form Name/Application.

Form

Form /Application

If you are defining a form function, select the name and application of your form.

Parameters

Enter the parameters you wish to pass to your function. Separate parameters with a space.

For a form function, if you specify the parameter QUERY_ONLY=YES, the form opens in query–only mode. Oracle Application Object Library removes this parameter from the list of form parameters before opening the form in query–only mode.

You can also specify a different form name to use when searching for help for a form in the appropriate help file. The syntax to use is:

HELP_TARGET = "*alternative_form_name*"

Your form name overrides the name of the form.

Some Oracle Applications forms are coded to accept particular form parameters. For example, the Submit Requests form accepts a TITLE parameter you can use to change the Submit Requests window title. The syntax you should use is:

TITLE="*appl_short_name*:*message_name*"

where *appl_shortname*:*message_name* is the application short name and message name of a Message Dictionary message. See: Customizing the

Submit Requests Window using Codes in the *Oracle Applications System Administrator's Guide.*

> ⚠ **Warning:** In general, System Administrators should not modify parameters passed to functions that are predefined as part of the Oracle Applications products. The few cases where function parameters may be modified by a System Administrator are documented in the relevant technical reference manual or product update notes.

Web Regions

The fields in the Web regions are only required if your function will be accessed from Oracle Self–Service Web Applications. You do not need to enter any of these fields for SmartClient and Web–deployed Applications functions.

Host Name

The URL (universal resource locator) or address required for your function consists of three sections: the Host Name, Agent Name, and the HTML Call. The Host name is the IP address or alias of the machine where the Webserver is running.

Agent Name

The second section of your functions URL is the Oracle Web Agent. The Oracle Web Agent determines which database is used when running your function. Defaults to the last agent used.

HTML Call

The last section of your functions URL is the HTML Call. The HTML Call is used to activate your function. The function may be either a static web page or a procedure.

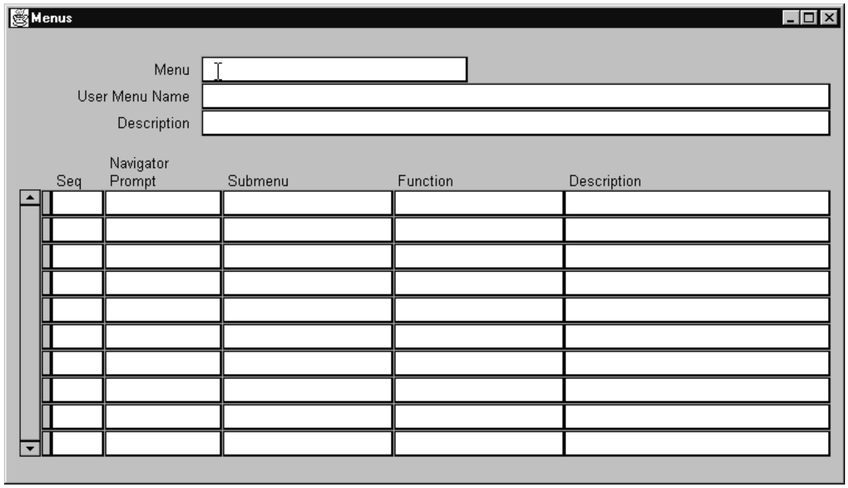
Secured

Secured is only required when your function is accessed by Oracle Workflow. Checking Secured enables recipients of a workflow E–Mail notification to respond using E–Mail.

Encrypt Parameters

Checking Encrypt Parameters adds a layer of security to your function to ensure that a user cannot access your function by altering the URL in their browser window. You must define Encrypt Parameters when you define your function to take advantage of this feature.

Menus Window



Define a new menu or modify an existing menu.

A menu is a hierarchical arrangement of functions and menus of functions. Each responsibility has a menu assigned to it.

A "full access" responsibility with a menu that includes all the functions in an application is predefined for each Oracle Applications product. As a System Administrator, you can restrict the functionality a responsibility provides by defining rules to exclude specific functions or menus of functions. In fact, we recommend that you use exclusion rules to customize a responsibility in preference to constructing a new menu hierarchy for that responsibility.

If you cannot create the responsibility you need by applying exclusion rules, you may build a custom menu for that responsibility using predefined forms (i.e., form functions) and their associated menus of subfunctions. However, we recommend that you do not disassociate a form from its developer–defined menus of subfunctions.

See:

Overview of Function Security
Implementing Function Security
Oracle Applications System Administrator's Guide

Prerequisites

- Register your application with Oracle Application Object Library using the Applications window.

- Define any menus that you intend to call from your menu. Define the lowest–level submenus first. A submenu must be defined before it can be called by another menu.

Suggestion: By calling submenus from your menu, you can group related windows together under a single heading on your menu. You can reuse your menu on other menus.

Menus Block

Menu entries detail the options available from your menu.

Menu

Choose a name that describes the purpose of the menu. Users do not see this menu name.

User Menu Name

You use the user menu name when a responsibility calls a menu or when one menu calls another.

Menu Entries Block

Sequence

Enter a sequence number to specify where a menu entry appears relative to other menu entries in a menu. The default value for this field is the next whole sequence number.

A menu entry with a lower sequence number appears before a menu entry with a higher sequence number.

Attention: If you change sequence numbers or frequently insert and delete menu entries, carefully check the default value. This value may be a duplicate sequence number or an out of sequence number.

Suggestion: You cannot replace a menu entry sequence number with another sequence number that already exists. If you want to add menu entries to a menu entry sequence that uses only integers, carefully renumber your menu entries to a sequence range well outside the sequence range you want, ensuring that you do not use existing sequence numbers

Once you save this work, you can go back and renumber each entry to have the final sequence number you want.

Navigator Prompt

Enter a user–friendly, intuitive prompt your menu displays for this menu entry. You see this menu prompt in the hierarchy list of the Navigator window.

Suggestion: Enter menu prompts that have unique first letters so that power users can type the first letter of the menu prompt to choose a menu entry.

Submenu

Call another menu and allow your user to select menu entries from that menu.

Function

Call a function you wish to include in the menu. A form function (form) appears in the Navigate window and allows access to that form. Other non–form functions (subfunctions) allow access to a particular subset of form functionality from this menu.

Description

Descriptions appear in a field at the top of the Navigate window when a menu entry is highlighted.

Function Security Reports: page 11 – 15

CHAPTER

12 Message Dictionary

This chapter provides you with information you need to implement Message Dictionary in your application.

- Overview of Message Dictionary
- Implementing Message Dictionary
- Message Dictionary APIs for PL/SQL Procedures
- Messages Window

Overview of Message Dictionary

Message Dictionary lets you catalog messages for display from your application without hardcoding them into your forms and programs. Using Message Dictionary, you can:

- Define standard messages you can use in all your applications
- Provide a consistent look and feel for messages within and across all your applications
- Define flexible messages that can include context–sensitive variable text
- Change or translate the text of your messages without regenerating or recompiling your application code

Major Features

Modifiable Message Text

Message Dictionary makes it easy for you to modify your messages. All your message text is available from one simple form, and you do not need to regenerate your forms or recompile your programs if you change your message text.

Message Dictionary displays your application messages in a format you choose. For example, you can display your messages in a dialog box or on the message line. You can also display messages without codes, such as warnings that have an intuitive remedy or do not need one.

Easy Translation

Message Dictionary facilitates translation of your messages by allowing you to easily modify your messages and by allowing you to define message notes for each message. Message Dictionary saves you time because you do not need to regenerate your forms or recompile your programs after translation of message text.

Standardized Messages

Message Dictionary lets you create standardized messages you can use in your application. Message Dictionary reduces redundant programming of standard messages by storing all of your messages as entries in Message Dictionary. Once you define your messages in the Message Dictionary, you can refer to them in your forms, concurrent programs, and other application modules using a simple message name you define. You can call the same message many times, from anywhere in your application. If you need to change your message, you only need to change it in one place.

Dynamic Message Text

Message Dictionary lets you include information in your message that Oracle Application Object Library derives at runtime. You can define your messages to accept variable text such as field values or module names. You specify the values of the variable message parts when you call Message Dictionary from a form or other application module. Message Dictionary inserts these values in the message before it returns the message to you. You can also include a field reference in your message call from a form, displaying the field's value in the message your user sees.

Definitions

Message Name

A non–updatable internal identifier for a message in your application. A message name can be up to 30 characters of text. A message name, together with your application name and language name, uniquely identifies your message text. You specify the message name when you call Message Dictionary from a form or program module.

Message

Text your application displays or prints to an output file. You can define your message to be up to about 1800 characters long (about 1260 in English to allow for translation into longer languages such as German).

Message Number

A number that appears with your message. If you define a non–zero message number for your message, Message Dictionary automatically prepends your message with the prefix APP– (or its translated equivalent).

Variable Token

A keyword you create to represent a value when you define a message. You specify the same variable token, along with its current value, when you call Message Dictionary from your form or program module. Message Dictionary replaces each variable token in your message with the current value you specify and then displays the message.

Implementing Message Dictionary

There are several steps to implementing Message Dictionary in your application:

1. Create your message directories: page 12 – 5
2. Define your messages: page 12 – 5
3. Create your message files: page 12 – 8
4. Code logic to set up messages: page 12 – 9
5. Code logic to display messages: page 12 – 11

Create Your Message Directories

On most operating systems, you should create a special subdirectory to hold your Message Dictionary files for your application. You must create your message directory (or some other location for your messages if your operating system does not support directories) before you define your messages so Oracle Application Object Library can store your message files. In general, name your subdirectory **mesg**, and create it directly under your application's base directory (exactly how you create a location for your Message Dictionary files depends on your operating system).

Building Your Application Framework
Message Dictionary
Oracle Applications Installation Manual

Define Your Messages

Use the Messages window to define your message information. You can include variable tokens in your message text when you define your messages. Message Dictionary inserts your values in the message automatically when it displays your message.

You can modify your messages at any time using the Messages window. If you want to change your message text, you need only change it once, instead of the many times your application may call it. You do not need to regenerate your forms or recompile your programs when you change your messages.

Messages Window (See page 12 – 27)

Message Naming Standards

The following suggested standards provide a consistent naming convention for message names.

- Message names should be all uppercase. Although message names are case insensitive, this provides consistency so that messages can be searched for easily.
- Message names should not have spaces; use underscore characters instead to break up words.
- Messages should belong to the application for which they are defined.
- Message names should not contain message numbers, as message numbers may change over time, but the message names are hardcoded into forms and programs and should never change.

Some examples of message names are:

App	Message Name
HR	SPEC_INFO_IN_USE
AR	MAND_PARAMETER_NULL
FND	ZOOM_BAD_FIELD
FND	AFDICT_HISTORY_BUTTON

Message Numbering Standards

In general, only use a number for errors that someone might need to get help with or look up in the messages manuals (for example: ”Contact your system administrator to help you...”). Do not use a non–zero number for ”obvious” error messages such as ”Please enter a value between 1 and 9”. Message Dictionary treats 0 and null as the same (and does not display the APP– or the message number).

Oracle Applications products all have assigned message number ranges. These ranges, and messages that have non–zero numbers appear in the *Oracle Applications Messages Manual*.

Message Ranges
Oracle Applications Messages Manual

For custom applications, the primary consideration for message numbering is to allow users to differentiate between Oracle Applications messages and custom messages. For this reason, we recommend that messages in custom applications use message numbers starting with 300,000 (the largest possible message number is about 2 billion).

Message Description Standards

Use the description field to detail the purpose of the message, where it might be used, and in what capacity. For example, if you define a message to use as a label for a Special menu entry, you should mention that in the description field for your message. The message description is especially important for short pieces of text that will be translated, so that translators have some idea of how the text would be used.

Message Token Standards

Using tokens to include values at runtime can make the message more flexible, but use them sparingly.

You should avoid hardcoding substitute text into your forms, since this would prevent you from updating or translating your message text later using the Messages window (and you would have to regenerate your form if you needed to change your message).

Use tokens only for strings that are unambiguous and unlikely to be translated, such as user names (from a profile option), file names, field values, or field names.

Avoid using tokens to substitute for words or phrases in a sentence, as these are nearly impossible to translate. For example:

```
Bad: This &ENTITY must be &ACTION.
Better: This purchase order must be approved.
```

This bad example is impossible to translate because the noun, ENTITY, could be either masculine or feminine, and singular or plural, all of which may affect translation of both the pronoun "This" and the verb "must" (for example, singular or plural, or masculine or feminine). You should use the TRANSLATE token with Message Dictionary routines for substitute text that may require translation.

You should make sure that substitutable text does not include phrases that depend on word order, as word order and sentence structure may change when translators translate a message.

Make token names clear and readable so that the message makes sense to a reader even before substitution takes place. For example:

```
Bad: &ENTITY1, &TOKEN, &NAME, &TYPE
```

```
Better: &PURCHASE_ORDER_TYPE, &USER_NAME
```

Always make tokens in your message uppercase so they are easy to identify in a messages manual or by a translator.

Message Content Standards

Messages should never exceed about 1260 characters in English if you expect the messages to be translated to ”longer” languages such as German.

For further standards on message content, see the *Oracle Applications User Interface Standards.*

Messages
Oracle Applications User Interface Standards

Create Your Message Files

Use the Generate Messages concurrent program to generate your message files. Note that in Release 10SC the client–side file (GUI) was different from the server–side file (character mode). In Release 11, both files are the same (GUI), and have the same filename. This procedure generates the GUI file.

To use the program to generate your message files:

Step 1 Using the Application Developer GUI responsibility, navigate to the Submit Requests window.

Step 2 Select the Generate Messages concurrent program in the Name field.

Step 3 In the Parameters window, select the language code for the language file you want to generate (for example, US for American English).

Step 4 Provide the appropriate application name for the message file you wish to create. Each application must have its own message file.

Step 5 Select the mode for the program. To generate your message file in this case, you would choose DB_TO_RUNTIME. The available options include:

DB_TO_ RUNTIME	Generate a runtime file (.msb) from the messages stored in the database.

DB_TO_SCRIPT	Generate a human–readable text file (.msg) from the messages stored in the database. This file is typically used by message writers and translators who need to do mass changes to the messages for an application.
RUNTIME_TO_DB	Load messages from a runtime file (.msb) into the database. This mode is not generally used because not all information is stored in the runtime file, and that information would be lost in the database after the loading.
RUNTIME_TO_SCRIPT	Generate a human–readable text file (.msg) from a runtime file (.msb). This mode is not generally used because not all information is stored in the runtime file, and that information would be lost in the resulting script file.
SCRIPT_TO_DB	Load (translated or rewritten) messages from the human–readable text file (.msg) into the database.
SCRIPT_TO_RUNTIME	Generate a runtime file (.msb) from the human–readable text file (.msg).

Step 6 Leave the Filename parameter blank in this case, as the message generator will create a file with a standard name (such as US.msb) in the mesg directory for your application on the server side (or an equivalent location for your platform). If you have chosen a mode that takes messages from a file, you would enter the name of the file.

Step 7 Make a copy of the resulting file (which is on the server side), and transfer the copy to the appropriate mesg directory for your application on the client machine. The file should have the same name (such as US.msb) in both locations.

Code Logic to Set Up Messages

Generating a message and showing it to a user is a two–step process: first you must set up the message (on the client side) or retrieve it from the server side, and then you must display it to the user (or write it to a file for a concurrent program). This section covers the setup part of the process.

When your application calls Message Dictionary, Message Dictionary finds the message associated with your application and the message name you specify, and replaces any variable tokens with your substitute text. If a concurrent process generates the message, depending on which routine it calls, Message Dictionary either writes the message to the concurrent program log or out file, or returns your message to your concurrent program so your program can write it to the log or out file. You can call Message Dictionary from any form or C concurrent program.

Client–side APIs for Retrieving and Setting up Messages

The following routines in the FND_MESSAGE package are used in client–side (that is, Oracle Forms) PL / SQL procedures to retrieve and set up messages for subsequent display.

SET_NAME Retrieves your message from Message Dictionary and sets it on the message stack.

SET_STRING Takes an input string and sets it on the message stack.

SET_TOKEN Substitutes a message token with a value you specify.

RETRIEVE Retrieves a message from the server–side message buffer, translates and substitutes tokens, and sets the message on the message stack.

GET (function) Retrieves a message from the message stack and returns a VARCHAR2.

CLEAR Clears the message stack.

Server–side APIs for Messaging

The following server–side routines are used to buffer a message (and if necessary, token / value pairs) so that a client–side PL / SQL Procedure (that is, one called from Oracle Forms) can retrieve and display it. Only one message can be buffered on the server.

SET_NAME Sets a message name in the global area without actually retrieving the message from Message Dictionary.

SET_TOKEN Adds a token / value pair to the global area without actually doing the substitution.

CLEAR Clears the message stack.

Code Logic to Display Messages

Once you have set up or retrieved the message and substituted any tokens, you can then display it to a user (on the client side) or write it to a file (on the server side for a concurrent program).

Client–side APIs for Displaying Messages

The following routines are used in client–side PL/SQL procedures to display messages. Each of these routines displays the message placed on the message stack by the most recent FND_MESSAGE.SET_NAME or FND_MESSAGE.RETRIEVE call in your program.

The FND_MESSAGE.ERROR, FND_MESSAGE.SHOW, FND_MESSAGE.WARN, and FND_MESSAGE.QUESTION routines each display a message in a forms modal window (on the client side). The primary difference between these routines is the icon they display next to the message in a forms modal window. For each routine, the icon is designed to convey a particular meaning. You should choose which routine to use based on the type of message you wish to display. For example, you should use the FND_MESSAGE.ERROR routine to display error messages, the FND_MESSAGE.SHOW routine to display informational messages, and so on.

Note that the look of the icons that the FND_MESSAGE.ERROR, FND_MESSAGE.SHOW, FND_MESSAGE.WARN, and FND_MESSAGE.QUESTION routines display is platform–dependent.

ERROR Displays an error message in a forms modal window or a concurrent program log file. (Example: “Invalid value entered.”)

SHOW Displays an informational message in a forms modal window or a concurrent program log file. (Example: “To complete this function, please enter the following... ”)

WARN Displays a warning message in a forms modal window and allows the user to either accept or cancel the current operation. (Example: “Do you wish to proceed with the current operation?”)

QUESTION Displays a message and up to three buttons in a forms modal window. (Example: “Please choose one of the following actions.”)

HINT Displays a message in the forms status line.

ERASE Clears the forms status line.

Methods for Server–side Messaging

Server–side PL/SQL currently has no I/O abilities by itself. Therefore, it relies on the environment that called the server–side routine to output the message.

There are three distinct, non–interchangeable methods for displaying messages that were set on the server:

Method 1: Set an error message on the server, to be displayed by the forms client that called the server procedure.

On the server, use FND_MESSAGE.SET_NAME and FND_MESSAGE.SET_TOKEN to set the message. Then call APP_EXCEPTION.RAISE_EXCEPTION (an APPCORE routine) to raise the application error PL/SQL exception. This exception is trapped when the server procedure is exited and control resumes on the client side in the standard Oracle Forms ON_ERROR trigger. The ON–ERROR trigger retrieves the message from the server and displays it.

Attention: All forms built to integrate with Oracle Applications should have a form–level ON–ERROR trigger that calls APP_STANDARD.EVENT('ON–ERROR'). APP_STANDARD.EVENT('ON–ERROR') in the ON–ERROR trigger automatically detects application errors raised on the server and retrieves and displays those error messages in a forms alert box.

Method 2: Set a message on the server, to be retrieved on the client side.

On the server, use FND_MESSAGE.SET_NAME and FND_MESSAGE.SET_TOKEN to set the message. Return a result code to the calling client code to indicate that a message is waiting. If there is a message waiting, the client calls FND_MESSAGE.RETRIEVE to pull the message from the server to the client, placing the message on the client's message stack. The client calls FND_MESSAGE.ERROR, FND_MESSAGE.SHOW, FND_MESSAGE.HINT, or FND_MESSAGE.WARN to display the message, or FND_MESSAGE.GET to retrieve the message to a buffer.

Method 3: Get a message into a buffer on the server

Use the FND_MESSAGE.SET_NAME, FND_MESSAGE.SET_TOKEN, and FND_MESSAGE.GET routines to get the message into a buffer. Or, use FND_MESSAGE.GET_STRING to get a single message into a string.

Calling Message Dictionary From Concurrent Programs

If you call message dictionary routines from your concurrent programs, the destinations of the messages are as follows:

Routine	Output Destination	Message Numbers	Messages Displayed
SHOW	out file	Not printed	One; top of stack
ERROR	log file	Printed if nonzero	One; top of stack

Table 12 – 1 (Page 1 of 1)

Message Dictionary APIs for PL/SQL Procedures

This section describes Message Dictionary APIs you can use in your PL/SQL procedures. This section also includes examples of PL/SQL procedure code using these Message Dictionary APIs.

Some of these PL/SQL procedures have C code analogs that you can use for concurrent programs written in C. The syntax for the C code API is included at the end of the PL/SQL API routine description. All of the Message Dictionary C routines require the use of the **fddutl.h** header file.

FND_MESSAGE.CLEAR

Summary

```
procedure FND_MESSAGE.CLEAR;
```

Location FNDSQF library and database (stored procedure)

Description Clears the message stack of all messages.

FND_MESSAGE.DEBUG

Summary

```
procedure FND_MESSAGE.DEBUG
         (value    IN varchar2);
```

Location FNDSQF library

Description Immediately show a string. This procedure is normally used to show debugging messages only, not messages seen by an end user. The string does not need to be defined in the Messages window. These strings may be hardcoded into the form and are not translated like messages defined in Message Dictionary.

value The string to display.

Example

```
/* as the last part of an item handler */
     ELSE
        fnd_message.debug('Invalid event passed to
            ORDER.ITEM_NAME: ' || EVENT);
     END IF;
```

FND_MESSAGE.ERASE

Summary

```
procedure FND_MESSAGE.ERASE;
```

Location FNDSQF library

Description Clears the Oracle Forms status line.

Suggestion: Due to the way that Oracle Forms handles I/O to the status line, changes made to the status line with HINT or ERASE may not appear immediately if the call is made in the middle of some PL/SQL routine that does not involve user input. In such cases it may be necessary to use the forms Synchronize built–in to force the message to get displayed on the status line.

FND_MESSAGE.ERROR

Summary

```
procedure FND_MESSAGE.ERROR;
```

Location FNDSQF library

Description Displays an error message in an Oracle Forms modal window or a concurrent program log file. (Example: "Invalid value entered.")

FND_MESSAGE.ERROR takes its message from the stack, displays the message, and then clears all the messages from the message stack.

Example

```
/* Display an error message with a translated token */
FND_MESSAGE.SET_NAME ('FND', 'FLEX_COMPILE_ERROR');
FND_MESSAGE.SET_TOKEN ('PROCEDURE', 'TRANS_PROC_NAME',
TRUE);
```

```
FND_MESSAGE.ERROR;
  /* Then either raise FORM_TRIGGER_FAILURE, or exit
     routine*/
```

C Code API

```
boolean afderror(/*_ void _*/);
```

Requires the **fddutl.h** header file.

FND_MESSAGE.GET

Summary

```
function FND_MESSAGE.GET
       return varchar2;
```

Location FNDSQF library and database (stored function)

Description Retrieves a translated and token–substituted message from the message stack and then clears that message from the message stack. This could be used for getting a translated message for a forms built–in or other function. Assumes you have already called FND_MESSAGE.SET_NAME and, if necessary, FND_MESSAGE.SET_TOKEN. GET returns up to 2000 bytes of message.

If this function is called from a stored procedure on the database server side, the message is retrieved from the Message Dictionary table. If the function is called from a form or forms library, the message is retrieved from the messages file on the forms server.

If you are trying to get a message from a stored procedure on the database server side to a form, you should use FND_MESSAGE.SET_NAME and, if necessary, FND_MESSAGE.SET_TOKEN in the stored procedure. The form should use Method 1 or Method 2 (See page 12 – 12) to obtain the message from the stored procedure. You should not use FND_MESSAGE.GET in the stored procedure for this case.

Example

```
/* Get translated string from message file */
declare
   msg varchar2(2000);
begin
   FND_MESSAGE.SET_NAME ('FND', 'A_2000_BYTE_MSG');
   msg := FND_MESSAGE.GET;
end;
  /*  We now have a translated value in the msg variable
      for forms built-in or other function */
```

C Code API

Pass this function a buffer and tell it the size, up to 2001 bytes (including the null terminator), of the buffer in bytes.

```
boolean afdget(/*_text *msg_buf, size_t buf_size _*/);
```

Requires the **fddutl.h** header file.

FND_MESSAGE.HINT

Summary

```
procedure FND_MESSAGE.HINT;
```

Location

FNDSQF library

Description

Displays a message in the Oracle Forms status line. FND_MESSAGE.HINT takes its message from the stack, displays the message, and then clears that message from the message stack.

The user may still need to acknowledge the message if another message immediately comes onto the message line.

FND_MESSAGE.QUESTION

Summary

```
function FND_MESSAGE.QUESTION
         (button1         IN varchar2 default 'YES',
          button2         IN varchar2 default 'NO',
          button3         IN varchar2 default 'CANCEL',
          default_btn     IN number   default  1,
          cancel_btn      IN number   default  3,
```

```
        icon          IN varchar2 default 'question'
) return number;
```

Location FNDSQF library

Description Displays a message and up to three buttons in an Oracle Forms modal window. (Example: “Please choose one of the following actions: ”)

FND_MESSAGE.QUESTION takes its message from the stack, and clears *that* message. After the user selects a button, FND_MESSAGE.QUESTION returns the number of the button selected.

For each button, you must define or use an existing message in Message Dictionary (under the Oracle Application Object Library application) that contains the text of the button. This routine looks for your button name message in the Oracle Application Object Library messages, so when you define your message, you must associate it with Oracle Application Object Library (the “FND” application) instead of your application.

Arguments (input)

button1 Specify the message name that identifies the text for your rightmost button. This name is identical to the message name you use when you define your button text using the Messages form.

button2 Specify the message name that identifies the text for your middle button. This name is identical to the message name you use when you define your button text using the Messages form.

button3 Specify the message name that identifies the text for your leftmost button. This name is identical to the message name you use when you define your button text using the Messages form.

Attention: Specifying no buttons produces a “Cancel / No / Yes” three–button display. Specifying one button displays that button as the first button, and defaults the second button to “Cancel”. Button one appears on the lower right of the window, button2 to the left of button1, and button3 to the left of button2.

```
button3           button2              button1
```

To specify two buttons without a cancel button, pass in arguments of '<FIRST_OPTION>', '<SECOND_OPTION>', and NULL.

default_btn Specify the number of the button that will be pressed when the user presses the "default" keyboard accelerator (usually the return or enter key). Passing NULL makes button 1 be the default.

cancel_btn Specify the number of the button that will be pressed when the user presses the "cancel" keyboard accelerator (usually the escape key). Passing NULL makes no buttons get pressed by the "cancel" keyboard button.

icon Specify the icon to display in the decision point box. If you do not specify an icon, a standard FND_MESSAGE.QUESTION icon is displayed.

Standard icons you can use include STOP, CAUTION, QUESTION, NOTE, and FILE. In addition, you can use any icon in the AU_TOP/resource directory on your platform.

Example 1

```
/* Display a message with two buttons in a modal window */
FND_MESSAGE.SET_NAME('INV', 'MY_PRINT_MESSAGE');
FND_MESSAGE.SET_TOKEN('PRINTER', 'hqunx138');
FND_MESSAGE.QUESTION('PRINT-BUTTON');
  /* If 'PRINT-BUTTON' is defined with the value "Print",
     the user sees two buttons: "Print", and "Cancel".  */
```

Example 2

```
/* Display a message with three buttons in a modal window.
   Use the Caution icon for the window. */

FND_MESSAGE.SET_NAME('FND', 'DELETE_EVERYTHING');
FND_MESSAGE.QUESTION('DELETE', NULL, 'CANCEL', 1, 3,
'caution');
```

Example 3

```
/* Display a message with two buttons in a modal window.
   "Yes" and "No" */

FND_MESSAGE.SET_NAME('FND', 'REALLY');
FND_MESSAGE.QUESTION('YES', 'NO', NULL);
```

FND_MESSAGE.RETRIEVE

Summary

```
procedure FND_MESSAGE.RETRIEVE;
```

Location

FNDSQF library

Description

Retrieves a message from the database server, translates and substitutes tokens, and sets the message on the message stack.

Example

```
/* Retrieve an expected message from the server side,
   then show it to the user */
FND_MESSAGE.RETRIEVE;
FND_MESSAGE.ERROR;
  /* Then either raise FORM_TRIGGER_FAILURE, or exit
     routine*/
```

FND_MESSAGE.SET_NAME

Summary

```
procedure FND_MESSAGE.SET_NAME
            (application    IN varchar2,
             name           IN varchar2);
```

Location

FNDSQF library and database (stored procedure)

Description (Forms)

Retrieves your message from Message Dictionary and sets it on the message stack. You call FND_MESSAGE.SET_NAME once for each message you use in your client–side PL/SQL procedure. You must call FND_MESSAGE.SET_NAME before you call FND_MESSAGE.SET_TOKEN.

Description (Database Server)

Sets a message name in the global area without actually retrieving the message from Message Dictionary.

Arguments (input)

application — The short name of the application this message is associated with. This short name references the application you associate with your message when you define it using the Messages form.

name — The message name that identifies your message. This name is identical to the name you use when you define your message using the Messages form. Message Dictionary names are not case sensitive (for example, MESSAGE_NAME is the same name as message_name).

Example 1

```
/* Display a warning, asking OK/Cancel question */
FND_MESSAGE.SET_NAME ('FND', 'WANT_TO_CONTINUE');
FND_MESSAGE.SET_TOKEN ('PROCEDURE', 'Compiling this
flexfield');
if (FND_MESSAGE.WARN) then
  /* User selected OK, so add appropriate logic ... */
```

Example 2

```
/* Display a warning, asking OK/Cancel question */
FND_MESSAGE.SET_NAME ('FND', 'WANT_TO_CONTINUE');
FND_MESSAGE.SET_TOKEN ('PROCEDURE', translated_text_vbl);
if (FND_MESSAGE.WARN) then
  /* User selected OK, so add appropriate logic ... */
```

Example 3

```
/* Show an informational message */
FND_MESSAGE.SET_NAME ('FND', 'COMPILE_CANCELLED');
FND_MESSAGE.SHOW;
```

Example 4

```
/* This code is on the server.  It sets up a message and
   then raises an error so the client will retrieve the
   message and display it to the user */
FND_MESSAGE.SET_NAME ('FND', 'FLEX_COMPILE_ERROR');
FND_MESSAGE.SET_TOKEN ('PROCEDURE', 'My Procedure');
APP_EXCEPTION.RAISE_EXCEPTION;
```

C Code API

```
boolean afdname(/*_ text *applsname, text *msg_name _*/);
```

Requires the **fddutl.h** header file.

FND_MESSAGE.SET_STRING

Summary

```
procedure FND_MESSAGE.SET_STRING
          (value    IN varchar2);
```

Location FNDSQF library

Description Takes an input string and sets it directly on the message stack. The string does not need to be defined in the Messages window. These strings may be hardcoded into the form and are not translated like messages defined in Message Dictionary.

Arguments (input)

value Indicate the text you wish to place on the message stack.

Example 1

```
/* Set up a specific string (from a variable) and show it */
FND_MESSAGE.SET_STRING (sql_error_message);
FND_MESSAGE.ERROR;
```

Example 2

```
/* Set up a specific string and show it */
FND_MESSAGE.SET_STRING ('Hello World');
FND_MESSAGE.SHOW;
```

FND_MESSAGE.SET_TOKEN

Summary

```
procedure FND_MESSAGE.SET_TOKEN
          (token          IN varchar2,
           value          IN varchar2,
           translate      IN boolean default FALSE);
```

Location FNDSQF library and database (stored function)

Description (Forms) Substitutes a message token with a value you specify. You call FND_MESSAGE.SET_TOKEN once for each token/value pair in a message. The optional translate parameter can be set to TRUE to indicate that the value should be translated before substitution. (The value should be translated if it is, itself, a Message Dictionary message name.)

Description (Database Server)

Same behavior as for client–side FND_MESSAGE.SET_TOKEN, except that adds a token / value pair to the global area without actually doing the substitution.

Arguments (input)

token — Specify the name of the token you want to substitute. This token name is identical to the token name you use when you define your message using the Messages form. Though you specify **&** before each of your variable tokens when you define messages, you should not include the **&** in your Message Dictionary calls.

value — Indicate your substitute text. You can include as much substitute text as necessary for the message you call.

You can specify a message name instead of actual substitute text. You must also specify TRUE for the translate argument in this case. If you are passing a Message Dictionary message this way, Message Dictionary looks for the message under the application specified in the preceding call to FND_MESSAGE.SET_NAME.

translate — Indicates whether the value is itself a Message Dictionary message. If TRUE, the value is ”translated” before substitution; that is, the value is replaced with the actual Message Dictionary message text. Note that if the ”token message” in turn contains a token, that token will not be substituted (that is, you cannot have ”cascading substitution”).

Example 1

```
/* Display a warning, asking OK/Cancel question */
FND_MESSAGE.SET_NAME ('FND', 'WANT_TO_CONTINUE');
FND_MESSAGE.SET_TOKEN ('PROCEDURE', 'Compiling this
flexfield');
if (FND_MESSAGE.WARN) then
  /* User selected OK ... */
```

Example 2

```
/* Display a warning, asking OK/Cancel question */
FND_MESSAGE.SET_NAME ('FND', 'WANT_TO_CONTINUE');
FND_MESSAGE.SET_TOKEN ('PROCEDURE', translated_text_vbl);
if (FND_MESSAGE.WARN) then
  /* User selected OK ... */
```

Example 3

```
/* Display an error message with a translated token */
FND_MESSAGE.SET_NAME ('FND', 'FLEX_COMPILE_ERROR');
FND_MESSAGE.SET_TOKEN ('PROCEDURE', 'TRANS_PROC_NAME',
TRUE);
FND_MESSAGE.ERROR;
  /* Then either raise FORM_TRIGGER_FAILURE, or exit
     routine*/
```

C Code API

The C code equivalent to SET_TOKEN(token_name, token_value, FALSE) is:

```
boolean afdtoken(/*_ text *token_name,
                  text *token_value _*/);
```

The C code equivalent to SET_TOKEN(token_name, token_value, TRUE) is:

```
boolean afdtrans(/*_ text *token_name,
                 text *token_value _*/);
```

The C code equivalent to SET_TOKEN(token_name, token_value, FALSE for number val* is:

```
boolean afdtkint(/*_ text *token_name, sb4 token_value _*/);
```

Requires the **fddutl.h** header file.

FND_MESSAGE.SHOW

Summary

```
procedure FND_MESSAGE.SHOW;
```

Location

FNDSQF library

Description

Displays an informational message in an Oracle Forms modal window or a concurrent program log file. (Example: “To complete this function, please enter the following... ”)

FND_MESSAGE.SHOW takes its message from the stack, displays the message, and then clears only *that* message from the message stack.

Example

```
/* Show an informational message */
FND_MESSAGE.SET_NAME ('FND', 'COMPILE_CANCELLED');
FND_MESSAGE.SHOW;
```

C Code API

```
boolean afdshow(/*_ void _*/);
```

Requires the **fddutl.h** header file.

FND_MESSAGE.WARN

Summary

```
function FND_MESSAGE.WARN
         return boolean;
```

Location

FNDSQF library

Description

Displays a warning message in an Oracle Forms modal window and allows the user to either accept or cancel the current operation. (Example: “Do you wish to proceed with the current operation?”)

FND_MESSAGE.WARN returns TRUE if the user accepts the message (that is, clicks OK), or FALSE if the user cancels.

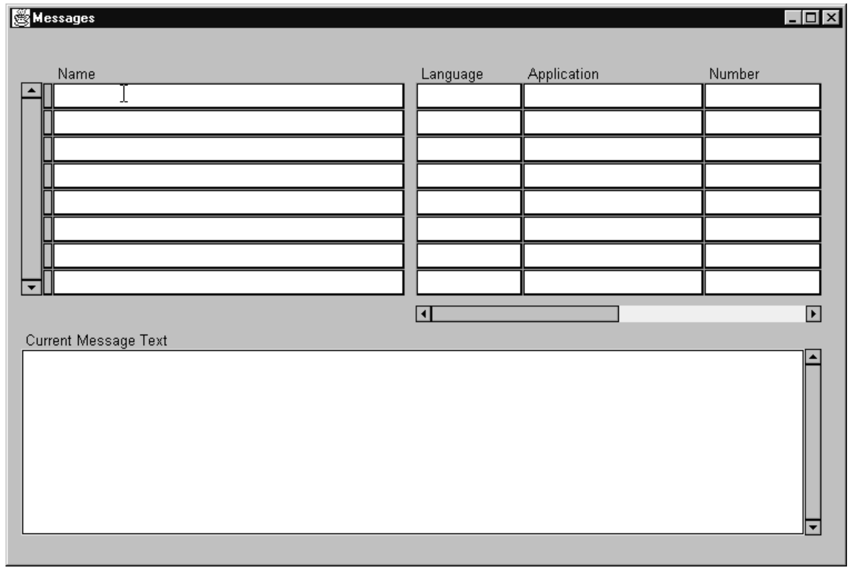
FND_MESSAGE.WARN takes its message from the stack, displays the message, and clears *that* message from the message stack.

Example

```
/* Display a warning, asking OK/Cancel question */
FND_MESSAGE.SET_NAME ('FND', 'WANT TO CONTINUE');
FND_MESSAGE.SET_TOKEN ('PROCEDURE', 'Compiling this
flexfield');
```

```
IF (FND_MESSAGE.WARN) THEN
  /* User selected OK, so add appropriate logic ... */
ELSE
  /* User selected Cancel, so add appropriate logic ... */
END IF;
```

Messages Window



Define your application messages before your routines can call them from a form and before your users can request detailed messages from a form. You should define your messages according to the message standards.

Once you leave the Messages window, after you make and save your changes, you should submit a concurrent request for the Generate Messages program to build your message file. Your new messages take effect as soon as your concurrent request finishes successfully and you have placed the new file in the appropriate directories.

Create Your Message Files: page 12 – 8

When you upgrade, any customizations you make to Oracle Applications messages will be overwritten. However, an upgrade does not overwrite messages you define using your own application.

Prerequisites

- Register your application.
- Create a **mesg** directory (or some other location if your operating system does not support directories) directly under your application's base directory where Oracle Application Object

Library can store your message files. You need a **mesg** directory on both the client machine and the server machine.

Messages Block

Application name and message name uniquely identify your message.

Name

Your message name can be any combination of letters, numbers, hyphens (–), underscores (_) and spaces, up to 30 characters in length. Message Dictionary names are not case sensitive (for example, MESSAGENAME is the same name as messagename).

You use this message name in your forms or concurrent programs when you invoke the Message Dictionary.

Language

Oracle Applications displays the correct language based on the user's current language.

Application

Enter the name of the application for which you wish to define message text.

When you upgrade, any customizations to Oracle Applications messages will be overwritten. However, an upgrade does not overwrite messages you define using your own application name.

Number

Enter a message number, if appropriate. If you define a non–zero message number for your message, Message Dictionary automatically prepends your message with the prefix APP– (or its translated equivalent). Message Dictionary treats 0 and null as the same (and does not display the APP– or the message number).

Description

You should enter information in this field that would help explain the context of this message to translators.

Current Message Text

Enter a message that describes the problem and its resolution. You can include variable tokens preceded by an ampersand (&) to indicate the location of substitute text. You supply the substitute text or field references in your form's message calls. For example, you could define an explanation for a message you call "Value Less Than Or Equal" like this:

```
Please enter a value that is less than or equal to &VALUE.
```

Your user sees the message explanation as:

```
Please enter a value that is less than or equal to $30.00.
```

You can specify your own variable tokens using numbers, letters, and underscores (_). Your variable tokens can be up to 30 characters long. You can use the same token more than once in your defined messages to repeat your substitute text.

CHAPTER

13

User Profiles

This chapter provides you with everything you need to know about Oracle Application Object Library user profiles. It includes an overview of using user profiles in your application, instructions on how to implement them, and detailed descriptions of the Oracle Application Object Library forms you use to implement user profiles.

The following topics are covered:

- Overview of User Profiles
- Implementing User Profiles
- FND_PROFILE: User Profile APIs
- User Profile C Functions
- Profiles Window

Overview of User Profiles

A user profile is a set of changeable options that affects the way your applications run. Oracle Application Object Library establishes a value for each option in a user's profile when the user logs on or changes responsibility. Your user can change the value of profile options at any time. Oracle Application Object Library provides many options that your users can set to alter the user interface of your applications to satisfy their individual preferences. For a complete list of all predefined user profile options, see the appendix in the *Oracle Applications System Administrator's Guide.*

Profile Options Appendix
Oracle Applications System Administrator's Guide

You, as a developer, can define even more profile options that affect your Oracle Application Object Library–based applications. And, because you may not want your users to be able to override values for each of your options, you can define options at one or more of four levels: Site, Application, Responsibility and User. You can choose at which of these levels you want your system administrator to be able to see and update option values. You also decide which profile options defined at the User level you want your users to be able to see and update.

For example, you can define a user profile option to determine which subset of the organization's data your end user sees. From the point of view of a system administrator or end user, user profile options you define are indistinguishable from those Oracle Application Object Library provides.

Definitions

User Profile Levels

User profile options exist at Site, Application, Responsibility and User levels. Oracle Application Object Library treats user profile levels as a hierarchy, where User is the highest level of the hierarchy, followed by Responsibility, Application and at the lowest level, Site. Higher–level option values override lower–level option values.

Each user profile option ordinarily exists at each level. For example, Oracle Application Object Library provides a site–level Printer option, an application–level Printer option, and so on. Oracle Application

Object Library enforces the level hierarchy to ensure that higher–level option values override lower–level values. As a result, if your Site–level Printer value is ”New York”, but your User–level Printer value is ”Boston”, your reports print on the Boston printer.

Site Level

Site is the lowest user profile level. Site–level option values affect the way all applications run at a given installation.

Application Level

Application is the user profile level immediately above Site. Application–level option values affect the way a particular application runs.

Responsibility Level

Responsibility is the user profile level immediately above Application. Responsibility–level option values affect the way applications run for all users of a responsibility.

User Level

User is the highest user profile level and is immediately above Responsibility. User–level option values affect the way applications run for an application user.

Defining New User Profile Options

When you develop a new application, you can define user profile options that affect the way your application runs. For example, you might define a user profile option to determine which subset of the organization's data your end user sees. When you define a new option, you specify criteria to describe valid values for that option. You can also specify whether your end users can change the value of an option.

Profiles Window (See page 13 – 13)

You can obtain the value of a user profile option using Oracle Application Object Library routines. You can construct your application to react to the value of a profile option as you choose.

Implementing User Profiles (See page 13 – 5)

Setting User Profile Option Values

A system administrator can set values for user profile options at each profile level. You can set default values for your new profile options by using the System Administrator responsibility. Typically, a system administrator sets site–level option values after installing Oracle Application Object Library–based applications at a site. These site–level option values then work as the defaults until the system administrator or end user sets them at higher levels.

Oracle Application Object Library derives run–time values for each user's profile options based on values the system administrator sets at each level. An option's run–time value is the highest–level setting for that option. For example, for a given end user, assume the Printer option is set only at the Site and Responsibility levels. When the end user logs on, the Printer option assumes the value set at the Responsibility level, since it is the highest–level setting for the option.

If the default value of a user profile option at any level is inappropriate, the system administrator can change it at any time. This change takes effect as soon as end users log on again or change responsibilities.

Update System Profile Options
Oracle Applications System Administrator's Guide

Setting Your Personal User Profile

Oracle Application Object Library establishes values for user profile options when you log on or change responsibilities. You can change the value of your own user–changeable options at any time.

You change an option value using the Update Personal Profile Options form. Using this form, you can display all your options and review the values your system administrator has set for them. You can also change those that are updatable if you like. Any change you make to a User–level profile option has an immediate effect on the way your applications run for that session. And, when you log on again, changes you made to your User–level options in a previous session are still in force.

If you never set your own User–level option values, your user profile options assume the Site–, Application–, Responsibility–, or User–level values your system administrator has set for them. Only the system administrator can set some profile options.

Implementing User Profiles

You should define user profile options whenever you want your application to react in different ways for different users, depending on specific user attributes.

To provide maximum flexibility, user profiles exist at Site, Application, Responsibility and User levels. When you define a profile option you decide whether your system administrator can set values for your option at each of these levels. You also decide whether your end users can view and update options you define at the User level. For example, you could define a VIEW_SECURE_INFORMATION option to be visible and updatable at all levels, so a system administrator could set values at any level, including values for individual users. You would also define the option such that your end users could not see or change its value.

Oracle Application Object Library provides many options that your users can set according to their needs. You can use these options, and profile options you define, in your application forms and concurrent programs.

Profiles Window (See page 13 – 13)

Predefined User Profile Options

Database Profile Options

Oracle Application Object Library provides many user profile options that the Oracle System Administrator or the users can see and update. The Oracle System Administration Reference Manual contains a complete list of predefined profile options.

Additional Information: Profile Options Appendix
Oracle Applications System Administrator's Guide

Internally Generated Profile Options

Oracle Application Object Library also provides a set of profile options that you can access via the user profile routines. You can retrieve values for these profile options in your forms and programs; however, except for the profiles CONC_PRINT_OUTPUT and CONC_PRINT_STYLE, you cannot change their values. System administrators and end users cannot see the values for, nor change the values of, these profile options.

USERNAME Your user’s current Oracle Application Object Library username.

USER_ID Your user’s current Oracle Application Object Library user ID.

RESP_ID Your user’s current responsibility ID.

APPL_SHRT_ NAME The short name of the application connected to your user’s current responsibility.

RESP_APPL_ID The application ID of the application connected to your user’s current responsibility.

FORM_NAME The name of the current form. Not available for concurrent programs.

FORM_ID The form ID of the current form. Not available for concurrent programs.

FORM_APPL_ NAME The name of the application for which the current form is registered. Not available for concurrent programs.

FORM_APPL_ID The application ID of the application for which the current form is registered. Not available for concurrent programs.

LOGON_DATE Your user’s logon date for the current session.

LAST_LOGON_ DATE Your user’s logon date for the previous session.

LOGIN_ID Your user’s Sign–On Audit login ID in Oracle Application Object Library.

CONC_ REQUEST_ID The request ID associated with a particular instance of your running current program. You can only use this profile option in a concurrent program. You use this profile option to fill the REQUEST_ID Who column.

Implementing Concurrent Processing (See page 15 – 21)
Tracking Data Changes with WHO (See page 3 – 2)

CONC_ PROGRAM_ID The program ID associated with a running current program. You can only use this profile option in a concurrent program. You use this profile option to fill the PROGRAM_ID Who column.

Implementing Concurrent Processing (See page 15 – 21)
Tracking Data Changes with WHO (See page 3 – 2)

CONC_ PROGRAM_ APPLICATION_ ID

The application ID associated with a running current program. You can only use this profile option in a concurrent program. You use this profile option to fill the PROGRAM_APPLICATION_ID Who column.

Implementing Concurrent Processing (See page 15 – 21)
Tracking Data Changes with WHO (See page 3 – 2)

CONC_LOGIN_ ID

The login ID associated with a running concurrent program. You can only use this profile option in a concurrent program. You can use this profile option to fill the LAST_UPDATE_LOGIN Who column.

Implementing Concurrent Processing (See page 15 – 21)
Tracking Data Changes with WHO (See page 3 – 2)

CONC_PRINT_ OUTPUT

The value Yes or No that you enter in the Print Output field when you register a concurrent program. You can use the routine **afpoput()** from your concurrent programs to change the value of this profile option for a particular instance of your running concurrent program. This profile option determines whether the concurrent managers print the concurrent program's output to the printer.

Implementing Concurrent Processing (See page 15 – 21)

Define Concurrent Program (See page 16 – 10)

CONC_PRINT_ STYLE

The print style of your concurrent program's output that you enter in the Print Style field when you register a concurrent program. You can use the routine **afpoput()** from your concurrent programs to change the value of this profile option for a particular instance of your running concurrent program.

Implementing Concurrent Processing (See page 15 – 21)
Implementing User Profiles (See page 13 – 5)

Define Concurrent Program (See page 16 – 10)

FND_PROFILE: User Profile APIs

This section describes user profile APIs you can use in your PL/SQL procedures. You can use these user profile routines to manipulate the option values stored in client and server user profile caches.

On the client, a single user profile cache is shared by multiple form sessions. Thus, when Form A and Form B are both running on a single client, any changes Form A makes to the client's user profile cache affect Form B's run–time environment, and vice versa.

On the server, each form session has its own user profile cache. Thus, even if Form A and Form B are running on the same client, they have separate server profile caches. Server profile values changed from Form A's session do not affect Form B's session, and vice versa.

Similarly, profile caches on the server side for concurrent programs are separate. Also, note that the server–side profile cache accessed by these PL/SQL routines is not synchronized with the C–code cache. If you put a value using the PL/SQL routines, it will not be readable with the C–code routines.

Any changes you make to profile option values using these routines affect only the run–time environment. The effect of these settings ends when the program ends, because the database session (which holds the profile cache) is terminated. To change the default profile option values stored in the database, you must use the User Profiles form.

FND_PROFILE.PUT

Summary

```
procedure FND_PROFILE.PUT
         (name      IN varchar2,
          value     IN varchar2);
```

Location

FNDSQF library and database (stored procedure)

Description

Puts a value to the specified user profile option. If the option does not exist, you can also create it with PUT.

All PUT operations are local—in other words, a PUT on the server affects only the server–side profile cache, and a PUT on the client affects only the client–side cache. By using PUT, you destroy the synchrony between server–side and client–side profile caches. As a result, we do not recommend widespread use of PUT.

Arguments (input)

name — The (developer) name of the profile option you want to set.

value — The value to set in the specified profile option.

FND_PROFILE.GET

Summary

```
procedure  FND_PROFILE.GET
         (name      IN varchar2,
          value     OUT varchar2);
```

Location FNDSQF library and database (stored procedure)

Description Gets the current value of the specified user profile option, or NULL if the profile does not exist. All GET operations are satisfied locally—in other words, a GET on the server is satisfied from the server–side cache, and a GET on the client is satisfied from the client–side cache.

The server–side PL/SQL package FND_GLOBAL returns the values you need to set Who columns for inserts and updates from stored procedures. You should use FND_GLOBAL to obtain Who values from stored procedures rather than using GET, which you may have used in prior releases of Oracle Applications.

Global APIs for PL/SQL Procedures (See page 27 – 4)

Arguments (input)

name — The (developer) name of the profile optionwhose value you want to retrieve.

Arguments (output)

value — The current value of the specified user profile option as last set by PUT or as defaulted in the current user's profile.

FND_PROFILE.VALUE

Summary

```
function FND_PROFILE.VALUE
         (name      IN varchar2) return varchar2;
```

Location FNDSQF library and database (stored function)

Description VALUE works exactly like GET**,** except it returns the value of the specified profile option as a function result.

Arguments (input)

name The (developer) name of the profile option whose value you want to retrieve.

User Profile C Functions

Oracle Application Object Library provides you with two functions you can call from concurrent programs you write in the C programming language. You can use these functions to store and retrieve profile option values.

☞ **Attention:** **fdpgov** and **fdppov** are not valid for Release 11. You should use **afpoget** and **afpoput** instead, and you should convert any old C code that uses **fdpgov** and **fdppov** to use **afpoget** and **afpoput** instead.

afpoget

Get the current value of a profile option. Returns TRUE if successful, FALSE if it cannot find the profile option value. Returns FALSE when retrieving a profile that exists but has no value. You must include the file **fdpopt.h** in your C code file (#include <fdpopt.h>) to use this C function. For concurrent programs, the current user is the user who submitted the concurrent request, and **afpoget()** reads the value at the time the request runs, not the time the user submitted the request. When the function **afpoget()** returns successfully, it sets option_value to the value of your requested user profile option. If you are not sure of the length of the value **afpoget()** will return, you should define option_value[] to be at least 241 characters long.

Syntax

```
boolean afpoget(option_name, option_value)
      text   *option_name;
      text   *option_value;
```

option_name the name of the profile option.

option_value the profile option value returned by the function.

afpoput

Change the value of a profile option for the current session. Create a profile option. Returns TRUE if successful, FALSE if it tries to change the value of a profile option for which the WRITE flag is set to No, or if it tries to create a profile option for which the ENABLE_CREATE flag is not set. You must include the file **fdpopt.h** in your C code file (#include <fdpopt.h>) to use this C function.

Use ENABLE_CREATE if you **afpoput()** to create an option if the option does not exist. This new option only exists within the current concurrent process, and it is not available to other processes. You can use the | (bitwise OR) operator to combine ENABLE_CREATE with the options ENABLE_WRITE and / or ENABLE_READ. You cannot use ENABLE_WRITE or ENABLE_READ to reset the privileges of an existing profile option. Use ENABLE_WRITE if you want to allow subsequent calls to **afpoput()** to change the value. Use ENABLE_READ if you want to allow subsequent calls to **afpoput()** to read the value.

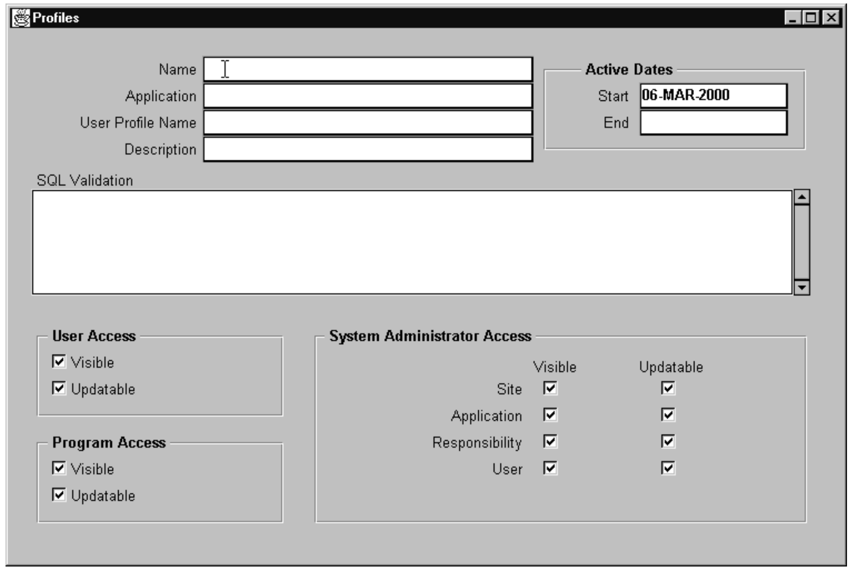
Syntax

```
boolean afpoput(option_name, option_value)
       text   *option_name;
       text   *option_value;
```

option_name — the name of the profile option.

option_value — The value to which you wish to change the profile option for the current session. All values are stored as text. The value may be at most 240 characters.

Profiles Window



Define a user profile option. You define new user profile options when you build an application. Once you define an option, you can control access for it at different profile levels.

Prerequisites

- Define your application using the Application window.

Profiles Block

You identify a profile option by application name and profile option name.

Name

The profile option name must be unique so that different profile options do not conflict. This is the name you use when you access your profile option using the Oracle Application Object Library profile option routines.

Application

Normally, you enter the name of the application you are building.

User Profile Name

This is the name your users see as their profile option, so it should be short but descriptive.

Description

Provide a better explanation of the content or purpose of this profile option. This description appears in a window with User Profile Name when a user or system administrator chooses profile options to set values.

Active Dates

Start Date/End Date

Enter the dates on which the profile option becomes active/inactive. The start date defaults to the current date, and if the end date is not entered, the option will be effective indefinitely. You cannot delete a user profile option, but you can disable it. Enter the current date if you want to disable the user profile option. If you wish to reactivate a disabled profile option, change the End Date to a date after the current date.

SQL Validation

If you want your profile option to provide a list of values (LOV) when the system administrator or user sets profile options, you must use the following syntax in the SQL Validation field.

To validate your user profile option, select the profile option value into the fields :PROFILE_OPTION_VALUE and :VISIBLE_OPTION_VALUE. The Profile Values form uses these fields to ensure that your user enters valid values for your profile option.

Syntax

```
SQL="SQL select statement"
COLUMN="column1(length), column2(length),..."
[TITLE="{title text|*application shortname:message name}"]
[HEADING="{heading1(length), heading2(length),...
```

|**application shortname*:*message name*|N}”]

SQL

A SELECT statement that selects the rows to display in your LOV. In the SQL statement you can specify column aliases, use an INTO clause to put values into form fields, display database values without selecting them into form fields (by selecting values INTO NULL), and mix values to put into form fields with display only values in the same INTO clause.

If you specify more than one column in your COLUMN option, the LOV displays the columns in the order you specify in your COLUMN statement.

Suggestion: Column aliases cannot be longer than 30 characters. Larger identifiers will cause errors.

The HEADING option overrides the COLUMN lengths and aliases.

This SQL statement differs from a normal SQL statement in some ways. First, if you want to include spaces in your column aliases, you must put a backslash and double quotes before and after the column alias, so that the LOV routine recognizes the double quotes as real double quotes, rather than the end of your parameter. For example, your SQL option might look like the following example:

```
SQL=”SELECT SALES_REPRESENTATIVE_ID,
   SALES_REPRESENTATIVE_NAME
INTO :PROFILE_OPTION_VALUE,
   :VISIBLE_OPTION_VALUE
   FROM OE_SALES_REPRESENTATIVES
   ORDER BY SALES_REPRESENTATIVE_NAME”
```

We recommend that you provide aliases for your column headings in the HEADING options below.

You can use GROUP BY or HAVING clauses in your SQL statement, but only in your main query; you cannot use them in sub–queries. You can use DISTINCT and ORDER BY clauses as you would normally.

Set functions such as MIN(), MAX(), SUM(), and COUNT() can be used in the SELECT or HAVING clause, but you cannot use them on the columns that you select into the PROFILE_OPTION_VALUE or VISIBLE_OPTION_VALUE fields.

Though you can use a fairly complex WHERE clause and / or an ORDER BY clause in your SQL definition, you cannot use UNION, INTERSECT, or MINUS in your main query. If you need a UNION, INTERSECT, or MINUS to select the proper values, you should create a view on your tables, then select from the view, or include these operators as part of a sub–query.

In addition, you cannot use a CONNECT BY or any other operation that would come after the WHERE clause of a SELECT statement.

Finally, if you use OR clauses, you should enclose them in parentheses.

We recommend that you put parentheses around complex columns in your SQL SELECT statements. For example, your SQL option could look like this:

```
SQL=”SELECT (DEPTNO ||’:’ ||DEPT_NAME)
   Department, LOCATION INTO
   :DEPT.DEPTNAME, :DEPT.LOCATION
   FROM DEPARTMENT”
```

`COLUMN`

Lists the names of columns (or column aliases) you want to display in your LOV window, the order in which to display them, and their display widths. If you specify more than one column in your COLUMN option, your LOV displays the columns in the order you list them. This order can differ from the column order in your SQL statement. You must specify column widths in the COLUMN= ”...” parameter, although any column widths you specify in the HEADING=”...” option below override these values.

You can specify static or dynamic column widths in your COLUMN option. Specify a static column width by following the column name with the desired width. Specify a dynamic width column by placing an asterisk instead of a number in the parentheses following the column name. When you specify dynamic width for a column, the LOV adjusts the size of the displayed column to accommodate the longest value in the list. Note that a dynamic column width may vary based on the subset of data queried. The following example shows a possible COLUMN option corresponding to the department and location example, and illustrates the use of static and dynamic column widths.

```
COLUMN="Department(20), LOCATION(*)"
```

If you do not use the HEADING option to supply column heading or suppress headings, then the LOV uses the names and widths from your COLUMN option to display the column headings. If you specify a column alias in your SQL statement and you want that column to appear in your QuickPick window, you must use that column alias in COLUMN. The column headings appear in the QuickPick window with the same upper– and lowercase letters as you define here. If your column alias has two words, you must put a backslash and double quotes on both sides of the alias. Column aliases cannot be longer than 30 characters. Using the first example from the SQL option, your COLUMN option would look like this:

```
COLUMN="\"Sales Representative\"(30)"
```

If your display width is smaller than your column name or column alias, the LOV uses the length of the column name or alias, even if you suppress headings in your LOV window (see the HEADING option). For your values to display properly, you *must* specify a number for the column width.

`TITLE` Text you want to display centered and highlighted on the top line of your QuickPick window. The default is no title.

You can specify a Message Dictionary token in your LOV definition by providing the application short name and the message name. Any title starting with "*" is treated as a Message Dictionary name, and the message contents are substituted for the title. For example:

```
TITLE="*FND:MY_MESG_NAME"
```

`HEADING` Lets you specify a list of column headings and column widths, separated by spaces or commas. There should be one heading in the HEADING="..." parameter for each column in the COLUMN="..." parameter. Specify column widths as numbers in parentheses following the column name, or as an asterisk in parenthesis for a dynamic column width.

Column widths you specify in the HEADING ="..." parameter override columns widths you specify in the COLUMN="..." parameter. We recommend setting the widths in the COLUMN option to * (dynamic width) when using the HEADING and COLUMN options together.

You can suppress headings in your LOV window altogether by setting HEADING="N".

You can specify a Message Dictionary token in your LOV definition by providing the application short name and the message name. Any heading starting with "*" is treated as a Message Dictionary name, and the message contents are substituted for the heading. For example:

```
HEADING="*FND:MY_MESG_NAME(*)"
```

If you do not provide an explicit TITLE and HEADING in your SQL validation, your profile has TITLE="user_profile_option_name" and HEADING="N" appended to the definition at runtime. This appended title overrides any heading defined in a COLUMN token or aliases in the SQL statement.

For example, suppose you have an option called SECURITY_LEVEL that uses the codes 1 and 2 to represent the values High and Low respectively. You should select the code column into :PROFILE_OPTION_VALUE and the meaning column into :VISIBLE_OPTION_VALUE. Then, if you want to change the meaning of your codes, you do not have to change your program or form logic. If the value of your profile option is user–defined, you can select the value into both fields. For example, suppose you have a table and form where you maintain equipment information, and you have a profile option called EQUIPMENT. You can select the same value into both :PROFILE_OPTION_VALUE and :VISIBLE_OPTION_VALUE.

Here is an example of a definition for a new profile option called SET_OF_BOOKS_NAME.

```
SQL="SELECT SET_OF_BOOKS_NAME, SET_OF_BOOKS_NAME \"Set of
Books\" '
    INTO :PROFILE_OPTION_VALUE, :VISIBLE_OPTION_VALUE,
    FROM SETS_OF_BOOKS"
COLUMN="\"Set of Books\"(30)"
```

If you do not enter validation criteria in this field, your user or system administrator can set any value for your profile option, if you allow them to update it.

If Oracle Application Object Library cannot successfully perform your validation, it does not display your profile option the user queries profiles options. If the profile option Utilities:Diagnostics is No, then no error messages appear either. For example, if a user cannot access the table you reference in your validation statement, Oracle Application Object Library does not display the profile option when the user queries profile options on the Profile Values window, and does not display any error message if Utilities:Diagnostics is set to No.

User Access

Visible

Indicate whether your end users can see and query this profile option in their personal profiles. Otherwise, they cannot query or update values for this option.

Updatable

Indicate whether your end users can change the value of this profile option using their Profile Values window. Otherwise, your system administrator must set values for this profile option.

Program Access Block

Visible

Indicate whether you can read the value of your profile option from a user exit or concurrent program.

If you enter Yes, you can construct your application to read the value of a user profile option using the Oracle Application Object Library profiles routines.

Updatable

Indicate whether you can change the value of this profile option using Oracle Application Object Library profiles routines.

System Administrator Access Block

Define the characteristics of your profile option at each profile level that the system administrator uses to define profile values. You can define the characteristics at the Site, Application, Responsibility and User levels.

Suggestion: You should specify Site–level characteristics of every user profile option you create so that the system administrator can assign a Site–level value for every profile option.

You should provide access to each option at the Site level. You can also provide access for any of the other three levels, Application, Responsibility, and User.

Profile option values set at the User profile level override values set at the Responsibility profile level, which override values set at the Application profile level. If no values are set at these three levels, then the value defaults to the value set at the Site profile level if the Site level value has been set.

If you want your end user to be able to update profile option values in the Profile Values window, that is, you chose Updatable in the User Access region, you must provide user visible and updatable access at the User level here.

Visible

Indicate whether your system administrator can see your profile option while setting user profile option values for the specified profile level.

Updatable

Indicate whether your system administrator can change the value of your profile option while setting user profile option values for the profile level you select.

CHAPTER

14

Flexfields

This chapter describes Oracle Application Object Library flexfields. It includes a conceptual overview that contains a summary of flexfields, definitions of key concepts, and an outline of the steps necessary to add flexfields to your application. This chapter also includes implementation sections that provide technical details of adding flexfields to your forms, and form descriptions that provide information about each form available to help you implement flexfields.

The following topics are covered:

- Overview of Flexfields
- Implementing Key Flexfields
- Implementing Descriptive Flexfields
- Adding Flexfields to Your Forms
- Flexfield Definition Procedures
- Key Flexfields Window
- Descriptive Flexfields Window

Overview of Flexfields

A flexfield is a field made up of segments. Each segment has a name you or your end users assign, and a set of valid values. There are two types of flexfields: key flexfields and descriptive flexfields.

For an explanation of flexfields features and concepts, as well as information on setting up flexfields in Oracle Applications, see the *Oracle Applications Flexfields Guide*. For information on entering and querying data using flexfields, see the *Oracle Applications User's Guide*.

Overview of Flexfield Concepts
Additional Terms and Concepts for Key Flexfields
Descriptive Flexfield Concepts
Overview of Setting Up Flexfields
Oracle Applications Flexfields Guide

Key Flexfields

Most businesses use codes made up of meaningful segments (intelligent keys) to identify accounts, part numbers, and other business entities. For example, a company might have a part number ”PAD–NR–YEL–8 1/2x14” indicating a notepad, narrow–ruled, yellow, and 14” by 8 1/2”. A key flexfield lets you provide your users with a flexible ”code” data structure that users can set up however they like using key flexfield segments. Key flexfields let your users customize your application to show their ”codes” any way they want them. For example, a different company might have a different code for the same notepad, such as ”8x14–PD–Y–NR”, and they can easily customize your application to meet that different need. Key flexfields let you satisfy different customers without having to reprogram your application.

In another example, Oracle General Ledger uses a key flexfield called the Accounting Flexfield to uniquely identify a general ledger account. At Oracle, we have customized this Accounting Flexfield to include six segments: company code, cost center, account, product, product line, and sub–account. We have also defined valid values for each segment, as well as cross–validation rules to describe valid segment combinations. However, other companies might structure their general ledger account fields differently. By including the Accounting Flexfield key flexfield, Oracle General Ledger can accommodate the needs of different companies. One company can customize the Accounting Flexfield to include six segments, while another company includes twelve segments, all without programming.

A key flexfield represents an intelligent key that uniquely identifies an application entity. Each key flexfield segment has a name you assign, and a set of valid values you specify. Each value has a meaning you also specify. Oracle General Ledger's Accounting Flexfield is an example of a key flexfield used to uniquely identify a general ledger account.

You can use key flexfields in many applications. For example, you could use a Part Flexfield in an inventory application to uniquely identify parts. Your Part Flexfield could contain such segments as product class, product code, size, color and packaging code. You could define valid values for the color segment, for example, to range from 01 to 10, where 01 means red, 02 means blue, and so on. You could even specify cross–validation rules to describe valid combinations of segment values. For example, products with a specific product code may only be available in certain colors.

Overview of Flexfield Concepts
Oracle Applications Flexfields Guide

Descriptive Flexfields

Descriptive flexfields let you satisfy different groups of users without having to reprogram your application, by letting you provide customizable "expansion space" on your forms. For example, suppose you have a retail application that keeps track of customers. Your Customers form would normally include fields such as Name, Address, State, Customer Number, and so on. However, your form might not include extra fields to keep track of customer clothing size and color preferences, or regular salesperson, since these are attributes of the customer entity that depend on how your users use your application. For example, if your retail application is used for a tool company, a field for clothing size would be undesirable. Even if you initially provide all the fields your users need, your users might later identify even more customer attributes that they want to keep track of. You add a descriptive flexfield to your form so that your users have the desired expansion space. Your users can also take advantage of the fact that descriptive flexfields can be context sensitive, where the information your application stores depends on other values your users enter in other parts of the form.

A descriptive flexfield describes an application entity, providing form and database expansion space that you can customize. Each descriptive segment has a name you assign. You can specify valid segment values or set up criteria to validate the entry of any value.

Oracle General Ledger includes a descriptive flexfield in its journal entry form to allow end users to add information of their own choosing. For example, end users might want to capture additional information about each journal entry, such as source document number or the name of the person who prepared the entry.

You could use a descriptive flexfield in a fixed assets application you build to allow further description of a fixed asset. You could let the structure of your assets flexfield depend on the value of an asset type field. For example, if asset type were ”desk”, your descriptive flexfield could prompt for style, size and wood type. If asset type were ”computer”, your descriptive flexfield could prompt for CPU chip and memory size.

Overview of Flexfield Concepts
Oracle Applications Flexfields Guide

Easy Customization

Flexibility is important. There is no way for you to anticipate all the form and database fields your end users might want, nor how each field should look as end user needs change. Using key and descriptive flexfields, you give end users the ability to customize your application to match their business needs, without programming. You should build a flexfield into your application whenever you need a flexible data structure.

Customizing a flexfield means specifying the prompt, length and data type of each flexfield segment. It also includes specifying valid values for each segment, and the meaning of each value to your application. You or your end users can even define cross–validation rules to specify valid combinations of segment values.

Ordinarily, your end users customize flexfields during application installation. However, you, as a developer, can customize flexfields while you are developing your application. Even if end users never change a flexfield once you have customized it, they can take advantage of useful flexfield features such as automatic segment validation, automatic segment cross–validation, multiple segment structures, and more.

Overview of Flexfield Concepts
Oracle Applications Flexfields Guide

Multiple Structures for a Single Flexfield

In some applications, different users need different segment structures for the same flexfield. Or, you might want different segments in a flexfield depending on, for example, the value of another form or database field.

Flexfields lets you define multiple segment structures for the same flexfield. Your flexfield can display different prompts and fields for different end users based on a data condition in your form or application data.

Oracle General Ledger, for example, provides different Accounting Flexfield structures for users of different sets of books. Oracle General Ledger determines which flexfield structure to use based on the value of a Set of Books user profile option.

Standard Request Submission Parameters

Most of the features used with your flexfield segments also apply to your parameter window for Standard Request Submission programs. For example, you can define security rules and special value sets for your report parameters.

Overview of Flexfields and Standard Request Submission
Oracle Applications Flexfield Manual

Definitions

For more explanation of flexfields features and concepts, see the *Oracle Applications Flexfields Guide.*

Overview of Flexfield Concepts
Additional Terms and Concepts for Key Flexfields
Descriptive Flexfield Concepts
Oracle Applications Flexfields Guide

Segment

For a key flexfield, a segment is a single piece of the complete code. For a descriptive flexfield, a segment is a single field or a single attribute of the entity. A segment is represented by a single column in a table.

Combination

For a key flexfield, a combination of segment values that make up the complete code or key. You can define valid combinations with simple cross–validation rules when you customize your key flexfield. Groups of valid combinations can be expressed as ranges.

Structure

A flexfield structure is a particular arrangement of flexfield segments. The maximum size of the structure depends on the individual flexfield. A flexfield may have one or more structures. Both key and descriptive flexfields can have more than one structure. Users can tailor structures for specific needs.

Combinations Table

For a key flexfield, a database table you include in your application to store valid combinations of key flexfield segment values. Each key flexfield must have a combinations table. It contains columns for each flexfield segment, as well as other columns. This is the same table you use as your entity table.

Combinations Form

For a key flexfield, a combinations form is the form whose base table (or view) is the combinations table. The only purpose of the combinations form is to maintain the combinations table. Most key flexfields have one combinations form, although some key flexfields do not have a combinations form. Key flexfields without combinations forms are maintained from other forms using dynamic insertion.

Dynamic Insertion

Dynamic insertion is the insertion of a new valid combination into a key flexfield combinations table from a form other than the combinations form.

For key flexfields whose combinations table does not contain any mandatory columns other than flexfield and WHO columns, you can choose to allow dynamic inserts when you set up your key flexfield. If you allow dynamic inserts, your user can enter new combinations of segment values using the flexfield window from a form other than the combinations form. If your end user enters a new combination that satisfies cross–validation rules, your flexfield dynamically inserts it into the combinations table. Otherwise, a message appears and the user is required to correct the segment values that violate the cross–validation rules.

If you create your key flexfield using a combinations table that contains mandatory columns other than flexfield or WHO columns, you cannot allow dynamic inserts, and your end user cannot enter new combinations through the flexfield window from any form other than the combinations form.

Flexfield Qualifier

A flexfield qualifier identifies a segment your end user should define when customizing your key flexfield. By specifying flexfield qualifiers when you build your application, you ensure your end user customizes your flexfield to include key segments that your application needs.

For example, suppose you build a general ledger accounting application that uses a key flexfield to uniquely identify accounts. Your application requires that one key segment be an account segment, and one be a balancing segment. You ensure your end user defines these key segments by defining two flexfield qualifiers, account and balancing. When customizing your accounting flexfield, your end user ties the account and balancing flexfield qualifiers to particular key segments. You, as the developer, need not know *which* key segment becomes the account or balancing segment, because the key flexfield takes care of returning account and balancing information to your application at run–time.

Segment Qualifier

A segment qualifier describes characteristics of key segment values. You use segment qualifiers to obtain information about segment values your end user enters while using your application.

For example, suppose your end user enters a value in the account segment of a flexfield that uniquely identifies general ledger accounts. Since you, as the developer, do not know which segment represents account, your application cannot reference the account value directly. However, you can construct your application so that each account value has an associated segment qualifier called ”Account type” that your application can easily reference.

Assume that account value 1000 (which means ”Cash”) has an account type of ”Asset”. Your application can reference this account type because your key flexfield returns it to a column you designate in your generic combinations table. Your application can contain logic that is conditional on account type.

You can define segment qualifiers when you define flexfield qualifiers. You can assign one or more segment qualifiers to each flexfield qualifier.

Structure Defining Column

A column you include in a combinations table or entity table so the flexfield can support multiple segment structures. You can construct your application so that it places a value in a structure defining column to determine the flexfield segment structure your end user sees.

For example, Oracle General Ledger places a "Chart of Accounts" identifier in the structure defining column of the combinations table for the Accounting Flexfield. As a result, Oracle General Ledger can provide different Accounting Flexfield structures (different charts of accounts) for different users.

Building a Flexfield into Your Application

To include a flexfield in an application you are building, you perform the following steps.

First, you decide which application entities require key or descriptive flexfields. You use a key flexfield to uniquely identify an entity that needs an intelligent key.

> ☞ **Attention:** We provide key flexfield information such as combinations table structure and form syntax so you can convert any existing non–Oracle Applications combinations forms you may have from SQL*Forms 2.3 to Oracle Forms 4.5. You may also use this information to integrate your custom forms and applications with key flexfields that Oracle Applications provides. For example, you may build foreign key forms that call Oracle Applications key flexfields. However, the API for key flexfields may change in future versions of Oracle Applications, so we recommend that you do not create any new key flexfields that are not provided by Oracle Applications.

You use a descriptive flexfield to provide context–sensitive expansion space for carrying additional information about an entity. To maximize your user's flexibility, you should consider defining a descriptive flexfield for every entity in your application.

After deciding that an application entity requires a flexfield, you design the flexfield into your applications database. You register the flexfield with Oracle Application Object Library, and if you like, assign flexfield and segment qualifiers for your key flexfields. Then, you develop application forms that include your flexfield and call Oracle Application Object Library routines to activate it.

After you are done defining a flexfield, you or your end user can customize it to include a specific set of segments.

Designing Flexfields into Your Application Database

You include flexfield columns in the database table that represents the application entity for which you are defining a flexfield. You include one column for each flexfield segment you or your end user might wish to customize. You need at least as many columns as the maximum number of segments a user would ever want in a single flexfield structure. If you have more segments than can fit on your screen when the flexfield window is open, you can scroll through them vertically.

For a key flexfield, a combinations table represents the application entity. A combinations table includes flexfield segment columns as well as other columns a key flexfield requires. Key flexfields provided by Oracle Applications already have combinations tables defined.

To permit the use of flexfield combinations from different application forms, you must include foreign key references to your combination table's unique ID column in other application tables. That way, you can display or enter valid combinations using forms not based on your combinations table. When you build a custom application that uses Oracle Applications key flexfields, you would include foreign key references in your custom application tables wherever you reference the flexfield.

To define a descriptive flexfield, you include descriptive segment columns in the application table you choose. You also include a structure defining column (sometimes called a context column), in case your end user wants to define multiple segment structures.

Implementing Key Flexfields (See page 14 – 11)
Implementing Descriptive Flexfields (See page 14 – 20)

Registering a Flexfield

You register a flexfield with Oracle Application Object Library after you design it into your database. By registering a flexfield, you notify Object Library that your flexfield exists in the database, and provide some basic information about it.

When you register a flexfield, you give it a name that end users see when they open your flexfield pop–up window (for example, ”Accounting Flexfield” or ”Vendor Flexfield”). End users can change the flexfield name you provide when they customize your flexfield.

Key Flexfields Window (See page 14 – 67)
Descriptive Flexfields Window (See page 14 – 73)

Building a Flexfield into a Form

To add a flexfield to a form, you define hidden form fields to represent the flexfield columns you defined in your application table (that is, unique ID, structure defining, segment, and other columns). You also define a visible form field to hold the concatenated segment value string that appears on your form after your end user enters segment values. You can optionally include a visible form field to hold a concatenated string of the meanings of each segment.

To activate your flexfield, you call Oracle Application Object Library routines from your form's triggers.

Implementing Key Flexfields (See page 14 – 11)
Implementing Descriptive Flexfields (See page 14 – 20)

Flexfields and Application Upgrades

Application upgrades do not affect the flexfields you have defined or customized. However, you may have to recompile your flexfields for some application upgrades. You recompile your key flexfields using the Key Flexfield Segments form, and you use the Descriptive Flexfield Segments form to recompile descriptive flexfields. Simply scroll through and save each row that defines your flexfield, and the form automatically recompiles your flexfield.

You can also recompile all of your frozen flexfields in one step from the operating system. See your installation manual for more information about compiling all your flexfields in one step after an application upgrade.

Implementing Key Flexfields (See page 14 – 11)

Key Flexfield Segments Window
Descriptive Flexfield Segments Window
Oracle Applications Flexfields Guide

Implementing Key Flexfields

To implement a key flexfield you must:

- Define key flexfield columns in your database
- Register your table with Oracle Application Object Library
- Register your key flexfield with Oracle Application Object Library
- Create key flexfield fields in your forms
- Add key flexfield routines to your forms

Key flexfields can be implemented for the following three types of forms, which are each implemented differently:

- Combinations form – The only purpose of a combinations form is to create and maintain code combinations. The combinations table (or a view of it) is the base table of this form and contains all the key flexfield segment columns. The combinations table also contains a unique ID column. This type of form is also known as a maintenance form for code combinations. You would have only one combinations form for a given key flexfield (though you might not have one at all). You cannot implement shorthand flexfield entry for a combinations form.
- Form with foreign key reference – The base table (or view) of the form contains a foreign key reference to a combinations table that contains the actual flexfield segment columns. You create a form with a foreign key reference if you want to use your form to manipulate rows containing combination IDs. The primary purpose of foreign key forms is generally unrelated to the fact that some fields may be key flexfields. That is, the purpose of the form is to do whatever business function is required (such as entering orders, receiving parts, and so on). You might have many foreign key forms that use a given key flexfield. You can choose to implement shorthand flexfield entry only for a form with a foreign key reference.
- Form with key flexfield range – The base table is a special ”combinations table” that contains two columns for each key flexfield segment so it can support both low and high values for each segment of your key flexfield. This type of form is rare.

For many applications, you would have one combinations form that maintains the key flexfield, where the key flexfield is the representation of an entity in your application. Then, you would also have one or more forms with foreign key references to the same key flexfield. For example, in an Order Entry/Inventory application, you might have a combinations form where you define new parts with a key flexfield for the part numbers. You would also have a form with foreign key reference where you enter orders for parts, using the key flexfield to indicate what parts make up the order.

The following diagrams show the relationship between a combinations form and a form with a foreign key reference, as well as the relationship of the two forms' underlying tables.

Figure 14 – 1

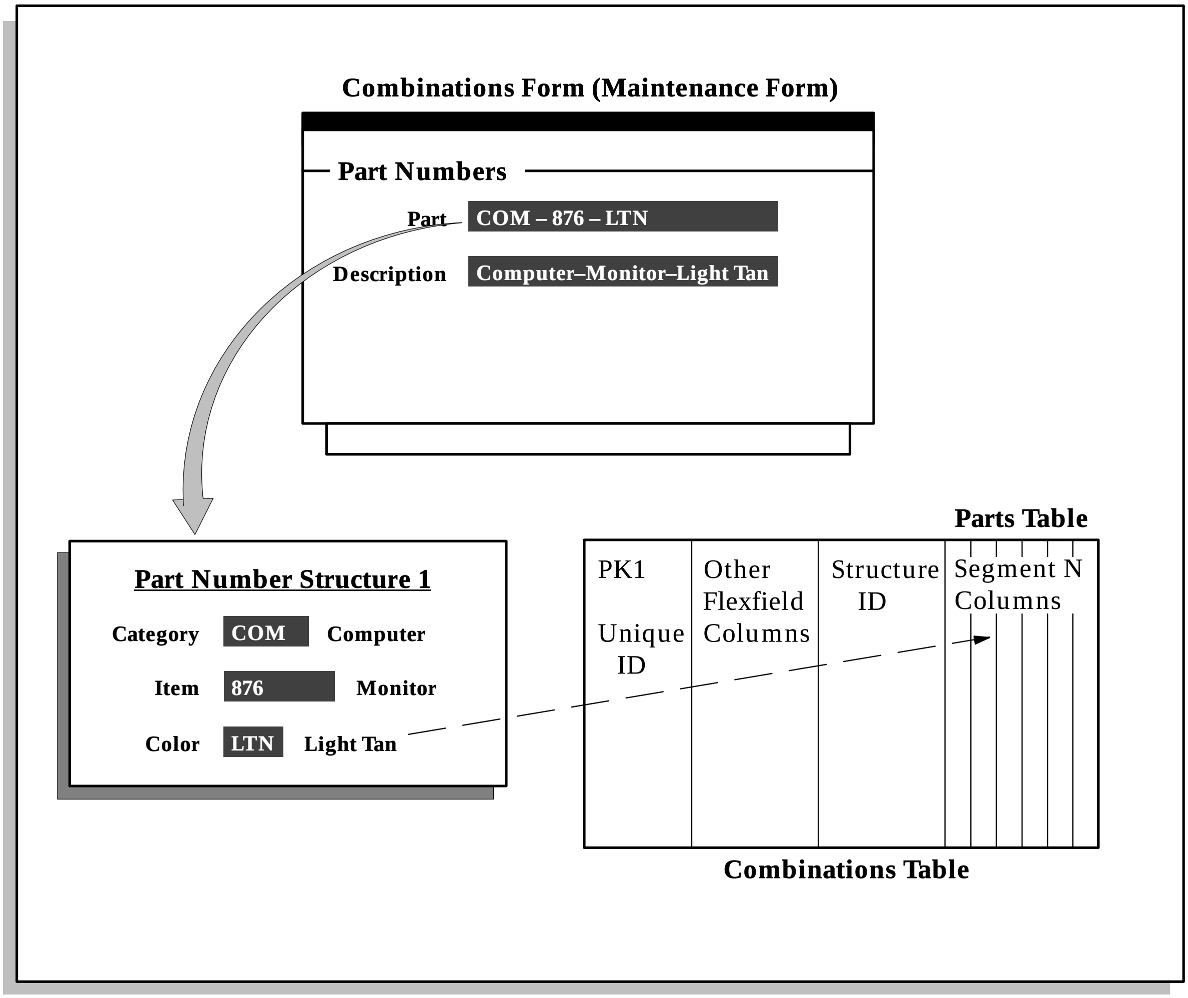


Figure 14 – 2

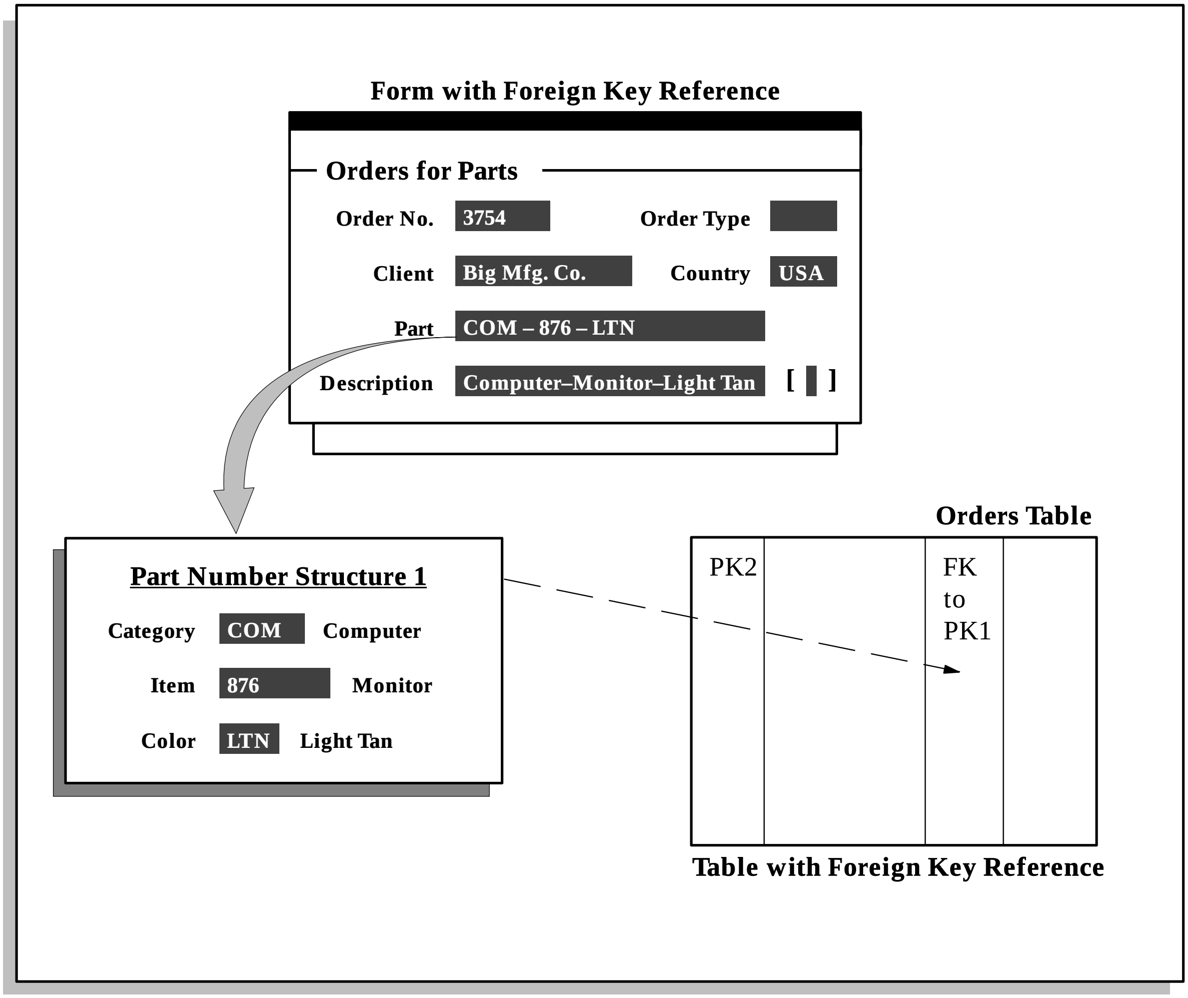
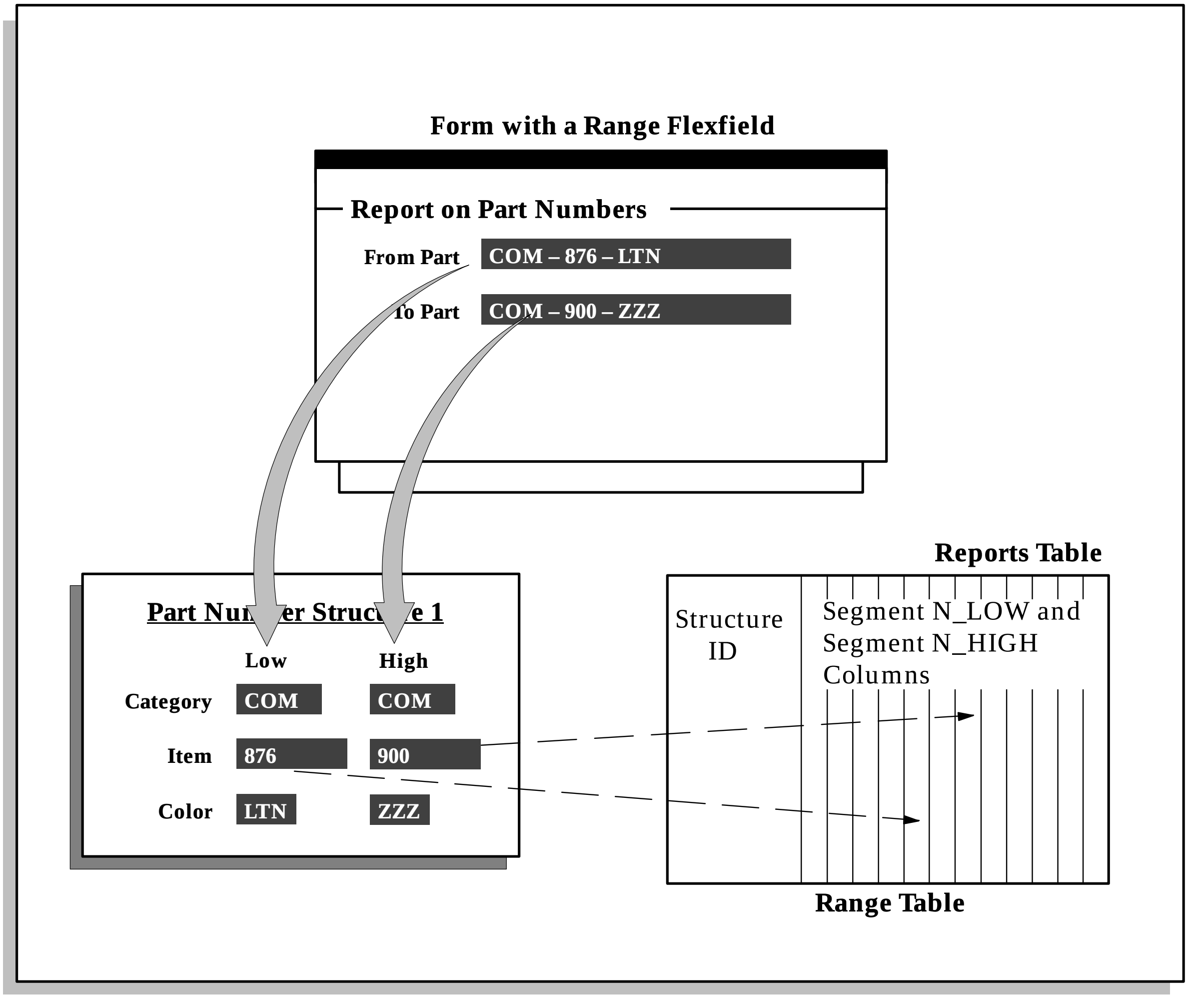

Form with Foreign Key Reference
Orders for Parts
Order No.
3754
Order Type
Client
Big Mfg. Co.
Country
USA
Part
COM – 876 – LTN
Description
Computer–Monitor–Light Tan
Part Number Structure 1
Category
COM
Computer
Item
876
Monitor
Color
LTN
Light Tan
Orders Table
PK2
FK to PK1
Table with Foreign Key Reference

Figure 14 – 3



Further, you can have another form, a form with a key flexfield range, that you use to manipulate ranges of your part numbers. This range flexfield form refers to the same key flexfield as both your combinations forms and your foreign key forms, though the ranges of segment values (a low value and a high value for each segment) are stored in the special range flexfield table that serves as the range form's base table.

Key Flexfield Range

A special kind of key flexfield you can include in your application to support low and high values for each key segment rather than just single values. Ordinarily, a key flexfield range appears on your form as two adjacent flexfields, where the leftmost flexfield contains the low values for a range, and the rightmost flexfield contains the high values.

In Oracle Application Object Library, we use a key flexfield range to help you specify cross–validation rules for valid combinations.

Defining Key Flexfield Database Columns

For each key flexfield you design into your application, you must create a combinations table to store the flexfield combinations that your users enter. You can build a special form to let them define valid combinations (the combinations form), or you can let Oracle Application Object Library dynamically create the combinations when users attempt to use a new one (from a form with a foreign key reference). You must have the combinations table even if you do not build a combinations form to maintain it. Key flexfields provided by Oracle Applications already have combinations tables defined.

In addition to the combinations table for your key flexfield, you may also have one or more tables for forms with foreign key references and for forms with key flexfield ranges.

Combinations table

Key flexfields support a maximum of 70 segment columns in a combinations table. For example, a combinations table includes a column for the unique ID that your key flexfield assigns to each valid combination. It also includes a structure defining column, in case your end user wants to define multiple structures. If you want to use segment qualifiers in your application, your table should include a derived column for each segment qualifier you define.

To create a key flexfield combinations table for your application entity, you must:

- Define an ID column to uniquely identify a row in your database table (type NUMBER, length 38, NOT NULL). You should name this column XXX_ID, where XXX is the name of your entity (for example, PART_ID). This column holds the unique ID number of a particular combination of segment values (also known as a code combination). The unique ID number is also known as a code combination ID, or CCID. Note that even though this column is a NUMBER(38) column, Oracle Application Object Library only supports code combination IDs up to two billion (2,000,000,000).

- Define a column for each key segment, SEGMENT1 through SEGMENT*n*, where *n* is the number of segments you want for your key flexfield (type VARCHAR2, length 1 to 60, all columns the same length, NULL ALLOWED). As a rule of thumb, you should create about twice as many segment columns as you think your users might ever need for a single key flexfield structure. The maximum number of key flexfield segment columns that Oracle Application Object Library supports in a combinations table is 70. However, for a combinations table that you want to use with a form with a foreign key reference, the number of segments is also limited by the maximum size of the field that holds the concatenated segment values and segment separators. That field is normally 2000 characters, so if you have 40 segments and 40 separators, each segment could only have an average width of about 49 characters. Having more segment columns than you need does not significantly impact either space requirements or performance. In fact, since you cannot add more segment columns to a flexfield combinations table once you have registered your flexfield, you should add at least a few "extra" segment columns to your combinations table initially to allow for future needs.
- Define SUMMARY_FLAG and ENABLED_FLAG (type VARCHAR2, length 1, NOT NULL).
- Define START_DATE_ACTIVE and END_DATE_ACTIVE (type DATE, NULL).
- Define a structure defining column (structure ID column) to allow multiple structures (type NUMBER, length 38, NOT NULL). You should name this column XXX_STRUCTURE_ID, where XXX is the name of your entity (for example, PART_STRUCTURE_ID). This column is optional but *strongly* recommended.
- Define a unique index on the unique ID column.
- Create an ORACLE sequence for your column with the same grants and synonyms as your combinations table (for insert privileges). Name your sequence *YOUR_TABLE_NAME*_S.
- Define the Who columns, LAST_UPDATE_DATE (type DATE, NOT NULL) and LAST_UPDATED_BY (type NUMBER, length

15, NOT NULL). All other Who columns should have NULL ALLOWED.

If you want your application to allow dynamic insertion of new valid combinations from a form with a foreign key reference, you must not include any mandatory application–specific columns in your combinations table. Your combinations table contains only the columns you need to define a key flexfield, such as unique ID, structure defining, and segment columns. It can, however, include *non*–mandatory application–specific columns and columns for derived segment qualifier values. If you include mandatory application–specific columns in your combinations table, you cannot allow dynamic insertion of new valid combinations from a form with a foreign key reference. If your table does not allow dynamic insertion, you *must* create a combinations form, based on your combinations table, for your users to create their valid combinations.

If you do not ever want to allow dynamic insertion of new valid combinations, you should develop a single form that allows your end user to directly display, enter, or maintain valid combinations in your combinations table (a combinations form). You can set up your key flexfield to not allow dynamic inserts (on a structure–by–structure basis) even if dynamic inserts are possible.

Warning: You should never insert records into a code combinations table through any mechanism other than Oracle Application Object Library flexfield routines. Doing so could lead to serious data corruption problems and compromise your applications.

Table with a foreign key reference

For each table you use as a base table for a form with a foreign key reference (to a combinations table's unique ID column), define one database column with the same name as the unique ID column in the corresponding combinations table (type NUMBER, length 38, and NULL or NOT NULL depending on your application's needs).

If you have a structure column in your combinations table, you also need to include a structure column in your foreign key table (with a corresponding form field), or provide some other method for passing the structure ID number to the NUM parameter in your calls to key flexfield routines. For example, you could store the structure number in a profile option and use the option value in the NUM parameter.

You do not need any SEGMENTn columns or other key flexfield columns for this type of table.

Table for a form with a key flexfield range

To create a table that supports a key flexfield range instead of a foreign key reference to a single combination, define SEGMENTn_LOW and SEGMENTn_HIGH columns, one pair for each SEGMENTn column in your combinations table (type VARCHAR2, length 1 to 60, all columns the same length, NULL).

If you have a structure column in your combinations table, you also need to include a structure column in your range table (with a corresponding form field), or provide some other method for passing the structure ID number to the NUM parameter in your calls to key flexfield routines. For example, you could store the structure number in a profile option and use the option value in the NUM parameter.

You do not need any other flexfield columns for this table.

Registering Your Key Flexfield Table

After you create your combinations table, you must register your table with Oracle Application Object Library using the Table Registration API.

Table Registration API (See page 3 – 11)
Key Flexfields Window (See page 14 – 67)

Registering Your Key Flexfield

Once your table is successfully registered, you register your key flexfield with Oracle Application Object Library. You register your key flexfield using the Key Flexfields window.

When you register a key flexfield, you identify the combinations table in which it resides, as well as the names of the unique ID and structure defining columns. Key flexfields provided by Oracle Applications are already registered.

Defining Qualifiers for Key Flexfields

When you register a key flexfield, you can define flexfield and segment qualifiers for it.

You should define flexfield qualifiers if you want to ensure your end user customizes your key flexfield to include segments your application needs. For example, Oracle General Ledger defines account and balancing flexfield qualifiers in the Accounting Flexfield to ensure that end users would define account and balancing segments.

You should define segment qualifiers if your application needs to know semantic characteristics of key segment values your end user enters. You assign one or more segment qualifiers to each flexfield qualifier. For example, Oracle General Ledger assigns a segment qualifier of ”account type” to the flexfield qualifier ”account” in the Accounting Flexfield. As a result, end users can define account value 1000 to mean ”Cash,” and assign it a segment qualifier value of ”Asset.”

Note that flexfield qualifiers can be *unique* or *global*, and *required* or not. You describe a flexfield qualifier as unique if you want your end user to tie it to one segment only. You describe a flexfield qualifier as global if you want it to apply to all segments. You can use a global flexfield qualifier as a convenient means for assigning a standard set of segment qualifiers to each of your flexfield's segments. You describe a flexfield qualifier as required if you want your end user to tie it to at least one segment.

In Oracle General Ledger's Accounting Flexfield, the ”Account” flexfield qualifier is required and unique because Oracle General Ledger requires one and only one account segment. Oracle General Ledger defines a flexfield qualifier as ”global” so the segment qualifiers ”detailed posting allowed” and ”detailed budgeting allowed” apply to each Accounting Flexfield segment.

Key Flexfield Segments Window
Segment Values Window
Value Sets Window
Oracle Applications Flexfield Manual

Derived Column

A column you include in a combinations table into which your flexfield derives a segment qualifier value. You specify the name of a derived column when you define a segment qualifier.

Add Your Flexfield to Your Forms

Once you have the appropriate table columns and your flexfield is registered, you can build your flexfield into your application forms.

Adding Flexfields to Your Forms (See page 14 – 25)

Implementing Descriptive Flexfields

You add a descriptive flexfield to provide customizable ”expansion space” for your entity. For example, suppose you have a retail application that keeps track of customer entities. Your entity table, CUSTOMERS, would normally include columns such as Name, Address, State, Sex, Customer Number, and so on. However, your table might not include extra columns to keep track of a customer's size and color preferences, or regular salesperson, since these are attributes of the customer entity that depend on how your users use your application. In fact, your users might later identify even more customer attributes that they want to keep track of. You add descriptive flexfield columns to your entity table (CUSTOMERS) so that your users have the desired expansion space. Your users can also take advantage of the fact that descriptive flexfields can be context sensitive, where the information your application stores depends on other values your users enter in the Customers form.

To implement a descriptive flexfield you must:

- Define descriptive flexfield columns in your database
- Register your table with Oracle Application Object Library
- Register your descriptive flexfield with Oracle Application Object Library
- Create descriptive flexfield fields in your forms
- Add descriptive flexfield routines to your forms

Planning for Reference Fields

Reference fields are fields from which a descriptive flexfield can get a context field value (optional, but recommended). Reference fields must be separate fields from the structure defining field (typically ATTRIBUTE_CATEGORY). Frequently, most of the existing (non–flexfield) fields in your form can also serve as reference fields. In general, fields that make good reference fields are those that have a short, fairly static list of possible values. You specify fields as reference fields when you register your descriptive flexfield in the Register Descriptive Flexfield form. Your users then have the option of using a reference field or not when they set up your flexfield.

For example, suppose you have a retail application that keeps track of ”customer” entities. Your Customers form would normally include fields such as Name, Address, State, Sex, Customer Number, and so on. Your end users may want to make the descriptive flexfield context–sensitive depending on what a user enters in the State field (if the state is Colorado, for example, you may want to keep track of customer preferences in ski–wear, while if the state is Florida, you may want to keep track of preferences in warm–weather–wear). Alternatively, your end users may want to make the descriptive flexfield context–sensitive depending on what a user enters in the Sex field (if the customer is female, for example, you may want to keep track of her size preferences using standard women's sizes, while if the customer is male, you may want to keep track of size preferences using standard men's sizes). By specifying both the State field and the Sex field as reference fields when you register your descriptive flexfield in the Register Descriptive Flexfield form, you give your users the option to set up the flexfield either way.

Suggestion: A descriptive flexfield can use only one form field as a reference field. You may derive the context field value for a descriptive flexfield based on more than one field by concatenating values in multiple fields into one form field and using this concatenated form field as the reference field.

Descriptive Flexfields WIndow (See page 14 – 73)

Descriptive Flexfield Segments Window
Oracle Applications Flexfield Manual

Defining Descriptive Flexfield Database Columns

To make your application very flexible, you should add descriptive flexfield columns to all of your entity tables.

Oracle Application Object Library reserves table names that contain the string "_SRS_" for the Standard Request Submission feature, so you should not give your descriptive flexfield table a name that includes this string.

To add descriptive flexfield columns into your database table, you:

- Define a column for each descriptive segment, ATTRIBUTE1 through ATTRIBUTE*n* (type VARCHAR2, length 1 to 150, all columns the same length, NULL ALLOWED).
- Define a structure defining column (context column) to identify your descriptive flexfield structures (type VARCHAR2, length 30, NULL ALLOWED). Although you can name this column

anything you wish, we recommend that you name it ATTRIBUTE_CATEGORY.

You should ensure you initially add enough segment columns to cover any future uses for your descriptive flexfield, since you cannot add extra segment columns to your flexfield later.

You determine the maximum number of segments you can have within a single structure when you define your ATTRIBUTEn columns in your table. You can define a maximum of 200 ATTRIBUTEn columns in one table. As a rule of thumb, you should create about twice as many segment columns as you think your users might ever need for a single descriptive flexfield structure.

Adding a Descriptive Flexfield to a Table with Existing Data

You can add flexfield columns to a table that has *never* had any flexfield columns but already contains data. However, you must be very careful not to create data inconsistencies in your application when you do so. To add your flexfield, you add columns, form fields, and invoke descriptive flexfield routines exactly the same as if you were creating a descriptive flexfield from the beginning. However, when you define your flexfield using the Descriptive Flexfield Segments form, you must consider whether any of the segments should use value sets that *require* values. If none of your new segments requires a value, your users will simply see an empty descriptive flexfield when they query up existing records. For this case, no further action is necessary.

For the case where one or more of your segments *require* values, you need to perform extra steps to prevent data inconsistencies. The simplest way to do this is to define your segment structures completely, navigate to your form with the new descriptive flexfield, query up each record in your table, and enter values in the descriptive flexfield for each record. Save your changes for each record. This method, while tedious, ensures that all values go into the correct columns in your entity table, including the structure defining (context) column.

For very large tables, you can add the values to your table directly using SQL*Plus. You need to update each row in your table to include a context field value (the structure defining column) as well as segment values, so you must first determine the segment/column correspondences for your structures. Your context (structure) values must exactly match your context field values in the Descriptive Flexfield Segments form. For example, if your context field value is mixed case, what you put in the structure column must match the mixed case. If you put an invalid context value into the structure column, a purely context–sensitive flexfield does not pop up at all for that record. If you have global segments enabled, the flexfield window will open. If Override Allowed is set to Yes, you will see the bad context field value in the context field of the window.

Note that you should never use SQL*Plus to modify data in Oracle Application Object Library tables.

Protected Descriptive Flexfields

In some cases, you may want to create a descriptive flexfield that cannot be inadvertently changed by an installer or user. This type of flexfield is called a protected descriptive flexfield. You build a protected descriptive flexfield the same way you build a normal descriptive flexfield. The main difference is that you check the Protected check box in the Descriptive Flexfields form after defining your segment structures. Once a descriptive flexfield is protected, you cannot query or change its definition using the Descriptive Flexfield Segments form. You should define your descriptive flexfield segments before you check the Protected check box in the Descriptive Flexfields form.

Descriptive Flexfields WIndow (See page 14 – 73)

In a case where your database table already includes a descriptive flexfield, you need to define segment columns that have names other than ATTRIBUTEn. For special purpose flexfields such as protected descriptive flexfields, you can name your columns anything you want. You explicitly enable these columns as descriptive flexfield segment columns when you register your descriptive flexfield. Note that you must also create a structure–defining column for your second flexfield. Flexfields cannot share a structure column.

If your database table contains segment columns with names other than ATTRIBUTEn, you create hidden fields corresponding to those columns instead.

Registering Your Descriptive Flexfield Table

After you add descriptive flexfield columns to your table, you must register your table with Oracle Application Object Library using the Table Registration API.

Table Registration API (See page 3 – 11)

Registering Your Descriptive Flexfield

You must register your descriptive flexfield with Oracle Application Object Library. You register your descriptive flexfield using the Register Descriptive Flexfield form. When you register a descriptive flexfield, you identify the application table in which it resides and the name of the structure defining column. If you have created reference fields in your form, you should enter their names as "context fields" when you register your flexfield.

Descriptive Flexfields Window (See page 14 – 73)

Add Your Flexfield to Your Forms

Once you have the appropriate table columns and your flexfield is registered, you can build your flexfield into your application forms.

Adding Flexfields to Your Forms (See page 14 – 25)

Adding Flexfields to Your Forms

There are four basic parts to calling a flexfield from an Oracle Forms window. These steps assume that your flexfield is already registered and defined in Oracle Application Object Library and that the flexfield table and columns already exist. These steps apply to both key and descriptive flexfields.

▶ **To code a flexfield into your form:**

- Create your hidden fields
- Create your displayed fields
- Create your flexfield definition
- Invoke your flexfield definition from several event triggers

Create Your Hidden Fields

In general, you create your hidden flexfield fields as part of creating your default form block from the database table (or view). Set the canvas property of the flexfield fields to null (so they do not appear on a canvas).

Your hidden ID (for key flexfields only), structure field, and segment or attribute fields must be text items on the null canvas. Note that these must be text items rather than display items, and that they should use the TEXT_ITEM property class. Set the field query lengths to 255 for most fields, with a query length of 2000 for hidden ID fields.

☞ **Attention:** You should never create logic that writes values to the hidden fields directly. Since the flexfield keeps track of whether a record is being inserted, updated, etc., putting values in these fields by any method other than the flexfield itself (or a query from the database) may cause errors and data corruption.

In some foreign key forms for key flexfields, you may need to create extra non–database fields that represent the segment columns (SEGMENT1 through SEGMENTn) in your combinations table. Put your SEGMENT1 through SEGMENTn fields on the null canvas (field length the same as your SEGMENTn columns). These fields help Oracle Application Object Library to create new code combinations from your form with a foreign key reference (using dynamic insertion).

Normally, Oracle Application Object Library can create new code combinations (dynamic insertion) from your form with a foreign key reference using only the concatenated segment values field. However, if you expect the concatenated length of your flexfield to be defined to be larger than 2000 (the sum of the defined segments' value set maximum sizes plus segment separators), then you should create these non–database fields to support the dynamic creation of new combinations from your form.

If you do not have these fields and your users define a long flexfield (> 2000 characters), your users can experience truncation of key flexfield data when trying to create new combinations.

If your key flexfield is registered with Dynamic Inserts Feasible set to No, you do not need to add these fields, though they are recommended. If you do not create these fields, and your users define a long flexfield, your users may see empty flexfield segments upon entering the flexfield pop–up window after a query. These blank segments do not adversely affect the underlying data, nor do they adversely affect flexfield changes if your user updates those segments after querying.

If you use these fields and you have more than one key flexfield in the same row (in a block) of your form, you should also create one extra set of non–database segment fields *per* flexfield. So, if you have three foreign–key–reference flexfields in your block, you should have four sets of segment fields (for example, SEGMENT1 to SEGMENTn as the main set; and SEGMENT1_A to SEGMENTn_A, SEGMENT1_B to SEGMENTn_B, and SEGMENT1_C to SEGMENTn_C as the extra sets). In addition, you should use the USEDBFLDS="Y" argument for your flexfield definition routine calls. When you do so, you must write trigger logic to explicitly copy the appropriate values into or out of these fields before your flexfield routine calls. You must copy your values into the main set from the appropriate extra set before the WHEN–NEW–ITEM–INSTANCE and the PRE–INSERT and PRE–UPDATE flexfield event calls. You must copy your values out of the main set into the appropriate extra set after the POST–QUERY, WHEN–NEW–ITEM–INSTANCE, WHEN–VALIDATE–ITEM, PRE–INSERT, or PRE–UPDATE calls.

For a descriptive flexfield, it is possible (though not recommended) to create your form such that the table containing the descriptive flexfield columns is *not* the base table (or included in the base view) of the form. To do this, you create all the hidden fields (the ATTRIBUTEn fields and the structure defining field) as non–database fields on the null canvas. Then, code trigger and table handler logic that keeps the data in the two tables synchronized. For example, when your form updates your base table, your ON_UPDATE table handler should update the ATTRIBUTEn and structure defining columns in the descriptive flexfield table. Likewise, when your form inserts new records, you should have logic in your ON_INSERT table handler that inserts into the descriptive flexfield table. Descriptive flexfields never write directly to a table (base table or otherwise); they always write to the hidden segment fields.

Create Your Displayed Fields

Create your concatenated segments field as a 2000 character displayed, non–database text item for either key or descriptive flexfields. For a range flexfield, you create two non–database fields with the same name but with the suffixes _LOW and _HIGH.

Use the TEXT_ITEM property class for your key and range flexfields. For a descriptive flexfield, use the property class TEXT_ITEM_DESC_FLEX and name the field DESC_FLEX.

Create Your Flexfield Definition

Call a flexfield definition procedure from your WHEN–NEW–FORM–INSTANCE trigger to set up your flexfield. Using this procedure, you specify the block and fields for your flexfield and its related fields, the flexfield you want, and other arguments. See: Flexfield Definition Procedures: page 14 – 30.

You may need to enable, disable, or modify your flexfield definition depending on conditions in the form. For example, you may want to have a flexfield be updatable under some conditions but not under other conditions. In this case you should also call an UPDATE_DEFINITION procedure after calling the appropriate DEFINE procedure. See: Updating Flexfield Definitions: page 14 – 57.

Invoke Your Flexfield Definition from Several Event Triggers

Code handlers for special procedures into several form level triggers. These procedures fire your flexfield at certain events such as WHEN–NEW–ITEM–INSTANCE, WHEN–VALIDATE–ITEM, and PRE–INSERT.

You call your flexfields from form level triggers using the FND_FLEX.EVENT(EVENT) procedure. You can also call your flexfields using this procedure from within your own procedures. This procedure takes the event name as its argument. Call FND_FLEX.EVENT in the following events:

Trigger	Procedure
PRE–QUERY	FND_FLEX.EVENT('PRE–QUERY');
POST–QUERY	FND_FLEX.EVENT('POST–QUERY');
PRE–INSERT	FND_FLEX.EVENT('PRE–INSERT');
PRE–UPDATE	FND_FLEX.EVENT('PRE–UPDATE');
WHEN–VALIDATE–RECORD	FND_FLEX.EVENT('WHEN–VALIDATE–RECORD');
WHEN–NEW–ITEM–INSTANCE	FND_FLEX.EVENT('WHEN–NEW–ITEM–INSTANCE');
WHEN–VALIDATE–ITEM	FND_FLEX.EVENT('WHEN–VALIDATE–ITEM');

Table 14 – 1 (Page 1 of 1)

These calls should usually be coded into your form as form–level triggers. If you define any of these triggers at the block or field level, you need to make sure the block or field level triggers have execution style set to "Before" so the form–level flexfield calls still execute, or you should include these procedure calls in those triggers as well.

While we recommend you code all the flexfields triggers at the form level for convenience and consistency, having the triggers at form level may cause performance problems for very large or complicated forms. In that case, you may code the PRE–QUERY, POST–QUERY, PRE–INSERT, PRE–UPDATE, and WHEN–VALIDATE–RECORD triggers at the block level on all blocks that have flexfields (key or descriptive). You would then code the WHEN–NEW–ITEM–INSTANCE and WHEN–VALIDATE–ITEM at the item level for items on which the flexfields are defined.

You only need to code one set of these triggers regardless of how many flexfields you have in your form (assuming these triggers are at the form level).

The following three procedures are already in the TEMPLATE form as part of the APPCORE library and have the appropriate FND_FLEX.EVENT calls in them.

Trigger	Procedure
KEY–EDIT	APP_STANDARD.EVENT('KEY–EDIT');
KEY–LISTVAL	APP_STANDARD.EVENT('KEY–LISTVAL');
POST–FORM	APP_STANDARD.EVENT('POST–FORM');

Table 14 – 2 (Page 1 of 1)

☞ **Attention:** If you have a block or item level POST–QUERY trigger that resets the query status of a record, you must set the Execution Style of that block or item level POST–QUERY trigger to After. Because the flexfield POST–QUERY logic updates field values for your flexfield, the record must be reset to query status after that logic has fired.

Opening a Flexfield Window Automatically

By default, descriptive flexfields open automatically without any special code so long as the user profile Flexfields:Open Descr Window is not set to No.

Normally, key flexfields do not open automatically. However, if you do need your key flexfield to open automatically, you must add an extra call to your form. You must put your WHEN–NEW–ITEM–INSTANCE trigger at the field level, instead of the form level, on the field that contains the flexfield. Within that trigger, you must call two flexfield routines as follows:

```
FND_FLEX.EVENT('WHEN-NEW-ITEM-INSTANCE');
FND_FLEX.EVENT('KEY-EDIT');
```

The KEY–EDIT call opens the window. You must have the WHEN–NEW–ITEM–INSTANCE call before the KEY–EDIT call to ensure that the window opens correctly.

Flexfield Definition Procedures

Flexfields packages and procedures are included in the FNDSQF library. Call item handlers that contain the following procedures from your WHEN–NEW–FORM–INSTANCE trigger to define key, range or descriptive flexfields.

Flexfield Type	Procedure Name
Key flexfield	FND_KEY_FLEX.DEFINE
Range (or type) flexfield	FND_RANGE_FLEX.DEFINE
Descriptive flexfield	FND_DESCR_FLEX.DEFINE

Table 14 – 3 (Page 1 of 1)

When you call these procedures, you specify three types of arguments:

- location(s) of the flexfield (block and fields, including the concatenated values field, the ID field if any, and any description or related fields)
- specific registered flexfield you want (application, flexfield, and structure if necessary)
- any additional arguments

If you have more than one flexfield, you call a complete flexfield definition procedure for each of your flexfields from handlers in the same WHEN–NEW–FORM–INSTANCE trigger.

Key Flexfield Definition Syntax

Use FND_KEY_FLEX.DEFINE for a key flexfield on a foreign key or combinations form.

☞ **Attention:** We provide combinations form syntax so you can convert any existing non–Oracle Applications combinations forms you may have from SQL*Forms 2.3 to Oracle Forms 4.5. However, the API for key flexfields may change in future versions of Oracle Applications, so we recommend that you do not create any new key flexfields that are not provided by Oracle Applications.

```
FND_KEY_FLEX.DEFINE(

  /* Arguments that specify flexfield location */
     BLOCK=>'block_name',
     FIELD=>'concatenated_segments_field_name',
     [DESCRIPTION=>'description_field_name',]
     [ID=>'Unique_ID_field',]
     [DATA_FIELD=>'concatenated_hidden_IDs_field',]

  /* Arguments that specify the flexfield */
     APPL_SHORT_NAME=>'application_short_name',
     CODE=>'key_flexfield_code',
     NUM=>'structure_number',

  /* Other optional parameters */
    [VALIDATE=>'{FOR_INSERT|FULL|PARTIAL|NONE|
              PARTIAL_IF_POSSIBLE}',]
    [VDATE=>'date',]
    [DISPLAYABLE=>'{ALL | flexfield_qualifier |
           segment_number}[\\0{ALL |
           flexfield_qualifier | segment_number}]',]
    [INSERTABLE=>'{ALL | flexfield_qualifier |
           segment_number}[\\0{ALL |
           flexfield_qualifier | segment_number}]',]
    [UPDATEABLE=>'{ALL | flexfield_qualifier |
           segment_number}[\\0{ALL |
           flexfield_qualifier | segment_number}]',]
    [VRULE=>'flexfield qualifier\\n
           segment qualifier\\n
           {I[nclude]|E[xclude]}\\n
           APPL=application_short_name;
           NAME=Message Dictionary message name\\n
           validation value1\\n
           validation value2...
           [\\0flexfield qualifier\\n
           segment qualifier\\n
           {I[nclude]|E[xclude]}\\n
           APPL=application_short_name;
```

```
        NAME=Message Dictionary message name\\n
        validation value1\\n
        validation value2...]',]
  [COPY=>'block.field\\n{ALL | flexfield
        qualifier | segment_number}
        [\\0block.field\\n{ALL | flexfield
        qualifier | segment_number}]',]
  [DERIVED=>'block.field\\nSegment qualifier',]
  [DERIVE_ALWAYS=>'{Y|N}',]
  [DINSERT=>'{Y|N}',]
  [VALATT=>'block.field\\n
        flexfield qualifier\\n
        segment qualifier',]
  [TITLE =>'Title',]
  [REQUIRED=>'{Y|N}',]
  [AUTOPICK=>'{Y|N}',]
  [USEDBFLDS=>'{Y|N}',]
  [ALLOWNULLS=>'{Y|N}',]
  [DATA_SET=>'set number',]
  [COLUMN=>'{column1(n) | column1 alias(n)
        [, column2(n), ...] [INTO block.field]}',]
  [WHERE_CLAUSE=>'where clause',]
  [COMBQP_WHERE=>'{where clause|NONE}',]
  [QUERY_SECURITY=>'{Y|N|}',]
  [QBE_IN=>'{Y|N}',]
  [READ_ONLY=>'{Y|N}',]
  [LONGLIST=>'{Y|N}',]
  [NO_COMBMSG=>'APPL=<application_short_
        name>;NAME=<message_name>',]
  [AUTOCOMBPICK=>'{Y|N}',]
  [LOCK_FLAG=>'{Y|N}',]
  [HELP=>'APPL=<application_short_name>;
         TARGET=<target_name>']
);
```

You should not use a colon (:) in block.field references for the VALATT, COPY, or DERIVED arguments. The arguments for these routines go to an Oracle Application Object Library cover routine and are not directly interpreted in PL/SQL.

Range (Type) Flexfield Definition Syntax

Use FND_RANGE_FLEX.DEFINE for a range flexfield. You use the same procedure for a ”type” flexfield (which may also include range flexfield segments) that contains extra fields corresponding to each segment of the related key flexfield. For example, a type flexfield for the Accounting Flexfield might contain one field for each Accounting Flexfield segment, but you might enter only the values Yes or No in those fields, instead of normal segment values. The Assign Function Parameters form uses a type flexfield for its segment usage field (you enter ”Yes” for any segment whose value you want to use). You may build a type flexfield that contains more than one ”type column” (a ”column” of fields in the flexfield pop–up window that correspond to the actual segment fields). If you do, you can specify your TYPE_ argument values multiple times, using \\0 to separate the values.

☞ **Attention:** You should *not* append ”_LOW” or ”_HIGH” to the FIELD, DESCRIPTION, DATA_FIELD or other values, since this procedure appends them automatically. When you use more than one type column, ensure that all TYPE_ arguments specify type columns in the same order to avoid having argument values applied to the wrong type column.

```
FND_RANGE_FLEX.DEFINE(

  /* Arguments that specify flexfield location */
     BLOCK=>'block_name',
     FIELD=>'concatenated_segments_field_name',
     [DESCRIPTION=>'description_field_name',]
     [DATA_FIELD=>'concatenated_hidden_IDs_field',]

  /* Arguments that specify the flexfield */
     APPL_SHORT_NAME=>'application_short_name',
     CODE=>'key_flexfield_code',
     NUM=>'structure_number',
```

```
/* Other optional parameters */
  [VALIDATE=>'{PARTIAL|NONE}',]
  [VDATE=>'date',]
  [DISPLAYABLE=>'{ALL | flexfield_qualifier |
         segment_number}[\\0{ALL |
         flexfield_qualifier | segment_number}]',]
  [INSERTABLE=>'{ALL | flexfield_qualifier |
         segment_number}[\\0{ALL |
         flexfield_qualifier | segment_number}]',]
  [UPDATEABLE=>'{ALL | flexfield_qualifier |
         segment_number}[\\0{ALL |
         flexfield_qualifier | segment_number}]',]
  [VRULE=>'flexfield qualifier\\n
         segment qualifier\\n
         {I[nclude]|E[xclude]}\\n
         APPL=application_short_name;
         NAME=Message Dictionary message name\\n
         validation value1\\n
         validation value2...
         [\\0flexfield qualifier\\n
         segment qualifier\\n
         {I[nclude]|E[xclude]}\\n
         APPL=application_short_name;
         NAME=Message Dictionary message name\\n
         validation value1\\n
         validation value2...]',]
  [TITLE =>'Title',]
  [REQUIRED=>'{Y|N}',]
  [AUTOPICK=>'{Y|N}',]
  [USEDBFLDS=>'{Y|N}',]
  [ALLOWNULLS=>'{Y|N}',]
  [DATA_SET=>'set number',]
  [READ_ONLY=>'{Y|N}',]

/* Parameters specific to type flexfields */
  [TYPE_FIELD=>'block.concatenated_type_values_
         field\\ntype field suffix',]
  [TYPE_VALIDATION=> 'Value set name\\n
         Required\\nDefault value',]
```

```
    [TYPE_SIZES=>'type_value_display_
           size\\nDescription_display_size',]
    [TYPE_HEADING=>'type column heading',]
    [TYPE_DATA_FIELD=>'block.type_data_field',]
    [TYPE_DESCRIPTION=>'block.type_
           description_field',]
    [SCOLUMN=>'single column title',]
    [HELP=>'APPL=<application_short_name>;
            TARGET=<target_name>']
);
```

Attention: TYPE_FIELD, TYPE_DATA_FIELD and TYPE_DESCRIPTION require the `block.fieldname` construction, unlike other flexfield arguments that specify field names without block names.

Descriptive Flexfield Definition Syntax

Use FND_DESCR_FLEX.DEFINE for a descriptive flexfield.

```
FND_DESCR_FLEX.DEFINE(
  /* Arguments that specify the flexfield location */
    BLOCK=>'block_name',
    FIELD=>'field_name',
    [DESCRIPTION=>'description_field_name',]
    [DATA_FIELD=>'concatenated_hidden_IDs_field',]

  /* Arguments that specify the flexfield */
    APPL_SHORT_NAME=>'application_short_name',
    DESC_FLEX_NAME=>'descriptive flexfield_name'

  /* Other optional parameters  */
    [VDATE=>'date',]
    [TITLE =>'Title',]
    [AUTOPICK=>'{Y|N}',]
    [USEDBFLDS=>'{Y|N}',]
    [READ_ONLY=>'{Y|N}',]
    [LOCK_FLAG=>'{Y|N}',]
    [HELP=>'APPL=<application_short_name>;
```

```
            TARGET=<target_name>']
        );
```

Flexfield Definition Arguments

The following arguments apply to all types of flexfields unless noted otherwise. For those arguments that you would want to specify more than once, you separate the multiple argument values using \\0 (as noted).

Arguments that Specify the Flexfield Location

BLOCK Name of the block that contains your flexfield. Your value field, ID field (if any), and description field (if any) must all be in the same block.

FIELD Name of the field where you want to put your flexfield. This is a displayed, non–database form field that contains your concatenated segment values plus delimiters.

DESCRIPTION Description field for your flexfield. This is a displayed, non–database, non–enterable field that contains concatenated descriptions of your segment values. If you do not specify the DESCRIPTION parameter, your form does not display concatenated segment descriptions.

ID For a key flexfield only. Specify the field, if any, that contains the unique ID (CCID) for your key flexfield.

DATA_FIELD The concatenated hidden IDs field is a non–displayed form field that contains the concatenated segment hidden IDs.

Arguments that Specify which Flexfield to Use

APPL_SHORT_NAME Shortname of the application with which your flexfield is registered.

CODE Key or range flexfields only. The short code that identifies your flexfield. This is the flexfield code specified in the Key Flexfields form. This code

must match the registered code, such as GL# for the Accounting Flexfield in Oracle Applications.

NUM

Key or range flexfields only. The structure number (or the :block.field reference containing the structure number) that identifies your key flexfield structure.

You can specify the non–displayed database *:block.field* that holds the identification number of your flexfield structure. You may also specify :$PROFILES$.*your_profile_option_name* to retrieve a value you set in a user profile option. You can "hardcode" a structure number, such as 101, into this parameter instead of providing a field reference, but such a number prevents you from using multiple structures for your flexfield. You must use this option if you are using multiple structures.

You can use the following SQL statement to retrieve the structure identification numbers for your flexfield:

```
SELECT ID_FLEX_NUM, ID_FLEX_STRUCTURE_NAME
FROM FND_ID_FLEX_STRUCTURES
WHERE ID_FLEX_CODE = 'flexfield code';
```

where *flexfield code* is the code you specify when you register your flexfield.

The default value for NUM is 101.

DESC_FLEX_NAME

Descriptive flexfields only. The registered name that identifies your descriptive flexfield.

Other Optional Arguments

If you do not specify a particular optional argument, the flexfield routine uses the usual default value for the argument.

VALIDATE

Key or range flexfields only. For a key flexfield, you typically use FOR_INSERT for a combinations form and FULL for a foreign key form. For a range flexfield, you typically use NONE to allow users to enter any value into a segment or PARTIAL to ensure that users enter valid individual segment values that do not necessarily make up an actual valid combination.

Use a validation type of FULL for a foreign key form to validate all segment values and generate a new code combination and dynamically insert it into the combinations table when necessary. If you specify FULL, your flexfield checks the values your user enters against the existing code combinations in the code combinations table. If the combination exists, your flexfield retrieves the code combination ID. If the combination does not exist, your flexfield creates the code combination ID and inserts the combination into the combinations table. If you (or an installer) define the flexfield structure with Dynamic Inserts Allowed set to ”No”, then your flexfield issues an error message when a user enters a combination that does not already exist. In this case, your flexfield does not create the new code combination. FULL is the usual argument for a form with a foreign key reference.

Use PARTIAL for a form where you want to validate each individual segment value but *not* create a new valid combination or check the combinations table for an existing combination. You would use PARTIAL when you want to have application logic that requires flexfield segment values but does not require an actual code combination. For example, the Oracle Applications Shorthand Aliases form requires that a user enters valid values for each segment, but does not require (or check) that the actual code combination already exists in the combinations table. The Shorthand Aliases form does not create the combination, either. PARTIAL_IF_POSSIBLE is a special case of PARTIAL. If you have dependent segments in your flexfield (with independent segments), PARTIAL does not provide a list of values on the dependent segment if the user has not entered a value for the associated independent segment. PARTIAL_IF_POSSIBLE, however, will attempt to provide a list of values on the dependent segment. That list of values contains all dependent values for all values of the associated independent segment (so, you would see multiple values 000 if that were your default dependent value).

Use NONE if you wish no validation at all.

The default value for a key flexfield is FULL. The default value for a range flexfield is NONE.

VDATE

date is the validation date against which the Start Date and End Date of individual segment values is checked. You enter a Start Date and End Date for each segment value you define using the Segment Values form.

For example, if you want to check values against a date that has already passed (say, the closing date of an accounting period), you might specify that date as VDATE using a field reference (VDATE=>':*block.field*') and compare your segment values against that date.

The default value is the current date (SYSDATE).

DINSERT

Key flexfields only. Use DINSERT to turn dynamic inserts off or on for this form.

The default value is Y (the form can do dynamic inserts).

DISPLAYABLE

Key or range flexfields only. The DISPLAYABLE parameter allows you to display segments that represent specified *flexfield qualifiers* or specified *segment numbers*, where *segment numbers* are the order in which the segments appear in the flexfield window, not the segment number specified in the Key Flexfield Segments form. For example, if you specify that you want to display only segment number 1, your flexfield displays only the first segment that would normally appear in the pop–up window (for the structure you specify in NUM).

The default value for DISPLAYABLE is ALL, which makes your flexfield display all segments. Alternatively, you can specify a *flexfield qualifier name* or a *segment number.*

You can use DISPLAYABLE as a toggle switch by specifying more than one value, separated by \\0 delimiters. For example, if you want your flexfield to display all but the first segment, you would specify:

```
DISPLAYABLE=>'ALL\\01'
```

Note that \\0 separates 1 from ALL.

If you do not display all your segments, but you use default values to fill in your non–displayed segments, you must also have hidden SEGMENT1 through SEGMENTn fields in your form. You need these hidden fields because your flexfield writes the values for all displayed fields to the concatenated values field, but does not write the values for the non–displayed defaulted fields. Since your flexfield normally uses the values in the concatenated values field to update and insert to the database, the default values for the non–displayed fields are not committed. However, if you have the extra hidden fields (similar to a combinations form), your flexfield writes flexfield values to those fields as well as to the concatenated segment values field. The non–displayed values are written only to the hidden fields, but are used to update and insert to the database.

UPDATEABLE INSERTABLE

Key or range flexfields only. The UPDATEABLE / INSERTABLE parameters determine whether your users can update or insert segments that represent specified unique *flexfield qualifiers* or *segment numbers*, where *segment numbers* are the order in which the segments appear in the flexfield window, not the segment number specified in the Key Flexfield Segments form.

The default value for each is ALL, which allows your user to update/insert all segments. Alternatively, you can specify a *flexfield qualifier name* or a *segment number.* You can enter UPDATEABLE=>'' or INSERTABLE=>'' (two single quotes) to prevent your user from updating or inserting values for **any** segments.

You can use these parameters as toggle switches by specifying more than one value, separated by \\0 delimiters. For example, if you want your user to be able to update all but the first segment, you would specify:

```
UPDATEABLE=>'ALL\\01'
```

Note that \\0 separates 1 from ALL.

If you use INSERTABLE=>'' to prevent your user from inserting values for any segments, Shorthand Flexfield Entry is disabled for that form.

TITLE

Specify the *window title* you want to appear at the top of the pop–up window. The default value for a key flexfield is the Structure Name you specify when you set up this flexfield using the Key Flexfield Segments form. For a descriptive flexfield, the default value is the flexfield title you specify when you set up this flexfield using the Descriptive Flexfield Segments form.

REQUIRED

Key or range flexfields only. Specify whether your user can exit the flexfield window without entering segment values.

The default value is Y.

If you specify Y, then your flexfield prevents your user from leaving any required segment (a segment whose value set has Value Required set to Yes) without entering a valid value for that segment. Also, if your user tries to save a row without ever entering the flexfield pop–up window, your flexfield attempts to use default values to fill in any required segments and issues an error message if not all required segments can be filled.

If you specify Y and VALIDATE as FULL, then when your user queries up a row with no associated flexfield (the foreign key flexfield ID column contains NULL), your flexfield issues an error message to warn the user that a NULL ID has been returned for a required flexfield.

If you specify N, your flexfield allows your user to save a row without ever entering the flexfield pop–up window. If you specify N, your user can navigate (without stopping) through a flexfield window without entering or changing any values. However, if a user enters or changes any segment value in the flexfield, the user cannot leave the flexfield window until all required segments contain valid values. If you specify N and a user does not open or enter values in the window, the user can save the row regardless of whether the flexfield has required segments. In this case, your flexfield does not save default values as segment values for the required segments, and it does not issue an error message.

If you specify N and VALIDATE as FULL, then when your user queries up a row with no associated flexfield (the foreign key flexfield ID column contains NULL), your flexfield validates the individual segment values returned by the query. Specify N if you want to query up non–required flexfields without getting an error message.

Note that even if REQUIRED is set to N, a user who starts entering segment values for this flexfield must either fill out the flexfield in full, or abandon the flexfield.

AUTOPICK

Determines whether a list of values window appears when your user enters an invalid segment value. The default value is Y.

COPY

Key flexfields only. Copies a non–null value from *block.field* into the segment representing the specified flexfield *qualifier* or *segment number* before the flexfield window pops up. Alternatively, if you specify ALL, COPY copies a set of non–null, concatenated set of segment values (and their segment separators) that you have in *block.field* into all of your segments. For example, if you have a three–segment flexfield, and your *block.field* contains 001.ABC.05, COPY puts 001 into the first segment, ABC into the second segment, and 05 into the third segment.

The value you COPY into a segment must be a valid value for that segment. The value you COPY overrides any default value you set for your segment(s) using the Key Flexfield Segments form. However, shorthand flexfield entry values override COPY values. COPY does not copy a NULL value over an existing (default) value. However, if the value you copy is not a valid value for that segment, it gives the appearance of overriding a default value with a NULL value: the invalid value overrides the default value, but your flexfield then erases the copied value because it is invalid. You should ensure that the field you copy from contains valid values.

When the flexfield window closes, your flexfield automatically copies the value in the segment representing the specified *flexfield qualifier* or *segment number* into *block.field*. Alternatively, if you specify ALL, your flexfield automatically copies the concatenated values of all your segments into *block.field*.

You can specify one or more COPY parameter values, separated by \\0 delimiters. Later COPY values override earlier COPY values. For example, assume you have a field that holds concatenated flexfield values, called Concatenated_field, and it holds the string 01–ABC–680. You also have a field, Value_field, that holds a single value that you want to copy into your second segment, and it holds the value XYZ. You specify:

```
COPY=>'block.Concatenated_field\\nALL\\0
       block.Value_field\\n2'
```

Note that \\0 separates the different parameter values.

When your user opens the flexfield window, Oracle Application Object Library executes the two COPY parameters in order, and your user sees the values in the window as:

```
01
XYZ
680
```

After the flexfield window closes, your flexfield copies the values back into the two fields as 01–XYZ–680 and XYZ respectively. Note that XYZ overrides ABC in this case.

DERIVED

Key flexfields only. Use DERIVED to get the derived value of segment qualifiers for a combination that a user types in. Use *block.field* to specify the block and field you want your flexfield to load the derived value into. Use *Segment qualifier* to specify the segment qualifier name you want. Note: do not put spaces around \\n, and \\n must be lowercase.

Your flexfield uses the following rules to get the derived qualifier value from the individual segment qualifier values: if the segment qualifier is unique, the derived value is the segment qualifier value; for non–unique segment qualifiers, if any segment's qualifier value = N, then the derived value is N, otherwise, the derived value is Y. The only exception to this rule is for the internal SUMMARY_FLAG segment qualifier; the rule for this is if any segment value is a parent, then the derived value of SUMMARY_FLAG is Y. Your flexfield loads derived values into the combinations table qualifier column that you specify when you define your qualifier.

You can specify one or more groups of DERIVED parameters separated by \\0.

DERIVE_ ALWAYS Key flexfields only. Use with the DERIVED parameter. If you specify Y, the derived values are computed even if the user navigates through the flexfield without changing any values (choosing the same value that is already in a segment does mark the flexfield as having changed).

The default value is N, where the derived values are calculated only if the flexfield is modified.

VRULE Key or range flexfields only. Use VRULE to put extra restrictions on what values a user can enter in a flexfield segment based on the values of segment qualifiers (which are attached to individual segment values). You can specify the name of a *flexfield qualifier* and a *segment qualifier*, whether to Include or Exclude the *validation values*, and the *Message Dictionary application short name* and *message name* for the message your flexfield displays if the user enters an improper value. The delimiter \\n must be lowercase, and you separate the application name from the message name using a semicolon.

For example, suppose you build a form where you want to prevent your users from entering segment values for which detail posting is not allowed into all segments of Oracle General Ledger's Accounting Flexfield.

DETAIL_POSTING_ALLOWED is the segment qualifier, based on the global flexfield qualifier GL_GLOBAL, that you want to use in your rule. You want to exclude all values where the value of DETAIL_POSTING_ALLOWED is N (No). Your message name is "GL Detail Posting Not Allowed", and it corresponds to a message that says "you cannot use values for which detail posting is not allowed." You would specify your rule as:

```
VRULE='GL_GLOBAL\\nDETAIL_POSTING_ALLOWED\\nE
\\nAPPL=\\'SQLGL';
NAME=GL Detail Posting Not Allowed\\nN'
```

When your user enters an excluded value in one of the segments affected by this qualifier, your user gets the message you specify. In addition, the excluded values do not appear in the list of values on your segments. All other values, not being specifically excluded, are included.

You can specify one or more groups of VRULE parameters separated by \\0. Oracle Application Object Library checks multiple VRULE parameters bottom–up relative to the order you list them. You should order your rules carefully so that your user sees the most useful error message first.

VALATT Key flexfields only. VALATT copies the *segment qualifier* value of the segment representing the **unique** *flexfield qualifier* into *block.field* when the flexfield window closes. The delimiter \\n must be lowercase.

USEDBFLDS For a combinations form, specify this parameter only if your combinations table contains both a full set of key flexfield columns (the primary flexfield) *and* a column that is a foreign key reference to another key flexfield (with a different combinations table). You set this parameter to N to keep the foreign key flexfield from using the database segment fields belonging to the primary flexfield (that your combinations form maintains).

For a foreign key form, specify this parameter if your form is based on a table that has foreign key references to two or more flexfields, and if you have non–database SEGMENT1 through N fields on your form (where N is the number of segments in your combinations table). If such fields exist, your flexfield by default will load values into them that correspond to the combination of segment values in the current flexfield. If you set this parameter to N, your flexfield will not load the segment fields for the current flexfield. If you have more than one flexfield on your form, use this parameter to specify which one should use the segment fields (specify Y for one flexfield's routine calls, and specify N for other flexfields' routine calls).

For a descriptive flexfield, specify N for this parameter to prevent the descriptive flexfield from using hidden segment fields (such as ATTRIBUTEn).

The default value is Y.

COLUMN

Key flexfields only. Use COLUMN to display other columns from the combinations table in addition to the current segment columns, where *n* is the display width of the column. You can place the values of the other columns into fields on the current form. The value is automatically copied into the field when the user selects an existing flexfield combination.

For example, to display a description column called SEG_DESC and an error message from E_FLAG with the column headings DESCRIPTION and ERROR FLAG, you could set

```
COLUMN=>'SEG_DESC DESCRIPTION(15),
       E_FLAG \"ERROR_FLAG\"(*)'
```

The (*) sets a dynamic column width, with the size determined by the value selected.

If you wanted to place the description into the field block_1.field_1 and the error message into block_1.field_2, you would set

```
COLUMN=>'SEG_DESC DESCRIPTION(15)
       INTO BLOCK_1.FIELD_1,
       E_FLAG \"ERROR_FLAG\" (*)
       into BLOCK1_FIELD_2'
```

You may only use 32 distinct INTO columns in your COLUMN= clause. Your maximum width for additional columns is 240 characters.

WHERE_ CLAUSE

Key flexfields only. Specify a WHERE clause to restrict which code combinations to display in the list of values window. This argument also prevents a user from entering a combination that does not fit the WHERE clause. This argument should not normally be used for a flexfield on the combinations form, since you would usually want to display all combinations on the combinations form.

Do not specify the word ”WHERE” in this WHERE clause argument. You should use this token with flexfields that do not allow dynamic inserts, either using DINSERTS as N or preventing dynamic inserts at the structure level.

You should not use the WHERE_CLAUSE argument for a flexfield that allows dynamic inserts.

COMBQP_ WHERE

Key flexfields only. The primary use of this argument is to disable the combination list of values for your flexfield on this form. Specify NONE to disable the combination list of values.

Alternatively, you could use this argument to specify any additional WHERE clause to further restrict which code combinations to display in the list of values window. This WHERE clause is appended to your WHERE_CLAUSE argument using an AND expression. It affects only the combination list of values however, and does not affect a combination that a user enters manually.

Do not specify the word ”WHERE” in this WHERE clause argument.

DATA_SET

Key or range flexfields only. Specify the *:block.field* that holds the set identifier for your flexfield. DATA_SET specifies which set of code combinations to use for this flexfield. For each flexfield structure, you can divide code combinations in your combinations table into sets (for example, parts with high prices, medium prices, and low prices).

You can only use DATA_SET if you implement a structure defining column (that is, you *must* specify NUM). The default for DATA_SET is your structure number (as specified in NUM). If you use DATA_SET, your application must maintain a separate table that contains the correspondences between sets and key flexfield structures. For example, your correspondences table could contain values such as:

Structure	Set	Set Description
101	1	Low priced truck parts
101	2	Medium priced truck parts
101	3	High priced truck parts
102	4	Low priced car parts
102	5	High priced car parts
103	6	Low priced motorcycle parts
103	7	High priced motorcycle parts

If you use DATA_SET, your flexfield stores the set number in the structure defining column instead of the structure number. Note that you cannot have duplicate set numbers in your correspondences table, though you can have more than one set number for a given structure number. You must derive DATA_SET and NUM from different :block.fields (or profile options, or ”hardcoded” numbers) since they are distinctly different numbers.

ALLOWNULLS Determines whether NULLs should be allowed into any segment. ALLOWNULLS only overrides the segment definition (Value Required is Yes) for each segment if you specify PARTIAL or NONE for the VALIDATE parameter.

QUERY_ SECURITY Key flexfields only. Determines whether flexfield value security applies to queries as well as inserts and updates. If you specify Y, your users cannot query up existing code combinations that contain restricted values. If you specify N, your users can query and look at code combinations containing restricted values. Users can update the restricted values to non–restricted values, but they cannot enter restricted values or update values to restricted values. The default value is N. This option has no effect unless your users have enabled and defined flexfield value security for your flexfield's value sets.

QBE_IN

Key flexfields only. Controls the type of subquery your flexfield uses to select the desired rows in flexfield query–by–example.

The default value is N.

If you specify N, your flexfield generates a correlated subquery. This query is effectively processed once for each row returned by the main query (generated by the rest of the form), and it uses the code combination ID as a unique index. Choose N if you expect your main query to return a small number of rows and you expect your flexfield query–by–example to return many rows.

If you specify Y, your flexfield generates a non–correlated subquery using the "IN" SQL clause. Your query is processed only once, but returns all the rows in your combinations table that match your flexfield query–by–example criteria. Choose Y when you expect your main query to return many rows and you expect your flexfield query–by–example to return a small number of rows (less than about 100). Such a condition usually corresponds to a small number of rows in the combinations table and many rows in the application table. For example, assume you have a Part Flexfield, where your company handles only a limited number of parts (say, 75), but you have thousands of orders for your parts (and a correspondingly large Orders table). For this case, choosing Y would greatly improve your application performance on flexfield queries–by–example.

LONGLIST

Key flexfields only. Specify Y or N to allow or disallow using LongList with this flexfield. LongList allows users to specify a partial value when querying a flexfield combination using Combination QuickPick.

NO_COMBMSG	Key or range flexfields only. If you wish to display your own message when a user enters an invalid combination, specify the message name here. Otherwise flexfields uses the standard Oracle Applications message.
READ_ONLY	Specify Y to prevent any updating of your flexfield segment values, whether by shorthand alias, copy, or any other method.
AUTO–COMBPICK	Key flexfields only. Determines the behavior of the combination list of values for direct entry flexfields with no dynamic inserts allowed when the user enters a non–existing code combination. If you specify Y, the combination list of values appears if the user enters an incorrect value in a single segment flexfield, or if there are non–constant values (%) or null segments in a multi–segment flexfield. If you specify N, the combination list of values does not appear, and the error message ”This combination does not exist...” is generated. The default value is Y.
LOCK_FLAG	Normally, when a user types a character into a flexfield segment, that action locks the base table of the form. However, in some cases you might want to avoid locking the table; for example, you might have several inquiry forms that use the same base table, and you do not want other users to have to wait until the table is unlocked. The default value is Y. Specify N to turn off the locking behavior, or specify D to lock the table only if the flexfield–related field is a database field.
HELP	New for Release 11. Use the HELP argument to specify a target name for online help specific to this instance of this flexfield. You specify the application short name for the application that owns the help file (not necessarily the same application that owns the flexfield or the form). You also specify the target name in your help file where the help resides. If the corresponding help target is not found, the user may receive an error message. If the HELP argument is not specified, the online help displays generic flexfields help. For example, to show specific help for the Accounting Flexfield from the Oracle General Ledger help file,

you would specify the following:

```
HELP=>'APPL=SQLGL;TARGET=FLEX.GL#'
```

Additional Optional Arguments for Type Flexfields

If you are building a type flexfield, you use these arguments in addition to other optional and required arguments. If you do not specify a particular optional argument, the flexfield routine uses the usual default value for the argument. You may build a type flexfield that contains more than one ”type column” (a ”column” of fields in the flexfield pop–up window that correspond to the actual segment fields). If you do, you can specify your TYPE_ argument values multiple times, using \\0 to separate the values. SCOLUMN can have only one value, however.

TYPE_FIELD

Range (type) flexfields only. Name of the field where you want to put your ”type” flexfield. This is a displayed, non–database form field that contains your concatenated segment type values plus delimiters.

You can include a suffix for all the fields related to your type field. If you include a suffix, such as TYPE1, your flexfield appends that suffix to all field names automatically. If you specify a suffix, you should not include the suffix in any of the type–related field names for your FND_RANGE_FLEX.DEFINE call. Note that if you specify a suffix, your flexfield expects to store each type value in a form field (one type field for each segment), so you should specify a suffix if you use those fields, but you should not specify a suffix if you use only the concatenated fields.

If you specify TYPE_FIELD, you must also specify TYPE_HEADING, TYPE_VALIDATION, and TYPE_SIZES. TYPE_DESCRIPTION and other type arguments are optional.

You can specify more than one type field and suffix. Each field and suffix must be unique so that the different types do not share the same underlying fields and columns. Separate your first field and suffix from your second field and suffix (and so on) using \\0.

TYPE_ DESCRIPTION

Range (type) flexfields only. Description field for your type flexfield. This is a displayed, non–database, non–enterable field that contains concatenated descriptions of your type segment values. If you do not specify this parameter, your form does not display concatenated type segment descriptions. If you specified a suffix for TYPE_FIELD, do not include it for TYPE_DESCRIPTION.

TYPE_DATA_ FIELD

Range (type) flexfields only. Name of the non–displayed form field that contains the concatenated type segment hidden IDs. If you specified a suffix for TYPE_FIELD, do not include it for this argument.

TYPE_ VALIDATION

Range (type) flexfields only. Specify the name of a value set, such as Yes_No, that you want to use for your type column (for all fields in the type column). You also specify Y if the user is required to enter a value for each field in the type column; specify N if values are not required. Finally, specify a single default value for all fields in your type column. This default value appears in each of the type fields when the pop–up window opens. You may use either a hardcoded constant value or a field reference (*:block.field*) for your default value.

If you have more than one type column, specify subsequent groups of values separated by \\0 delimiters.

TYPE_SIZES

Range (type) flexfields only. Specify the maximum display size for the value set your type field uses, as well as the maximum display size for the value description. The value display size must be at least 1 and not larger than the maximum size of the corresponding value set (whose maximum size must not be greater than the size of the underlying database column). The description display size may be 0 or larger.

If you have more than one type column, you specify sizes for each pair of values and descriptions, separated by the \\0 delimiter.

TYPE_HEADING Range (type) flexfields only. Specify a title that you want to appear above the type segments in the flexfield pop–up window.

If you have more than one type column, specify additional headings separated by the \\0 delimiter.

SCOLUMN Range (type) flexfields only. The presence of the SCOLUMN argument indicates that this is a ”single column type flexfield” (a flexfield that uses only SEGMENTn_LOW and one or more type columns, but does not use SEGMENTn_HIGH). Specify a title for the SEGMENTn_LOW fields that you want to display in the flexfield pop–up window. The flexfield still assumes that the _LOW suffix applies to each SEGMENTn field and related concatenated fields, regardless of the title you specify.

Flexfield Definition Examples

Simple Key Flexfield Example

Here is an example of a simple key flexfield definition. This definition provides the default structure (101) of the Accounting Flexfield.

```
FND_KEY_FLEX.DEFINE(
  BLOCK=>’ORDERS’,
  FIELD=>’KFF_CONCATENATED_VALUES’,

  APPL_SHORT_NAME=>’SQLGL’,
  CODE=>’GL#’,
  NUM=>’101’);
```

Key Flexfield Example with Additional Arguments

Here is an example of a more complex key flexfield definition. This definition provides the default structure (101) of the Accounting Flexfield.

```
FND_KEY_FLEX.DEFINE(
  BLOCK=>’ORDERS’,
  FIELD=>’KFF_CONCATENATED_VALUES’,
```

```
APPL_SHORT_NAME=>'SQLGL',
CODE=>'GL#',
NUM=>'101',

DISPLAYABLE=>'ALL'
INSERTABLE=>'ALL'
UPDATEABLE=>'');
```

Key Flexfield Example with Variable Arguments

Here is an example from the Shorthand Aliases form, which overrides several of the arguments and uses *:block.field* references to pass field values to the procedure. Note that this example also provides three fields related to the flexfield (FIELD, DESCRIPTION, and DATA_FIELD):

```
FND_KEY_FLEX.DEFINE(
     BLOCK=>'ALIASES',
     FIELD=>'SEGMENTS',
     DESCRIPTION=>'SEGMENT_DESCRIPTIONS',
     DATA_FIELD=>'CONCATENATED_SEGMENTS',

     APPL_SHORT_NAME=>':FLEX.APPLICATION_SHORT_NAME',
     CODE=>':FLEX.ID_FLEX_CODE',
     NUM=>':FLEX.ID_FLEX_NUM',

     REQUIRED=>'Y',
     USEDBFLDS=>'N',
     VALIDATE=>'PARTIAL',
     ALLOWNULLS=>'Y');
```

In this example you override the default values for the arguments REQUIRED, USEDBFLDS, VALIDATE and ALLOWNULLS.

Descriptive Flexfield Example

Here is an example of a simple descriptive flexfield definition. This definition provides the descriptive flexfield on the Shorthand Aliases form in the Oracle Applications.

```
FND_DESCR_FLEX.DEFINE(
  BLOCK=>'ALIASES',
  FIELD=>'DF',

  APPL_SHORT_NAME=>'FND',
  DESC_FLEX_NAME=>'FND_SHORTHAND_FLEX_ALIASES');
```

Range Flexfield Example

Here is an example of a simple range flexfield definition.

```
FND_RANGE_FLEX.DEFINE(
  BLOCK=>'RANGES',
  FIELD=>'SEGMENTS',
  DESCRIPTION=>'DESCRIPTIONS'

  APPL_SHORT_NAME=>'SQLGL',
  CODE=>'GL#',
  NUM=>'101',

  VALIDATE=>'PARTIAL');
```

Note that the actual form fields corresponding to FIELD and DESCRIPTION are SEGMENTS_LOW, SEGMENTS_HIGH, DESCRIPTIONS_LOW and DESCRIPTIONS_HIGH.

Range with Types Flexfield Example

The following example uses the Accounting Flexfield with two type fields.

```
FND_RANGE_FLEX.DEFINE(
  BLOCK=>'RANGES',
  FIELD=>'SEGMENTS',
  DESCRIPTION=>'DESCRIPTIONS',

  APPL_SHORT_NAME=>'SQLGL',
  CODE=>'GL#',
  NUM=>'101',

  VALIDATE=>'PARTIAL',
  TYPE_FIELD=>'RANGES.SEGMENTS\\n_TYPE1\\0
```

```
     RANGES.SEGMENTS\\n_TYPE2',
TYPE_DATA_FIELD=>'RANGES.TYPE_DATA\\0
     RANGES.TYPE_DATA',
TYPE_DESCRIPTION=>'RANGES.TYPE_DESC\\0
     RANGES.TYPE_DESC',
TYPE_HEADING=>'Type 1\\0Type 2',
TYPE_VALIDATION=>'Yes_No\\nN\\nYes\\0
     Yes_No\\nN\\nNo',
TYPE_SIZES=>'4\\n4\\04\\n4');
```

Single Range Column with Types Flexfield Example

The SCOLUMN argument is used to define a "Single Column Flexfield". If SCOLUMN has a value, instead of having the "Low", "High" and "Type" columns this flexfield will have only the "Low" and "Type" columns. Since the title "Low" is not appropriate here (since we don't have a "High" column), the value passed in through the SCOLUMN argument is used as the column title. The range flexfield still writes to the underlying segments appended with the suffix "_LOW", and assumes that the "_LOW" suffix is appended to the concatenated segments, description and data_field fields.

The same flexfield as above but when only one column is used.

```
FND_RANGE_FLEX.DEFINE(

  BLOCK=>'RANGES',
  FIELD=>'SEGMENTS',
  DESCRIPTION=>'DESCRIPTIONS',

  APPL_SHORT_NAME=>'SQLGL',
  CODE=>'GL#',
  NUM=>'101',

  VALIDATE=>'PARTIAL',
  SCOLUMN=>'Accounting Flexfield',
  TYPE_FIELD=>'RANGES.SEGMENTS\\n_TYPE1\\0
       RANGES.SEGMENTS\\n_TYPE2',
  TYPE_DATA_FIELD=>'RANGES.TYPE_DATA\\0
       RANGES.TYPE_DATA',
  TYPE_DESCRIPTION=>'RANGES.TYPE_DESC\\0
```

```
     RANGES.TYPE_DESC',
TYPE_HEADING=>'Type 1\\0Type 2',
TYPE_VALIDATION=>'Yes_No\\nN\\nYes\\0
     Yes_No\\nN\\nNo',
TYPE_SIZES=>'4\\n4\\04\\n4');
```

Updating Flexfield Definitions

Normally you define a flexfield only once in your form, usually at the form startup event. However, sometimes you need to change this definition later. For example, you may want to make the flexfield non–updatable and non–insertable. Instead of redefining the entire flexfield with UPDATEABLE=>'' and INSERTABLE=>'' and all the other arguments the same as before, you can use the following update procedures to change only the arguments that need to be modified.

You can use the update procedures to control any of the "other optional arguments" that you specify in your flexfield definition procedures. You cannot use these procedures to change arguments such as which fields your flexfield uses, since those arguments essentially identify the flexfield rather than modify it. For example, you may specify new values for the VALIDATE argument, but you may not specify new values for the DESCRIPTION or DATA_FIELD arguments.

Enabling or Disabling a Flexfield

Once a flexfield has been defined in your form, whenever the FND_FLEX.EVENT calls occur at various block or form level triggers, these events apply to all flexfields defined in the block or form. This makes it difficult to handle situations where you want to make FND_FLEX.EVENT calls for some flexfields but not others. For example, you may not want to call VALID for a particular key flexfield in PRE–UPDATE, but want to call it for all other flexfields in the block. Using the update procedures you can enable and disable a flexfield definition so that the FND_FLEX.EVENT calls do not apply to disabled flexfield definitions.

The update procedures provide a special argument, ENABLED, in addition to the optional arguments you can specify. You specify N for this argument to disable the flexfield, and you specify Y to enable the flexfield. You cannot use ENABLED in your normal flexfield definition procedure calls (which automatically enable the flexfield).

Update Key Flexfield Definition Syntax

Use FND_KEY_FLEX.UPDATE_DEFINITION to update the definition for a key flexfield on a foreign key or combinations form. Other than the ENABLED argument, which you can only use for update procedures, the arguments are the same as you use for the flexfield definition procedures (See page 14 – 36).

```
FND_KEY_FLEX.UPDATE_DEFINITION(

  /* Arguments that specify flexfield location and
           thus identify the flexfield */
     BLOCK=>'block_name',
     FIELD=>'concatenated_segments_field_name',

  /* Argument to enable or disable flexfield */
    [ENABLED=>'{Y|N}',]

  /* Other optional parameters */
    [VALIDATE=>'{FOR_INSERT|FULL|PARTIAL|NONE|
              PARTIAL_IF_POSSIBLE}',]
    [VDATE=>'date',]
    [DISPLAYABLE=>'{ALL | flexfield_qualifier |
           segment_number}[\\0{ALL |
           flexfield_qualifier | segment_number}]',]
    [INSERTABLE=>'{ALL | flexfield_qualifier |
           segment_number}[\\0{ALL |
           flexfield_qualifier | segment_number}]',]
    [UPDATEABLE=>'{ALL | flexfield_qualifier |
           segment_number}[\\0{ALL |
           flexfield_qualifier | segment_number}]',]
    [VRULE=>'flexfield qualifier\\n
           segment qualifier\\n
           {I[nclude]|E[xclude]}\\n
           APPL=application_short_name;
           NAME=Message Dictionary message name\\n
           validation value1\\n
           validation value2...
           [\\0flexfield qualifier\\n
           segment qualifier\\n
```

```
        {I[nclude]|E[xclude]}\\n
        APPL=application_short_name;
        NAME=Message Dictionary message name\\n
        validation value1\\n
        validation value2...]',]
 [COPY=>'block.field\\n{ALL | flexfield
        qualifier | segment_number}
        [\\0block.field\\n{ALL | flexfield
        qualifier | segment_number}]',]
 [DERIVED=>'block.field\\nSegment qualifier',]
 [DINSERT=>'{Y|N}',]
 [VALATT=>'block.field\\n
        flexfield qualifier\\n
        segment qualifier',]
 [TITLE =>'Title',]
 [REQUIRED=>'{Y|N}',]
 [AUTOPICK=>'{Y|N}',]
 [USEDBFLDS=>'{Y|N}',]
 [ALLOWNULLS=>'{Y|N}',]
 [DATA_SET=>'set number',]
 [COLUMN=>'{column1(n) | column1 alias(n)
        [, column2(n), ...]}',]
 [WHERE_CLAUSE=>'where clause',]
 [COMBQP_WHERE=>'{Y|N}',]
 [QUERY_SECURITY=>'{Y|N}',]
 [QBE_IN=>'{Y|N}',]
 [READ_ONLY=>'{Y|N}',]
 [LONGLIST=>'{Y|N}',]
 [NO_COMBMSG=>'{Y|N}',]
 [LOCK_FLAG=>'{Y|N}',]
 [AUTOCOMBPICK=>'{Y|N}',]
 [DERIVE_ALWAYS=>'{Y|N}',]
 [HELP=>'APPL=<application_short_name>;
         TARGET=<target_name>']
);
```

Update Range (Type) Flexfield Definition Syntax

Use FND_RANGE_FLEX.UPDATE_DEFINITION for a range flexfield. You use the same procedure for a "type" flexfield (which may also include range flexfield segments) that contains extra fields corresponding to each segment of the related key flexfield.

Other than the ENABLED argument, which you can only use for update procedures, the arguments are the same as you use for the flexfield definition procedures

Flexfield Definition Arguments (See page 14 – 36)

Attention: You should *not* append "_LOW" or "_HIGH" to the FIELD, DESCRIPTION, DATA_FIELD or other values, since this procedure appends them automatically. When you use more than one type column, ensure that all TYPE_ arguments specify type columns in the same order to avoid having argument values applied to the wrong type column.

```
FND_RANGE_FLEX.UPDATE_DEFINITION(

  /* Arguments that specify flexfield location */
     BLOCK=>'block_name',
     FIELD=>'concatenated_segments_field_name',

  /* Argument to enable or disable flexfield */
    [ENABLED=>'{Y|N}',]

  /* Other optional parameters */
    [VALIDATE=>'{PARTIAL|NONE}',]
    [VDATE=>'date',]
    [DISPLAYABLE=>'{ALL | flexfield_qualifier |
           segment_number}[\\0{ALL |
           flexfield_qualifier | segment_number}]',]
    [INSERTABLE=>'{ALL | flexfield_qualifier |
           segment_number}[\\0{ALL |
           flexfield_qualifier | segment_number}]',]
    [UPDATEABLE=>'{ALL | flexfield_qualifier |
           segment_number}[\\0{ALL |
           flexfield_qualifier | segment_number}]',]
```

```
      [VRULE=>’flexfield qualifier\\n
             segment qualifier\\n
             {I[nclude]|E[xclude]}\\n
             APPL=application_short_name;
             NAME=Message Dictionary message name\\n
             validation value1\\n
             validation value2...
             [\\0flexfield qualifier\\n
             segment qualifier\\n
             {I[nclude]|E[xclude]}\\n
             APPL=application_short_name;
             NAME=Message Dictionary message name\\n
             validation value1\\n
             validation value2...]’,]
      [TITLE =>’Title’,]
      [REQUIRED=>’{Y|N}’,]
      [AUTOPICK=>’{Y|N}’,]
      [USEDBFLDS=>’{Y|N}’,]
      [ALLOWNULLS=>’{Y|N}’,]
      [DATA_SET=>’set number’,]
      [READ_ONLY=>’{Y|N}’,]

    /* Parameters specific to type flexfields */
      [TYPE_FIELD=>’block.concatenated_type_values_
             field\\ntype field suffix’,]
      [TYPE_VALIDATION=> ’Value set name\\n
             Required\\nDefault value’,]
      [TYPE_SIZES=>’type_value_display_
             size\\nDescription_display_size’,]
      [TYPE_HEADING=>’type column heading’,]
      [TYPE_DATA_FIELD=>’block.type_data_field’,]
      [TYPE_DESCRIPTION=>’block.type_
             description_field’,]
      [SCOLUMN=>’single column title’]
      [HELP=>’APPL=<application_short_name>;
              TARGET=<target_name>’]
);
```

Attention: TYPE_FIELD, TYPE_DATA_FIELD and TYPE_DESCRIPTION require the `block` construction, unlike other flexfield arguments that specify field names without block names.

Update Descriptive Flexfield Definition Syntax

Use FND_DESCR_FLEX.UPDATE_DEFINITION for a descriptive flexfield. Other than the ENABLED argument, which you can only use for update procedures, the arguments are the same as you use for the flexfield definition procedures (See page 14 – 36).

```
FND_DESCR_FLEX.UPDATE_DEFINITION(
  /* Arguments that specify the flexfield location */
    BLOCK=>'block_name',
    FIELD=>'field_name',

  /* Argument to enable or disable flexfield */
    [ENABLED=>'{Y|N}',]

  /* Other optional parameters  */
    [VDATE=>'date',]
    [TITLE =>'Title',]
    [AUTOPICK=>'{Y|N}',]
    [USEDBFLDS=>'{Y|N}',]
    [READ_ONLY=>'{Y|N}',]
    [LOCK_FLAG=>'{Y|N}',]
    [HELP=>'APPL=<application_short_name>;
            TARGET=<target_name>']
     );
```

Updating Flexfield Definition Example

Suppose you do not want to call VALID for a particular key flexfield in PRE–UPDATE, but want to call it for all other flexfields in the block. Here is an example of disabling and enabling a simple key flexfield definition. This definition provides the default structure (101) of the Accounting Flexfield. You would code your PRE–UPDATE trigger for the block as:

```
FND_KEY_FLEX.UPDATE_DEFINITION(
  BLOCK=>'ORDERS',
  FIELD=>'KFF_CONCATENATED_VALUES',
  ENABLED=>'N');

FND_FLEX.EVENT('PRE-UPDATE');

FND_KEY_FLEX.UPDATE_DEFINITION(
  BLOCK=>'ORDERS',
  FIELD=>'KFF_CONCATENATED_VALUES',
  ENABLED=>'Y');
```

Using Key Flexfields in Find Windows

You can use a key flexfield in your Find window if you wish to restrict your query to certain segment values. Create a concatenated segments field in your Find window as a 2000 character displayed text item. You do not need the individual SEGMENTn fields in your Find window. Define the key flexfield you want on this field using the FND_KEY_FLEX.DEFINE procedure. This can be done at the same point where you define the flexfield in your base block. Do not pass values for the ID, DESCRIPTION and DATA_FIELD arguments. The following arguments should be set to the values specified below:

```
VALIDATE   => 'PARTIAL_IF_POSSIBLE',
REQUIRED   => 'N',
USEDBFLDS  => 'N',
ALLOWNULLS => 'Y'
INSERTABLE => 'ALL', -- Default value
UPDATEABLE => 'ALL', -- Default value
```

☞ **Attention:** You should set DISPLAYABLE to the same value you used in the definition of the flexfield in your base block.

The above definition allows users to choose values for some segments and leave other segments blank.

Follow the steps described for coding Find windows. In the PRE–QUERY trigger of your base block call the procedure FND_FLEX_FIND.QUERY_KFLEX. The arguments to this function are the application short name, the flexfield code, the structure number, the concatenated segment values and the name of the concatenated segments field in your base block. The procedure specification is given below.

```
PROCEDURE query_kflex(appl_short_name VARCHAR2,
                      code            VARCHAR2,
                      num             NUMBER,
                      segments        VARCHAR2,
                      segments_field  VARCHAR2);
```

☞ **Attention:** The call to FND_FLEX.EVENT('PRE–QUERY') must be made *after* the FND_FLEX_FIND.QUERY_KFLEX procedure is called.

Query Find Window Example Using Key Flexfields

The following example shows how the Accounting Flexfield can be used in a Find window.

```
FND_KEY_FLEX.DEFINE(
  BLOCK           => 'MY_BLOCK_QF',
  FIELD           => 'SEGMENTS',
  APPL_SHORT_NAME => 'SQLGL',
  CODE            => 'GL#',
  NUM             => 101,
  VALIDATE        => 'PARTIAL_IF_POSSIBLE',
  REQUIRED        => 'N',
  USEDBFLDS       => 'N',
  ALLOWNULLS      => 'Y');
```

The PRE–QUERY trigger on MY_BLOCK will be:

```
...
   IF (:parameter.G_query_find = 'TRUE') THEN
     ...
     fND_FLEX_FIND.QUERY_KFLEX('SQLGL', 'GL#', 101,
                               :MY_BLOCK_QF.SEGMENTS,
                               'MY_BLOCK.SEGMENTS');
     ...
     :parameter.G_query_find = 'FALSE';
   END IF;
   ...
   FND_FLEX.EVENT('PRE–QUERY');
```

Using Range Flexfields in Query Find Windows

It is often useful to code a range flexfield in your Find window so that users can specify a value range for the flexfield segments instead of a single value. Create two concatenated segments fields (for low and high values) in your Find window as 2000 character displayed text items. The field names should be of the form XXXX_LOW and XXXX_HIGH. You do not need the individual SEGMENTn fields in your Find window. Define the range flexfield you want on this field using the FND_RANGE_FLEX.DEFINE procedure. This can be done at the same point where you define the flexfield in your base block. Do not pass values for the ID, DESCRIPTION and DATA_FIELD arguments. The following arguments to the define call should be set to the values specified below:

```
VALIDATE   => 'NONE',
REQUIRED   => 'N',
USEDBFLDS  => 'N',
ALLOWNULLS => 'Y'
INSERTABLE => 'ALL', –– Default value
UPDATEABLE => 'ALL', –– Default value
```

☞ **Attention:** You should set DISPLAYABLE to the same value you used in the definition of the flexfield in your base block.

The value for the VALIDATE argument can be 'PARTIAL' if you want users to enter valid segment values as the upper and lower limit of the ranges they want to query on.

The above definition will allow users to choose values for some segments and leave other segments blank. They can also leave either the high or the low segment value blank to set either the lower limit or the upper limit on the segment values. They can enter the same value for both the low and high fields to query on a specific segment value.

Follow the steps for coding Find windows. In the PRE–QUERY trigger of you base block call the procedure FND_FLEX_FIND.QUERY_ KFLEX_RANGE. The arguments to this function are the application short name, the flexfield code, the structure number, the concatenated low segment values, the concatenated high segment values and the name of the concatenated segments field in your base block. The procedure specification is given below.

```
  PROCEDURE query_kflex_range(appl_short_name VARCHAR2,
                              code            VARCHAR2,
                              num             NUMBER,
                              low_segments    VARCHAR2,
```

```
high_segments   VARCHAR2,
segments_field  VARCHAR2);
```

☞ **Attention:** The call to FND_FLEX.EVENT('PRE–QUERY') must be made *after* the FND_FLEX_FIND.QUERY_KFLEX_RANGE procedure is called.

Query Find Window Example Using Range Flexfields

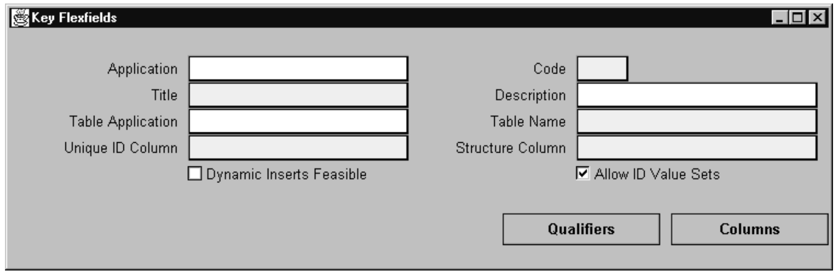
If you choose to use a range flexfield instead of a key flexfield in the preceding example the flexfield definition and the PRE–QUERY trigger will be:

```
FND_RANGE_FLEX.DEFINE(
  BLOCK           => 'MY_BLOCK_QF',
  FIELD           => 'SEGMENTS',
  APPL_SHORT_NAME => 'SQLGL',
  CODE            => 'GL#',
  NUM             => 101,
  VALIDATE        => 'NONE',
  REQUIRED        => 'N',
  USEDBFLDS       => 'N',
  ALLOWNULLS      => 'Y');
```

The PRE–QUERY trigger on MY_BLOCK will be:

```
...
IF (:parameter.G_query_find = 'TRUE') THEN
  ...
  FND_FLEX_FIND.QUERY_KFLEX_RANGE('SQLGL', 'GL#', 101,
                                   :MY_BLOCK_QF.SEGMENTS_LOW,
                                   :MY_BLOCK_QF.SEGMENTS_HIGH,
                                   'MY_BLOCK.SEGMENTS');
  ...
  :parameter.G_query_find = 'FALSE';
END IF;
...
FND_FLEX.EVENT('PRE–QUERY');
```

Key Flexfields Window



Register a key flexfield after defining the flexfield combinations table in the database, and after registering your table using the table registration API.

Attention: Do not modify the registration of any key flexfield supplied with Oracle Applications. Doing so can cause serious application errors. To enable an Oracle Applications key flexfield, define and freeze it using the Key Flexfield Segments window.

Attention: Do not attempt to make a copy of an Oracle Applications key flexfield (using the same table, same title, or same flexfield code), since the duplicates may cause errors in forms that call these flexfields.

If you are using segment qualifiers with your flexfield, you should define the LOV values for your segment types using the Lookups window.

You name your flexfield and associate it with an application and a database table. You also specify which table column you want to use as a unique ID column and which table column you want to use as a structure column.

See:

Table Registration API: page 3 – 11

Key Flexfields Block

Application

An application installer sees this application name when defining your flexfield segments in the Key Flexfield Segments window. Forms and flexfield routines use the combination of application and flexfield name to uniquely identify your flexfield. You use this application name when you use flexfield routines to call your key flexfield from your forms or programs.

Code

You use this short, unique code to invoke a key flexfield from a form trigger.

Title

Enter the user–friendly title of your flexfield. You see this title whenever you choose this flexfield in a flexfield window.

Table Application

Enter the name of the application with which your flexfield combinations table is registered.

Table Name

Enter the name of your combinations table. Your combinations table must already exist in the database, and it must have the appropriate flexfield columns.

You must register your combinations table with Oracle Applications before you can use it in this field.

Unique ID Column

Enter the name of the column in your combinations table that contains the unique ID for this flexfield. Other tables which reference this flexfield should use this column as a foreign key.

Structure Column

Enter the name of the column in your combinations table that your flexfield can use to differentiate among flexfield structures. If you enter a column in this field you must also use the NUM= parameter in all of the flexfield calls in your forms.

Dynamic Inserts Feasible

Indicate whether dynamic inserts are feasible for this key flexfield. Dynamic inserts are feasible only if your combinations table contains no mandatory, non–flexfield columns.

Dynamic inserts cannot be feasible if your application requires special validation of new segment combinations.

Allow ID Value Sets

Indicate whether to allow values sets that use a hidden ID in your flexfield.

Detail Buttons

Qualifiers Choose this button to open the Qualifies window where you define flexfield and segment qualifiers.

Columns Choose this button to open the Columns window where you enable the columns to use with your flexfield segments

Flexfield Qualifiers Window

Define flexfield and segment qualifiers. A flexfield qualifier applies to specific segments your user define, and a segment qualifies applies to specific values in your flexfield segments. You must define a flexfield qualifier before you can define segment qualifiers.

Name

Use this unique name to reference key flexfield structure information.

Prompt

When you set up your key segments this prompt asks you for the qualifiers information for your key flexfield. Since flexfield qualifiers use check boxes in the Key Flexfield Segments form, you should specify your prompt so it makes sense as the prompt of a check box.

Global

Global flexfield qualifiers apply to all segments, and provide a convenient mechanism for assigning a group of segment qualifiers to all segments.

Required

Required flexfield qualifiers must apply to at least one but possibly more segments.

Unique

Unique flexfield qualifiers apply to one segment only.

Segment Qualifiers

A segment qualifier applies to specific values your end user defines using the Segment Values window. Segment qualifiers expect QuickCodes values (lookups).

Name

Use this unique name to reference key segment value information in flexfield routine calls and your application logic.

Prompt

The Segment Values window displays this prompt to ask you for information about each segment value when you define key segment values. Since segment qualifiers receive QuickCode values in the Segment Values window, you should specify your prompt so it makes sense to your end user.

Derived Column

Enter the name of a database column in your combinations table that holds the derived value of this segment qualifier. Flexfields automatically derives a value for your segment qualifier into this column whenever your end user enters a new valid combination.

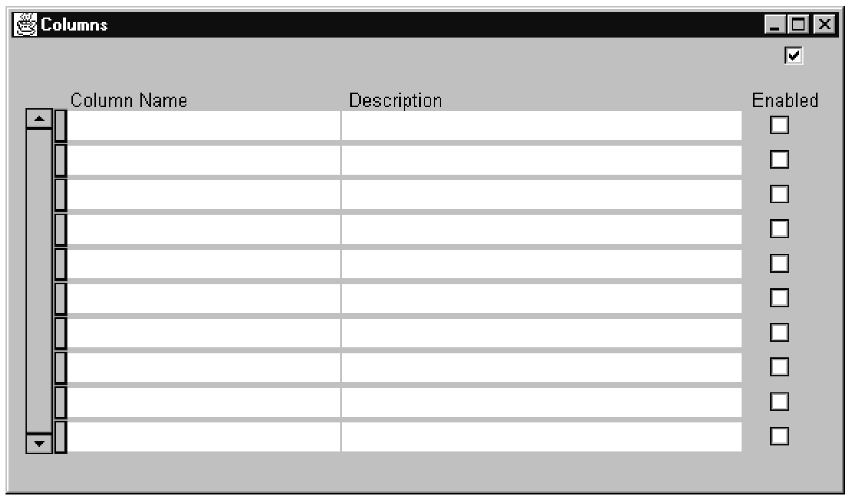
QuickCode Type

Enter a Special QuickCode type for this segment qualifier. A Special QuickCode type defines the group of values you wish to allow for this segment qualifier. For example, if you have a segment qualifier called ”Account Type” you might want a Special QuickCode type called ”ACCOUNT_TYPE” that has several codes and meanings. You define Special QuickCode values using the Lookups form.

Default Value

A default value must be one of the defined Special QuickCode values for the Special QuickCode type you choose in the QuickCode Type field.

Columns Window



Specify the columns your key flexfield can use as segment columns. This window automatically queries up most of the columns you registered when you registered your table. If you have recently added columns to your table, you should reregister your table to ensure you see all your columns. The table columns you specify as your unique ID column or your structure column in the Key Flexfield zone do not appear.

If your table contains columns with names of the form SEGMENT1, SEGMENT2, SEGMENT3, and so on, those columns are automatically Enabled for your flexfield. You must enable any other column you want to use for your segment columns by changing the value of the Enabled check box.

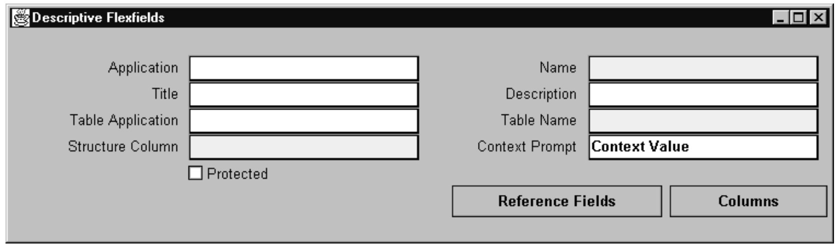
For example, if you have more than one key flexfield, your second key flexfield may have different segment column names such as TAX1, TAX2, TAX3 and TAX4. In this case, you would enable TAX1, TAX2, TAX3 and TAX4 and disable SEGMENT1, SEGMENT2, SEGMENT3, and so on for your second key flexfield.

Warning: If you are redefining the Accounting Flexfield for Oracle General Ledger (this key flexfield is used by most of the Oracle Applications products), you must not use any columns other than those named SEGMENT1 through SEGMENT30. Since the names of these columns are embedded in the Oracle Applications products, using other columns may adversely affect your application features such as summarization.

Enabled

Indicate whether this column can be used as a segment column for your key flexfield. If you enable a column as a segment column for a key flexfield, you should not enable the same column for another key flexfield that uses the same table.

Descriptive Flexfields Window



Register your flexfield after adding the descriptive flexfield columns to your table and registering your table. You must register a descriptive flexfield before you can use it in an application.

Use this window to provide information about your descriptive flexfield. Give your flexfield a name and associate it with an application and a database table. Also specify which table column you want to use as a structure column.

Descriptive Flexfields Block

Forms and flexfield routines use the combination of application name and flexfield name to uniquely identify your flexfield.

Application

An application installer sees this application name when defining your descriptive flexfield in the Descriptive Flexfield Segments window. Use this application name when you use flexfield routines to call your descriptive flexfield from your forms or programs.

Descriptive Flexfield Segments Window
Oracle Applications Flexfields Guide

Name

Use this name when you use flexfield routines to call your descriptive flexfield from your forms or programs.

Title

Flexfields displays this unique title at the top of the flexfield window when your users enter your descriptive flexfield. An application installer can modify this title using the Descriptive Flexfield Segments window.

Descriptive Flexfield Segments Window
Oracle Applications Flexfields Guide

Table Name

Enter the name of the table that contains your descriptive flexfield columns. Your table must already exist in the database, and it should already have columns for your descriptive flexfield segments, as well as a structure column. These segment columns are usually called ATTRIBUTE1, ATTRIBUTE2, ..., ATTRIBUTEn.

You must register your table with Oracle Applications before you can use it in this field.

Structure Column

Enter the name of the column, such as ATTRIBUTE_CATEGORY, in your table that your flexfield uses to differentiate among descriptive flexfield structures. Your descriptive flexfield uses this column to let your users see different descriptive flexfield structures based on data supplied by the form or the user. You must have a structure column even if you only intend to use one descriptive flexfield structure.

Context Prompt

Enter a default context field prompt that asks your user which descriptive flexfield structure to display. Depending upon how your application installer defines your descriptive flexfield, your user may or may not see a context field as part of the descriptive flexfield pop–up window. Descriptive flexfield windows display this context field prompt if the installer allows the end user to override the default context field value.

If your application installer defines it, the context field appears to the user as if it were simply another flexfield segment (with the prompt you specify here). Your user enters a value in the context field, and other descriptive flexfield segments pop up based on that value. The installer can modify the context field prompt using the Descriptive Flexfield Segments window.

Protected

In some cases, you may want to create a descriptive flexfield that cannot be inadvertently changed by an installer or user. This type of flexfield is called a protected descriptive flexfield. You build a protected descriptive flexfield the same way you build a normal descriptive flexfield. The main difference is that you check the Protected check box after defining your segment structures. Once a descriptive flexfield is protected, you cannot query or change its definition using the Descriptive Flexfield Segments window. You should define your descriptive flexfield segments before you change the Protected check box.

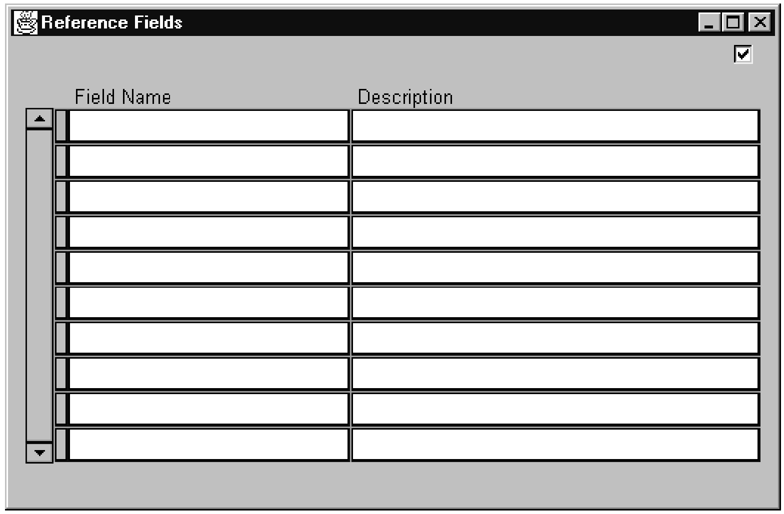
Descriptive Flexfield Segments Window
Oracle Applications Flexfields Guide

Detail Buttons

Reference Fields Choose this button to open the Reference Fields window where you select possible reference fields for your descriptive flexfield.

Columns Choose this button to open the Columns window where you enable table columns for your descriptive flexfield segments.

Reference Fields Window



Use this window to specify any form fields that might serve as descriptive flexfield reference fields. Your flexfield can use values in one of these fields (context field values) to determine which flexfield structure to display.

An installer using the Descriptive Flexfield Segments window can choose to use one of these window fields to obtain the context field value for your descriptive flexfield.

You should specify all form fields that contain information an installer might use to obtain a context field value. For example, the descriptive flexfield in an application form may be used to capture different information based on which country is specified in a field on that form, or based on a name specified in another field. In this case, both the country field and the name field should be listed as potential reference fields, and the installer can decide which reference field to use (or neither).

An installer typically defines different structures of descriptive flexfield segments for each value that the reference field would contain. Though the installer does not necessarily define a structure for *all* the values the reference field could contain, a field that has thousands of possible values may not be a good reference field. In general, you should only list fields that will contain a relatively short, static list of possible values, such as a field that offers a list of countries.

A good reference field usually has a defined list of values. You should not list fields that could contain an infinite number of unique values, such as a PO Number field. Often the business uses of the particular form dictate which fields, if any, are acceptable reference fields.

You may specify additional fields to be available as reference fields even after you have registered your flexfield.

Field Name

Enter the name of a reference field your flexfield can use to obtain context field values.

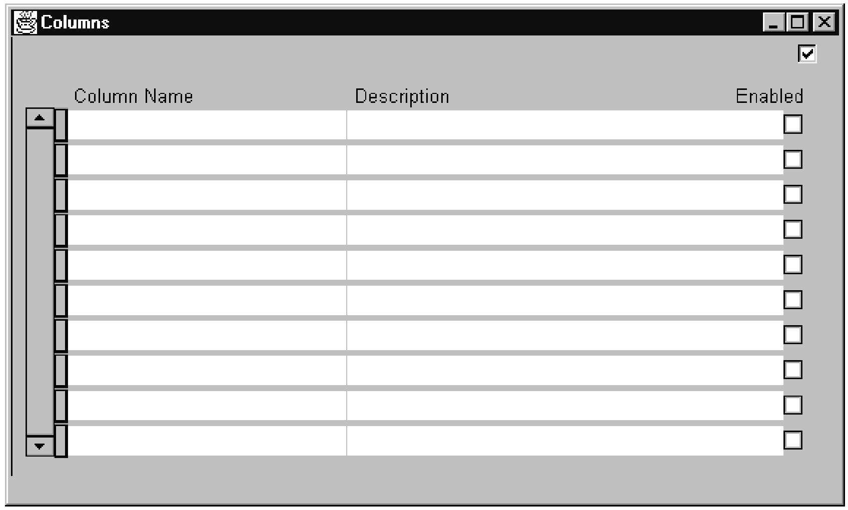
Enter the actual (hidden) Oracle Forms name of the field, rather than the boilerplate name of the field (the field prompt). Do not include the block name. The Descriptive Flexfield Segments window displays this field name in a list an installer sees when defining descriptive flexfield segments.

This field must exist in the same block as the descriptive flexfield. In addition, if you call your descriptive flexfield from several different forms or zones, the same field must exist in all form blocks that contain this descriptive flexfield.

Description

Since the actual Oracle Forms field names often do not match the boilerplate prompts for your fields, we recommend that you enter the visible field prompt as part of your description of your context reference field so an installer can easily tell which field to define as the reference field for your descriptive flexfield.

Columns Window



Use this window to specify the columns your descriptive flexfield can use as segment columns. When you navigate into this block, this window automatically queries up most of the columns you registered when you registered your table.

If you have recently added columns to your table, you should reregister your table to ensure you see all your columns in this block. This window does not display the table column you specify as your structure column in the Descriptive Flexfields block.

If your table contains columns with names ATTRIBUTE1, ATTRIBUTE 2, ATTRIBUTE3, and so on, those columns are automatically Enabled. To use other columns for your flexfield segments, you must set explicitly enable them.

For example, if you have more than one descriptive flexfield, your second descriptive flexfield may be a protected descriptive flexfield with different segment column names such as TAX1, TAX2, TAX3 and TAX4. In this case, you would enable TAX1, TAX2, TAX3 and TAX4 and disable ATTRIBUTE1, ATTRIBUTE 2, ATTRIBUTE3, and so on for your protected descriptive flexfield.

Enabled

Indicate whether this column can be used as a segment column for your descriptive flexfield. If you enable a column as a segment column for a descriptive flexfield, you should not enable the same column for another descriptive flexfield that uses the same table.

Any columns you enable here appear when an installer defines segments using the Descriptive Flexfield Segments window.

CHAPTER

15

Overview of Concurrent Processing

This chapter provides an overview of Concurrent Processing in Oracle Application Object Library. It includes a summary of Concurrent Processing features, definitions of key concepts and an outline of the steps necessary to add Concurrent Processing to your application. The implementation section provides you with the technical details of adding Concurrent Processing to your application.

- Overview of Concurrent Processing
- Overview of Designing Concurrent Programs
- Overview of Implementing Concurrent Programs

Overview of Concurrent Processing

In Oracle Applications, concurrent processing simultaneously executes programs running in the background with online operations to fully utilize your hardware capacity. You can write a program (called a ”concurrent program”) that runs as a concurrent process. Typically, you create concurrent programs for long–running, data–intensive tasks, such as posting a journal or generating a report.

For more information on concurrent processing from a user's viewpoint, see the *Oracle Applications User's Guide* and the *Oracle Applications System Administrator's Guide.*

You can use PL/SQL to create a stored procedure concurrent program. In addition, any PL/SQL procedure you develop—whether it runs on the client, or on the server as a stored procedure or a database trigger—can submit a concurrent request to run a concurrent program.

Concurrent Processing APIs for PL/SQL Procedures (See page 20 – 2)

Basic Application Development Needs

Oracle Application Object Library Concurrent Processing provides you with all the features you need to satisfy the following application development needs:

- Ensure consistent response time, regardless of the variability of data your applications process
- Allow end users to keep working at their terminals while long–running processes run concurrently
- Allow you to fully use all the capacity of your hardware by executing many application tasks at once

Major Features

Online Requests

You and your end users can submit requests from forms to start any concurrent program. Once your request has been submitted, the concurrent managers immediately take over and complete the task with no further online involvement.

Automatic Scheduling

Oracle Application Object Library automatically schedules requests based on when they were submitted, their priority, and their compatibility with programs that are already running. As a developer, you can define which programs in your application should not run simultaneously. A request for a program that is incompatible with a currently running program does not start until the incompatible program completes.

Concurrent Processing Options

You and your end users can control certain runtime options for each of your concurrent requests. Profile options allow you to determine printing decisions such as which printer to use and how many copies to print.

Online Request Review

Your end users can review the progress of their concurrent requests online. Once the request completes, they can view the report output and log files from their concurrent requests. They can see the status of a concurrent request without printing out the entire report by selecting Requests from the default help menu.

Concurrent Managers

Concurrent managers are components of Concurrent Processing that monitor and run time–consuming, non–interactive tasks without tying up your terminal. Whenever you or your users submit a request to run a task, a concurrent manager processes that request and does the work in the background, giving you the ability to run multiple tasks simultaneously.

Oracle Application Object Library predefines the Internal Concurrent manager, which functions as the ”boss” of all other managers. The Internal Concurrent Manager starts up, verifies the status of, resets, and shuts down the individual managers. It also enforces program incompatibility rules by comparing program definitions for requested programs with those programs already running in an Oracle username designated as a logical database.

After installation, the system administrator can create and tailor the concurrent managers as needed. The system administrator chooses the number of requests that each concurrent manager can process simultaneously and what types of requests it can process. Concurrent managers provide total control over the job mix and throughput in your application.

Simultaneous Queuing

Simultaneous Queuing lets requests wait in many queues at once to ensure that the first available concurrent manager starts your request. Use Oracle System Administration to set up Concurrent Processing so that requests can be run by more than one concurrent manager. When the first available concurrent manager starts a request, it automatically removes the request from all the other queues.

Multiple Concurrent Program For Each Executable

Concurrent program executables allow you to use the same execution file for multiple concurrent programs. To create specialized versions of a concurrent program, either define a new concurrent program using the same executable, or copy the concurrent program. You can specialize a concurrent program by required printers, specialization rules, or application name so that the concurrent programs run using the same execution file but with different parameters.

Unified C API

The Unified C API **afprcp()** function allows you to write C or Pro*C programs using a standard format and use your programs with either the spawned or immediate execution method. The same execution file can run both as a spawned process or as a subroutine of the concurrent manager.

Definitions

Concurrent Program

A concurrent program is an instance of an execution file, along with parameter definitions and incompatibilities. Concurrent programs use concurrent program executables to locate the correct execution file. Several concurrent programs may use the same execution file to perform their specific tasks, each having different parameter defaults and incompatibilities.

Concurrent Program Executable

A concurrent program executable links an execution file or and the method used to execute it with a defined concurrent program. Under Concurrent Processing, an execution method may be a program written in a standard language, a reporting tool, or an operating system language.

An execution method can be a PL/SQL Stored Procedure, an Oracle Tool such as Oracle Reports or SQL*Plus, a spawned process, or an operating system host language.

Concurrent Program Execution File

A concurrent program execution file is an operating system file or database stored procedure which contains your application logic and can be executed by either invoking it directly on the command line or by invoking a program which acts upon it. For example, you run a Pro*C program by invoking it on the command line. You run a SQL script by running SQL*Plus and passing the name of the SQL script without the **.**sql extension.

Concurrent Program Subroutine

A concurrent program subroutine is a Pro*C routine which contains your application logic and is linked in with the Concurrent Manager code.

Execution Method

The execution method identifies the concurrent program executable type and the method Oracle Application Object Library uses to execute it.

An execution method can be a PL/SQL Stored Procedure, an Oracle Tool such as Oracle Reports or SQL*Plus, a spawned process, or an operating system host language.

Oracle Tool Concurrent Program

A concurrent program written in Oracle Reports, PL/SQL, SQL*Loader, or SQL*Plus.

☞ **Attention:** Starting with Release 11, SQL*Report (RPT) is no longer supported.

Spawned Concurrent Program

A concurrent program that runs in a separate process (on most operating systems) than that of the concurrent manager that starts it. You write spawned concurrent programs as C or Pro*C stand–alone executable files. On some operating systems, you can also write spawned concurrent programs in your operating system language.

Spawned concurrent programs are the recommended execution method for new C or Pro*C execution files.

Immediate Concurrent Program

A concurrent program that runs in the same process as the concurrent manager that starts it. You write immediate concurrent programs as C or Pro*C subroutines and link them in with a concurrent manager.

Attention: The immediate concurrent program functionality is provided for backward compatibility only. You should not be creating new immediate concurrent programs.

Program Library

A program library is a set of linked immediate concurrent programs that are assigned to concurrent managers. A concurrent manager can run any spawned or Oracle Tool concurrent programs, but only immediate concurrent programs in its own program library. Use Oracle System Administration to further restrict what concurrent programs a concurrent manager can run when defining a concurrent manager with specialization rules.

You register your program library with Oracle Application Object Library. List the short names of the immediate concurrent programs in your program library. Then, use Oracle System Administration to assign program libraries to concurrent managers.

You can include an immediate concurrent program in different libraries. Each concurrent manager that runs immediate concurrent programs must have a program library, although several managers can share the same library.

Request Type

A request type groups similar concurrent programs to save time in defining and maintaining concurrent managers.

Grouping Programs As Request Types
Oracle Applications System Administrator's Guide

Parent Request

A parent request is a concurrent request that submits another concurrent request. In the case of Standard Request Submission, a report set is a parent. When you submit a report set, the report set submits the reports or programs that you have included in the report set. A parent request may be sequential or parallel which determines whether the requests it submits run one at a time or all at once.

Child Request (Sub–request)

A child request is a concurrent request submitted by another concurrent request. When you submit a concurrent request to run a report set, the report set submits the reports in the set for concurrent processing. The requests submitted by the report set are child requests.

Logical Database

A logical database is a set of logically related data stored in one or more ORACLE IDs. Concurrent managers use logical databases to determine the scope of concurrent program compatibilities.

When you define a concurrent program, you specify what programs are incompatible with this program and cannot run together with this program in the same logical database. A user in a logical database submits a concurrent request to run a concurrent program. If a concurrent manager that can process your request finds that there are no incompatible programs currently running in the user's logical database, then the concurrent manager processes the concurrent request. Concurrent managers use logical databases to ensure that incompatible programs do not run together.

Overview of Designing Concurrent Programs

Any program or any portion of a form that can be separately constructed to be non–interactive could potentially be run as a concurrent process. Generally, you should consider making any application function that could tie up your end user's terminal into a concurrent program. Among the functions that best take advantage of concurrent processing are reports and functions that perform many database operations.

You should design your concurrent program to use the features and specifications of concurrent processing most efficiently. Your program should expect values to be passed as concurrent program parameters, and should handle failure gracefully, allowing the concurrent manager to restart your program without creating data integrity problems. If you want to generate error messages and output, you should instruct your program to write to separate log and output files. This makes diagnosing any problems much easier. If you are writing programs under a custom application, include that custom application in the datagroups for the responsibilities with access to your program. Finally, if your program access custom tables you build, implement your tables to accept Who column values.

Users submit concurrent requests using Standard Request Submission or from a form by performing an action that initiates a trigger step in the form. If you want your users to submit your program through Standard Request Submission, you must check the "Use in SRS" check box" and register your program parameters when you define your concurrent program. To let your users submit a concurrent request for your program from a form, you call an Oracle Application Object Library user exit from a trigger step and specify the name of your program and its arguments. Typically, your program takes arguments from fields on your form. You can also submit a concurrent request from within a program. Oracle Application Object Library provides a standard interface between the calling form or program and your concurrent program that is independent of the operating system you are using.

When a user submits a concurrent request, the request waits in the queue of each concurrent manager defined to be able to run the user's concurrent request. Use Oracle System Administration to set the priority of requests that a user submits and change the priority of individual requests. The request's priority affects the request's position in the queues.

The first available concurrent manager that can process the user's request looks at predefined or system administrator–defined data groups in Oracle Application Object Library tables. The data group assigned the user's responsibility contains a list of application names and their corresponding ORACLE IDs. The concurrent manager automatically uses the ORACLE ID associated with the concurrent program's application to run the program. Use Oracle System Administration to set up data groups for each responsibility.

Define Data Group
Oracle Applications System Administrator's Guide

When you write and define your concurrent program, you define what programs are incompatible to run with your program in the same logical database.

Implementing Standard Request Submission (See page 21 – 8)

If no concurrent manager is currently running an incompatible program in the same logical database as the user's concurrent request, the concurrent manager removes the user's concurrent request from any other queues and runs your concurrent program. Your concurrent program writes to a report output file and a log file. The concurrent manager automatically prints the report output if you defined your program to print output. The system administrator or user can specify printing information such as the printer destination and the number of copies using user profiles, in this case "Printer" and "Concurrent:Report Copies". If the program is submitted using Standard Request Submission, the user can specify printing and submission information at runtime.

You can write different types of concurrent programs: Oracle Tool programs written in SQL*Plus, PL/SQL, SQL*Loader, or Oracle Reports; programs written in C or Pro*C, or host language programs.

Attention: Starting with Release 11, SQL*Report (RPT) is no longer supported.

Creating Concurrent Programs

The basic process of creating a concurrent program is the same regardless of the execution method. The steps you take include:

- Write your program execution file and place it in the appropriate location for your operating system.
- Define the concurrent program executable with Oracle Application Object Library to specify an execution file and an execution method for your program.

- Define the concurrent program with Oracle Application Object Library, define program parameters if necessary, and define any incompatibilities with other concurrent programs
- Request your program from the Run Reports form, from a trigger step in an application form, or from an Pro*C concurrent program.

Define Concurrent Program Executable (See page 16 – 6)
Define Concurrent Program (See page 16 – 10)

Implementing Concurrent Processing (See page 15 – 21)
Implementing Standard Request Submission (See page 21 – 8)

Concurrent Processing
(Oracle Applications Installation Manual)

Concurrent Program Parameters

The concurrent manager processes up to 100 arguments for a concurrent program. Each argument can be no longer than 240 characters. For spawned Pro*C concurrent programs, the concurrent manager can process arguments that are longer than 240 characters if you use extended syntax to submit your program. When using extended syntax, the concurrent manager can process a total argument string (the length of all your arguments combined) of up to 24,000 characters.

Handling System Failures

If a concurrent manager terminates abnormally while it is processing requests (for example, if your system crashes), it remembers the requests that are running at the time of the failure. When you restart the concurrent managers, they automatically restart those requests. To ensure that your concurrent program handles system failures properly, you should design your program so that a concurrent manager can restart it from the beginning without your program creating data inconsistencies or receiving errors such as ”duplicate key in index” errors.

The simplest way to do this is to avoid committing transactions until the last step in your program. If this is not feasible due to the amount of data your program could potentially process, you have several alternatives.

You can commit intermediate transactions to temporary tables, and then perform one final transaction at the end of your program to transfer data from your temporary tables to your main tables. When your program starts, it should delete any data from the temporary tables that might have resulted from a previous system crash.

Another alternative is to include a special status column in your tables that you update in your program to indicate the rows that are "being processed." You can then set the column to "done" in the last transaction in your program. You should ensure that your other application programs ignore rows with the value "being processed" in the status column.

Writing Concurrent Programs

You can write concurrent programs using a variety of execution methods. For example, you can use Oracle Tools, programming languages such as Pro*C, or PL/SQL procedures to create the execution files that your concurrent programs invoke.

This section concentrates on using PL/SQL procedures as execution files and on calling your concurrent programs from PL/SQL procedures.

For detailed information on writing concurrent programs using other execution methods, see the following chapters:

Coding Oracle Reports Concurrent Programs (See page 18 – 2)
Coding C or Pro*C Concurrent Programs (See page 21 – 8)
Coding Oracle Tools Concurrent Programs (See page 17 – 2)

C and Pro*C Concurrent Programs Implementation

Spawned and immediate concurrent programs are programs written in C or Pro*C. On some operating systems, you can write your spawned concurrent programs in your operating system command language.

While spawned concurrent programs run in an independent spawned process, immediate programs run as a subroutine of the concurrent manager's process. If you use the Unified C API **afprcp()**, you can use the same execution file with both the spawned and immediate execution methods.

If you use the immediate execution method, you must complete extra steps before submitting your concurrent request.

- Create or modify a program library that includes your immediate concurrent programs.
- Rebuild your program library with Oracle Application Object Library and then link it.
- Assign new program libraries to concurrent managers using Oracle System Administration.

Register Concurrent Program Library (See page 16 – 23)

Define Concurrent Manager
(Oracle Applications System Administrator's Guide)

Concurrent Processing
(Oracle Applications Installation Manual)

Stored Procedure Concurrent Programs

You can implement your concurrent program as a stored procedure. Beginning with Release 11, file I/O operations are supported for stored procedures.

A benefit of developing your concurrent program as a stored procedure is that it runs as part of the concurrent manager's database connection and does not cause an additional process to be spawned, as do other concurrent processing execution methods. Therefore, the "Stored Procedure" execution method is appropriate for frequently–executed concurrent programs (including those you develop to replace immediate concurrent programs from prior releases of Oracle Applications).

Stored procedure concurrent programs accept input parameters only, and are submitted with the FND_REQUEST package. Following is an example specification of a PL/SQL procedure you could create to run as a concurrent program:

```
REM /* Beginning of SQL Script */
REM
CREATE PROCEDURE FND60.SAMPLE_PROC (ERRBUF    OUT VARCHAR2,
                                    RETCODE   OUT VARCHAR2,
                                    ARGUMENT1 IN VARCHAR2,
                                    ARGUMENT2 IN VARCHAR2,
                                    ARGUMENT3 IN VARCHAR2,
                                            .
                                            .
```

```
                .
                .
ARGUMENT100 IN VARCHAR2,
)
```

Your stored procedure concurrent program is restricted to 100 parameters in addition to the first two parameters, which are required and must be specified exactly as indicated in the example above. (You must take these two parameters into account when you create your stored procedure.) Use *errbuf* to return any error messages, and *retcode* to return completion status. The parameter *retcode* returns 0 for success, 1 for success with warnings, and 2 for error. After your concurrent program runs, the concurrent manager writes the contents of both *errbuf* and *retcode* to the log file associated with your concurrent request.

Attention: You should restart your concurrent managers whenever you create or reinstall a stored procedure concurrent program.

PL/SQL Stored Procedures (See page 17 – 3)

Testing Concurrent Programs

The easiest way to test your concurrent program is to submit the program for concurrent processing using the CONCSUB utility. You also have the option to submit the request from the Submit Request form if you are developing your concurrent program for Standard Request Submission. Another way to test your program is to use the form that submits it. Monitor the progress of the request until it completes, then check its completion message and output. If your process completes abnormally, the log file can give you the information you need to take corrective action.

From the operating system, use CONCSUB to submit a concurrent program. By using the WAIT token, the utility checks the request status every 60 seconds and returns you to the operating system prompt upon completion of the request. Your concurrent manager does not abort, shut down, or start up until the concurrent request completes.

If your concurrent program is compatible with itself, you can check it for data integrity and deadlocks by submitting it many times so that it runs concurrently with itself.

Implementing Concurrent Processing (See page 15 – 21)
Implementing Standard Request Submission (See page 21 – 8)

Submitting Concurrent Programs, (CONCSUB)
Oracle Applications Installation Manual

Submitting a Concurrent Request

Your PL/SQL procedures can submit a request to run a program as a concurrent process by calling FND_REQUEST.SUBMIT_REQUEST**.** Before submitting a request, your procedure can optionally call three functions to set concurrent request attributes that determine printing and resubmission behavior:

- FND_REQUEST.SET_OPTIONS
- FND_REQUEST.SET_REPEAT_OPTIONS
- FND_REQUEST.SET_PRINT_OPTIONS

If any of these functions should fail, all setup function parameters are reset to their default values.

In addition, before you call FND_REQUEST.SUBMIT_REQUEST from a database trigger, you must call FND_REQUEST.SET_MODE.

When you call FND_REQUEST.SUBMIT_REQUEST**,** you pass any arguments required by the concurrent program you are requesting. FND_REQUEST.SUBMIT_REQUEST returns the ID of the submitted request if successful, and 0 otherwise.

Upon completion of the FND_REQUEST.SUBMIT_REQUEST function, all the setup function parameters are reset to their default values. It is up to the caller to perform a commit to complete the request submission.

The FND_REQUEST functions are fully described in the Concurrent Processing APIs for PL/SQL Procedures section of this chapter.

Concurrent Processing APIs for PL/SQL Procedures (See page 20 – 2)

Concurrent requests do not submit until they are committed. It is sometimes desirable to immediately commit the requests, bet be aware that there is no way to commit the request without committing all other changes in the form. Do not attempt to commit just the server side, because this releases any locks the user has. To avoid getting a ”no changes to commit” message when the user doesn’t have any changes (check SYSTEM.FORM_STATUS), use the APP_FORM.QUIET_COMMIT routine.

Checking Request Status

Your PL/SQL procedure can check the status of a concurrent request by calling FND_CONCURRENT.GET_REQUEST_STATUS**.**

FND_CONCURRENT.GET_REQUEST_STATUS returns the current status of a concurrent request. If the request has already completed, FND_CONCURRENT.GET_REQUEST_STATUS also returns the completion message associated with the request.

The FND_CONCURRENT.WAIT_FOR_REQUEST function waits for request completion, and then returns the request’s phase/status and completion message to the caller. This function sleeps for a developer–specified interval between checks for request completion.

The FND_CONCURRENT functions are fully described in the Concurrent Processing APIs for PL/SQL Procedures section of this chapter.

Concurrent Processing APIs for PL/SQL Procedures (See page 20 – 2)

Submitting Concurrent Programs on the Client

Oracle Application Object Library for Windows comes with a user interface program which can be used to start and view the results of concurrent programs. The interface for ”Start Concurrent Programs” is modelled on the Run dialog in the Program Manager.

The dialog contains fields for the path to a concurrent program, a database connect string and optional arguments. There is also a combo box which lists concurrent programs which have been installed. To use the program:

Step 1 Select a concurrent program to run, either by typing a path into the Path field, using the Browse button to select a program, or by selecting one of the concurrent programs listed in the Program combo box.

Step 2 Enter a valid database connect string including username and password in the Connect String field.

Step 3 Click on the Run button. A "Working..." message should appear in the bottom left corner of the dialog. When the program finishes, a Done message will appear. At this point you may view the log and output files (if any) for the concurrent program by pressing the View Log or View Output buttons.

Step 4 If you type an invalid connect string, an alert will appear saying ABNORMAL PROGRAM TERMINATION. Click on Close, fix the connect string and try again.

☞ **Attention:** The program for Start Concurrent Program (**startcp.exe**) and the associated program item are installed only if you install the development Application Object Library.

Using Concurrent Processing

You can construct your application so that your end user is unaware of concurrent processing. Even after a user or form submits a concurrent request, your end user can keep working at the terminal without interruption. However, your end user can modify a request's concurrent processing options up until it starts running. For example, your user can change which printer prints a report.

Your end user can monitor the progress of a concurrent request using the Requests window. For example, your end user can see when a request starts running and then view the completion status and a completion message for a concurrent request.

Concurrent Processing Options

Oracle Application Object Library uses the values of user profile options as default values to apply to all concurrent requests that a user submits.

- Number of report copies to print of a report
- Save report output to a file in your operating system
- Printer on which to print a report
- Start date and time for a concurrent request
- Run requests sequentially
- Hold a request temporarily
- Priority of a concurrent request

• Who can view the report

Users can set some of these options for an entire login session using user profiles, and they can change some of these values at any time. If the request is submitted through Standard Request Submission, they can change printing and submission information when submitting the request. After users submit a concurrent request, they or your system administrator can modify these processing options for a concurrent request up until the time it starts running. Use the Requests form to modify the request's concurrent options. During runtime, you can use the Oracle Application Object Library utility **fdppov()** (See page NO TAG) in your Pro*C concurrent programs to change user profile options.

This change is only effective during the runtime of your concurrent programs.

Viewing the Status of Concurrent Requests

Your end user can check on a request to find out whether it is running, waiting to be started, has completed successfully, or has terminated with an error. You can build your concurrent programs so that if they fail to complete successfully, the concurrent manager displays an error message indicating the problem. If a request is pending normally, your user can determine how soon it will start by reviewing the request's position in the queues of the various concurrent managers that can run that particular request.

Automated Recovery for Concurrent Processing

Concurrent processing is an important component for your day to day operation of Oracle Applications. You can operate your Oracle Applications smoothly if you understand how concurrent managers react to different kinds of unforeseen situations. Your concurrent manager can detect concurrent programs or concurrent processes that terminate abnormally. It is also capable of automatically recovering from abnormal situations like operating system or internal failures such as segmentation faults. This section describes the actions the concurrent manager takes to recover from typical system problems.

Aborting Concurrent Programs

After you or your user submit a concurrent request, there may be situations where you want to terminate the running request. You can terminate a running request by changing the status to Completed in the Requests form. You should always terminate running requests from these forms whenever possible so that your concurrent manager can update the status of these requests accordingly.

The Requests form only allows you to cancel programs that you submitted and that are in your report security group. Use the privileged System Administration form Concurrent Requests Summary to cancel other requests as necessary.

If a concurrent request process is interrupted by a system signal or segmentation fault, your concurrent manager detects the disruption of the running request and updates the request phase/status to Completed/Error. Your concurrent manager then goes on to process other pending concurrent requests. If the disrupted request is a sub–request, your concurrent manager updates its status to Error and restarts the parent request. The parent request then communicates with your concurrent manager whether to abort or continue processing its remaining sub–requests. No other recovery procedures are required to resume concurrent processing.

Concurrent Manager Process Terminations

When you start up your concurrent processing facility, the internal concurrent manager starts up all the concurrent manager processes defined. The internal concurrent manager monitors the operation of these concurrent manager processes to make sure they function as defined. If any of these processes exits abnormally, the internal concurrent manager starts up a new process to ensure the correct number of concurrent manager processes are running. This monitoring process is completely invisible to you or your users.

Typically, if a concurrent manager process terminates abnormally while running a request, the request then completes with a phase/status of Complete/Error. If the running request is a sub–request (such as a member of a report set), the request completes with an Error status. When the parent request (such as a report set) restarts and detects the failure of the report, it notifies the concurrent manager process whether to abort or continue processing other sub–requests. If the running request is a parent request (such as a report set), the request completes with an Error status and the status of its sub–requests are all updated to Error.

If the failing concurrent manager process is an internal concurrent manager process, all you need to do is to restart your concurrent processing facility. Although the internal concurrent manager terminates abnormally, other concurrent manager processes continue to operate and newly submitted concurrent requests keep going into other available concurrent manager queues.

The only concurrent requests affected by a failure of the internal concurrent manager process are run alone concurrent programs and concurrent programs that have incompatibilities. If these concurrent requests are submitted after the internal concurrent manager exits abnormally, they remain in pending status until you restart the internal concurrent manager process. If these concurrent requests are submitted before the internal concurrent manager's abnormal exit, they remain pending and hold up all other concurrent requests belonging to the same logical database unless you put these affected requests on hold. Once your internal concurrent manager is running again, it resumes the duty of monitoring other concurrent manager processes and coordinating the processing of concurrent programs that are run alone or have incompatibilities.

Administer Concurrent Managers
Oracle Applications System Administrator's Guide

Shutdowns of Operating System and Database

Unusual operating system exits and abnormal database shutdowns are two common reasons that cause concurrent manager processes to fail. In these situations, all the concurrent manager processes are terminated without any notice and the phase and status of your concurrent requests remain as they are. All you have to do to resume normal concurrent processing is restart your concurrent processing facility. Once you restart your concurrent processing facility, your concurrent managers rerun all the requests that were running at the time the concurrent manager processes failed. After processing the previously running requests, the concurrent managers then process other pending concurrent requests.

Printer Support

Oracle Application Object Library provides printer drivers that correspond to print styles for concurrent program output. These drivers provide the four print styles for a variety of printers.

- L (Landscape)
- P (Portrait)
- A (A4)
- W (Landwide)

First the concurrent manager looks for a printer driver you can define with the name of printer type concatenated with the print style. The printer type is associated with the printer. The print style is associated with the concurrent program. For Oracle Reports, every printer driver is associated with an Oracle Reports driver of the form (L.prt). These Oracle Reports drivers contain printer specific formatting characters such as bold, italic, underline and new page markers.

When you review your reports on line, reports that use the Oracle Application Object Library printer drivers display correctly. Reports that use custom printer drivers may not display correctly on certain terminals.

Overview of Implementing Concurrent Processing

To build applications that take advantage of concurrent processing, you should understand aspects common to all types of concurrent programs as well as how to implement each type.

Choosing Your Implementation

Oracle Application Object Library provides several different implementation methods for concurrent programs:

- Oracle Tool concurrent programs
- Pro*C concurrent programs
- Host language concurrent programs

Before you begin writing your program, you should weigh the advantages of each method and choose the one that best fits your needs.

Oracle Tool Concurrent Programs

Oracle Reports, PL/SQL, SQL*Loader, and SQL*Plus programs are the simplest to write and integrate with Oracle Application Object Library. You can also write PL/SQL stored procedures as concurrent programs.

Attention: Starting with Release 11, SQL*Report (RPT) is no longer supported.

Concurrent Processing with Oracle Reports (See page 18 – 2)

Pro*C Concurrent Programs

You can write either spawned or immediate concurrent programs in C and Pro*C. Spawned concurrent programs are stand–alone programs that run in a separate process. (On some operating systems, you can also write spawned concurrent programs in your operating system command language. See your *Oracle Applications Installation Manual* for specific details.) Immediate concurrent programs run as subroutines of a concurrent manager.

Spawned concurrent programs are not linked with a concurrent manager. On most operating systems, concurrent managers start spawned concurrent programs in a separate operating system process than the concurrent manager. Spawned concurrent programs therefore require more system resources. In a spawned concurrent program, your SQL statements do not remain parsed between separate invocations of the program.

Immediate concurrent programs run as subroutines in C or Pro*C. You build a program library of immediate concurrent programs and assign the program library to concurrent managers. To call an immediate concurrent program, a concurrent manager invokes a subroutine call.

Immediate concurrent programs execute in the same operating system process as the concurrent manager on most operating systems. Since the concurrent manager is already connected to the database, your program does not have to explicitly connect. Additionally, because the process does not end when your program ends, the SQL statements in your program can remain parsed after the first invocation of the program. Subsequent invocations of the same program can run faster because the database does not have to parse the SQL statements.

However, immediate programs are also harder to maintain and support. Since they run as a subroutine of the concurrent manager, failures can sometimes affect the manager itself. We recommend implementing new Pro*C concurrent programs as spawned. In future releases, we will only support Pro*C programs as spawned, as PL/SQL stored procedures provide a mechanism for running concurrent programs in the same process as the concurrent manager.

Attention: The immediate concurrent program functionality is provided for backward compatibility only. You should not create new immediate concurrent programs.

Host Language Concurrent Programs

Depending on which operating system you are using, you implement host language concurrent programs in different ways. Use the Concurrent Programs section of the Application Object Library Appendix to the Installation manual to obtain specific details for your platform.

You can use host language programs to integrate an external program or package with Oracle Applications. For example, on some platforms you can create a shell script as a host language program. Your script can call a third–party program and then determine whether that program completes successfully. The host program then passes this information back to the concurrent manager.

Oracle Application Object Library Reference Material
Oracle Applications Installation Manual

Writing to Log and Output Files

Since your concurrent programs run non–interactively, they must print output to files. Your Pro*C program should write report output to the an out file and debugging or other technical messages to a log file. See the *Oracle Applications Installation Manual* for details on where these files are located for your platform.

Log and Output Filenames
Oracle Applications Installation Manual

Suggestion: Writing error messages to a log file rather than to an output file makes it easier for users and system administrators to find reports and diagnostic information, and does not confuse novice users who may be unprepared to see technical error messages when they review their reports online.

There are several methods to write to a standard log file or report output file.

The first method is the easiest method. It is also the only portable method across platforms. You call an Oracle Application Object Library utility routine from your C or Pro*C concurrent program:

fdpwrt() Writes a message to a standard log or report output file. Oracle Application Object Library names this file in the standard naming convention.

We highly recommend this method as the simplest to support and the most robust across platform and release changes.

The second method is to use standard C functions such as **fopen()** or **fclose()** to write to files. If you use this method, you must use the Oracle Application Object Library standard naming convention for your file names. Oracle Application Object Library uses the standard naming convention to find your report and log files to display online in the View Requests form and to automatically print your report output.

See the Application Object Library Reference Material Appendix in your *Oracle Applications Installation Manual* for the location of the **log** and **out** directories and the standard naming conventions for these files on your operating system. The *Oracle Applications Installation Manual* also contains information on how to change the default file protection on your report output and log files.

This second method exists for compatibility with concurrent programs written with prior versions of Oracle Application Object Library. When writing new concurrent programs, choose the first method and use the **fdpwrt()** utility routine.

Concurrent Processing Utility Routines (See page 20 – 2)

Printing Report Output Files

When you define your concurrent program, you can specify whether the report output prints automatically and the maximum and minimum row and columns it needs. You can also specify a recommended or mandatory print style. The concurrent manager uses the values of user profile options to send copies of report output to a specific printer. Reports submitted through Standard Request Submission have printing and submission options specified at submission time.

A user can change the printer, number of report copies, and the print style when requesting a reprint of report output in the Detail zone of the Submit Requests form.

Implementing User Profiles (See page 13 – 5)

Define Concurrent Program (See page 16 – 10)

Data Groups

If you want users of another application to be able to run your program, your application must be included in the appropriate data groups. Please read the Cross–application Reporting section in the Standard Request Submission chapter for more details.

Managing Concurrent Programs and Reports
Oracle Applications System Administrator's Guide

Tracking Data Changes With Who

If you add special Who columns to your tables, your application users can track changes made to their data. Oracle Application Object Library lets users differentiate between changes they make using forms and changes they make using concurrent programs. Columns that tell you information about the creation and the last update of a row are:

- LAST_UPDATED_BY
- LAST_UPDATE_DATE
- LAST_UPDATE_LOGIN

- CREATED_BY
- CREATION_DATE

Add the following columns to tell you information about the concurrent program that updated a row:

- REQUEST_ID, type NUMBER(15)
- PROGRAM_APPLICATION_ID, type NUMBER(15)
- PROGRAM_ID, type NUMBER(15)
- PROGRAM_UPDATE_DATE, type DATE

You should allow NULLs in these columns because they do not have values when a user inserts a new row using a form.

If you include Who columns in a table, your concurrent program should update some of them during every insert and update operation. Oracle Application Object Library loads user profile options with the value you use to update these columns. Call the function **fdpgov()** at the beginning of your concurrent program to get the current values of these user profile options. Use the values of the following user profile options to update your Who columns. Update the CREATED_BY column only if you insert a row.

Who Column Name	Profile Option Name
REQUEST_ID	CONC_REQUEST_ID
PROGRAM_ APPLICATION_ID	CONC_PROGRAM_ APPLICATION_ID
PROGRAM_ID	CONC_PROGRAM_ID
LAST_UPDATE_LOGIN	CONC_LOGIN_ID
CREATED_BY	USER_ID

Table 15 – 1 (Page 1 of 1)

Use your operating system's current date or SQL's SYSDATE to update the following Who columns. Update the CREATION_DATE column only if you insert a row.

- PROGRAM_UPDATE_DATE
- CREATION_DATE

Implementing User Profiles (See page 13 – 5)
Concurrent Processing Utility Routines (See page 20 – 2)

CHAPTER

16 Defining Concurrent Programs

This chapter provides an overview of how to define your concurrent program within Oracle Applications. It describes the forms necessary to define your program, executable, and library.

The following topics are covered:

- Overview of Defining Concurrent Programs
- Concurrent Program Executable Window
- Concurrent Programs Window
- Concurrent Program Libraries Window

Defining Concurrent Programs

After you have defined the logic for your program, you must define a concurrent program that you call from your forms or from Standard Request Submission. You first define a concurrent program executable that tells the application where to find your program logic. You then define as many concurrent programs as you need that use your concurrent program executable. You can define which concurrent programs are incompatible with your concurrent program, and if you use Standard Request Submission, you define the parameters used by your executable. You can specify validation information for your parameters to control the values users select on the Submit Request form.

If your program is a Pro*C immediate program, you must include the concurrent program in a program library. After you change a concurrent program library, you must rebuild the library and then relink it. Only concurrent managers using a library that includes your concurrent program can run your concurrent program.

If your concurrent program uses Standard Request Submission, you should use Oracle System Administration to add your concurrent program to the appropriate request groups. Users can then submit your concurrent program using the Submit Request form.

Define Your Concurrent Program Executable

When you have completed your program and placed the execution file in the correct directory of your application's base directory, you must define a concurrent program executable with Oracle Application Object Library, using the Concurrent Program Executable form. When you define your program executable, use the name that corresponds to your program's execution file without an extension. You use this name when you define a concurrent program using this executable.

Development Environment
Oracle Applications Installation Manual

Specify the execution method appropriate for your execution file. Oracle Application Object Library looks for the execution file based on which execution method you identify. Your concurrent programs use the concurrent program executable to locate and run the correct execution file.

If your program is a Pro*C routine, and you want to run it as a subprocess linked in with a concurrent manager, specify the name of your subroutine as well as the execution file name and select immediate as your execution method. The subroutine name is the name you substituted for SUBROUTINE_NAME in the program templates. Although you select an execution method here, you may create both spawned and immediate concurrent program using this executable. You must include any concurrent program you write using the immediate execution method in a concurrent program library.

We recommend using the spawned execution method for your Pro*C programs whenever possible, as they are easier to maintain and support than immediate programs. In future releases, only spawned Pro*C programs will be supported, as PL/SQL stored procedures provide you with a mechanism for writing custom concurrent programs that do not spawn independent processes.

PL/SQL stored procedures are immediate programs in that they do not spawn an independent process, but you should specify the name of the procedure in the execution file field and select PL/SQL Stored Procedure as your execution method.

Define Concurrent Program Executable (See page 16 – 6)

Define Your Concurrent Programs

Define your concurrent program using your executable with the Oracle Application Object Library form Concurrent Programs. Give each concurrent program a user–friendly program name used when selecting your concurrent program from an end–user List of Values. This name should convey the program's purpose. Specify a short name for the applications to pass to the concurrent manager or for you to use when submitting your request through CONCSUB, the FND_REQUEST PL/SQL API, or #FND CONCURRENT. For example, in your program file you might write an initial function called **glpost()**, and then define your executable with the name GL_POST. The concurrent program you define with the name General Ledger Posting and the short name GL_POST.

If you do not wish to make your concurrent program available through Standard Request Submission, you leave the "Use in SRS" box unchecked. If your program is a Standard Request Submission report, you can permit the report to access all values, including obsolete or disabled values, or you can limit the report to current values. You can disable your program at any time. Disabled programs do not show up in end–user Lists of Values, and concurrent managers do not run requests for disabled programs.

Specify the concurrent program executable you previously defined for your program. You can decide whether to run your Pro*C program as spawned or immediate (linked in with the concurrent manager process) if you specified both a execution file name and a subroutine.

If your concurrent program generates output, you should specify the maximum and minimum dimensions of the printed report. This information determines which printer styles can accommodate your report requirements. You can choose to have the concurrent manager automatically print the report output file if your program creates one, and you can choose a print style.

When you define your program, define any incompatibilities it has with other concurrent programs or itself. List any concurrent programs that your program should not run against in the same logical database. If the concurrent program cannot run while any other concurrent program runs, specify that your program is a Run Alone program.

If your concurrent program uses Standard Request Submission, define the parameters to pass to your execution file. You can define value sets and set parameter validation similar to defining a flexfield segment. This information determines the behavior of the segments in the pop–up parameter window on the Run Reports form.

If your concurrent program uses Oracle Reports, you should specify the tokens you defined for each parameter.

After you define your concurrent program, use Oracle System Administration to reset the concurrent managers. This forces the concurrent managers to recognize new concurrent programs or any changes you have made in program registration. However, if you change the Execution Method of your program from Spawned to Immediate, your system administrator needs to shutdown and restart the concurrent managers so that the concurrent managers recognize the change.

Define Concurrent Program (See page 16 – 10)

Defining Your Concurrent Program Library

Use the Register Concurrent Program Library form to define your program library. Specify the Library Name in the Program Library zone and the application whose base directory your execution file resides in. In the Concurrent Programs zone, list all the concurrent programs that you defined as immediate with Oracle Application Object Library that you want to include in this program library.

Before you can run your immediate Pro*C concurrent program, use Oracle System Administration to assign the library including the program to a concurrent manager. If an immediate concurrent program is not in a concurrent manager's library, that concurrent manager cannot process a request to run that immediate concurrent program.

Rebuild your library and relink it whenever you add new concurrent programs. Then restart your concurrent manager before requesting your concurrent program.

The Oracle Applications installation process defines the Oracle Application Object Library FNDLIBR program library. This library contains Oracle Applications immediate concurrent programs. You should assign this library to at least one concurrent manager.

Register Concurrent Program Library (See page 16 – 23)

Give Access to the Program

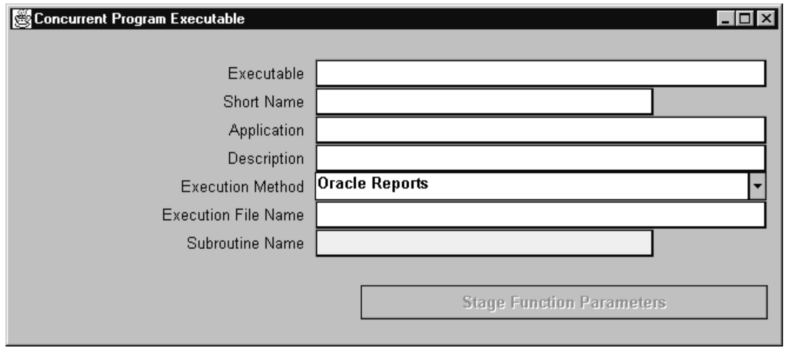
If you want users of another application to be able to run your program, you should include your program's application in the data groups for the responsibilities of the other application. The concurrent program runs in the ORACLE ID associated with its application in the current responsibility's data group.

Define Data Group
Oracle Applications System Administrator's Guide

To allow users to run your Standard Request Submission program from the Submit Requests form, add your program to the request group for their responsibility.

Define Report Group
Oracle Applications System Administrator's Guide

Concurrent Program Executable Window



Define a concurrent program executable for each executable source file you want to use with concurrent programs. The concurrent program executable links your source file logic with the concurrent requests you and your users submit to the concurrent manager.

The Installation Guide for your operating system details where to place the execution files for each execution method.

☞ **Attention:** You cannot add new immediate programs to a concurrent manager program library. We recommend that you use spawned concurrent programs instead.

Define Concurrent Program : page 16 – 10
Register Concurrent Program Library : page 16 – 23

See: Concurrent Processing
(*Oracle Applications Installation Guide)*

Concurrent Program Executable Block

The combination of application name plus program name uniquely identifies your concurrent program executable.

See: Concurrent Programs Window: page 16 – 10

Executable

Enter a name for your concurrent program executable. In the Concurrent Programs window, you assign this name to a concurrent program to associate your concurrent program with your executable logic.

Application

The concurrent managers use the application to determine in which directory structure to look for your execution file.

Execution Method

The execution method cannot be changed once the concurrent program executable has been assigned to one or more concurrent programs in the Concurrent Programs window.

The possible execution methods are:

FlexRpt	The execution file is wrnitten using the FlexReport API.
FlexSql	The execution file is written using the FlexSql API.
Host	The execution file is a host script.
Oracle Reports	The execution file is an Oracle Reports file.
PL/SQL Stored Procedure	The execution file is a stored procedure.
SQL*Loader	The execution file is a SQL script.
SQL*Plus	The execution file is a SQL*Plus script.
Spawned	The execution file is a C or Pro*C program.
Immediate	The execution file is a program written to run as a subroutine of the concurrent manager. We recommend against defining new immediate concurrent programs, and suggest you use either a PL/SQL Stored Procedure or a Spawned C Program instead.
Request Set Stage Function	PL/SQL Stored Function that can be uesd to calculate the completion statuses of request set stages.

Execution File Name

Enter the operating system name of your execution file. Some operating systems are case sensitive, so the name entered here should match the file name exactly.

Do not include spaces or periods (.) in the execution file name, unless the execution method is PL/SQL stored procedure or Request Set Stage

Function. See the *Oracle Applications Installation Guide* for details on the path Oracle Applications uses to find each executable file.

The maximum size of an execution file name is 60 characters.

Subroutine Name

Enter the name of your C or Pro*C program subroutine here. Do not use spaces or periods (.) in this field.

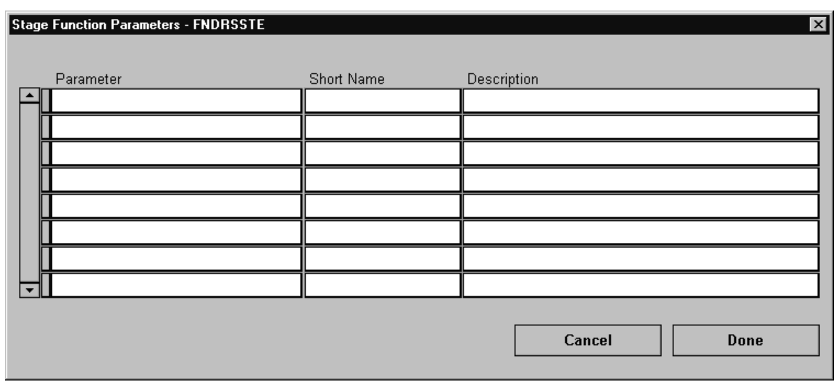
Only immediate programs or spawned programs using the Unified C API use the subroutine field.

We recommend against defining new immediate concurrent programs, and suggest you use either a PL/SQL Stored Procedure or a Spawned C Program instead.

Stage Function Parameters

The Stage Function Parameters button opens a window that allows you to enter parameters for the Request Set Stage Function. This button is only enabled when you select Request Set Stage Function as your Executions Method.

Stage Function Parameters Window



List the Parameters that your custom Stage Function uses.

Parameter

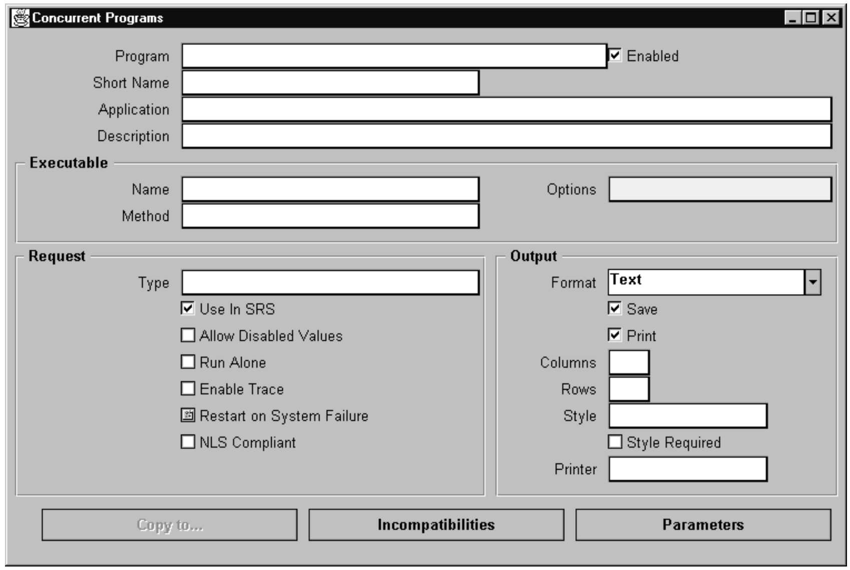
Enter a name for the Parameter. This name will be displayed in the Stage Functions Parameter window of the Request Set form.

Short Name

Enter a short name that will be used by the function to reference the parameter.

Implementing Concurrent Processing : page 15 – 21

Concurrent Programs Window



Use this window to define and modify your concurrent programs.

Overview of Concurrent Processing: page 15 – 2

Define Concurrent Program Executable: page 16 – 6
Register Concurrent Program Library: page 16 – 23

Prerequisites

- Build the execution file for your concurrent program.
- Use the Concurrent Program Executables window to define a concurrent program executable for your operating system program.

Concurrent Programs Block

The combination of application name plus program name uniquely identifies your concurrent program.

Program

You see this longer, more descriptive name when you view your requests in the Requests window. If this concurrent program runs through Standard Request Submission, you see this name in the Submit Requests window when you run this program.

Short Name

Enter a brief name that Oracle Applications can use to associate your concurrent program with a concurrent program executable.

Application

The program's application determines what ORACLE username your program runs in and where to place the log and output files.

Enabled

Indicate whether users should be able to submit requests to run this program and the concurrent managers should be able to run your program.

Disabled programs do not show up in users' lists, and do not appear in any concurrent manager queues. You cannot delete a concurrent program because its information helps to provide an audit trail.

Executable

Name

Select the concurrent program executable that can run your program. You define the executable using the Concurrent Program Executables window. You can define multiple concurrent programs using the same concurrent program executable. See: Concurrent Program Executables: page 16 – 6.

Options

If you define a concurrent program with the bitmapped version of Oracle Reports, you can control the orientation of the bitmapped report by passing the ORIENTATION parameter or token. For example, to generate a report with landscape orientation, specify the following option in the Options field:

```
ORIENTATION=LANDSCAPE
```

Do not put spaces before or after the execution options values. The parameters should be separated by only a *single* space. You can also specify an orientation of PORTRAIT.

You can control the dimensions of the generated output with the PAGESIZE parameter. A specified <*width*>x<*height*> in the Options field overrides the values specified in the report definition. For example:

```
ORIENTATION=LANDSCAPE PAGESIZE=8x11.5
```

The units for your width and height are determined by your Oracle Reports definition. You set the units in your Oracle Reports menu under Report => Global Properties => Unit of Measurement.

If the page size you specify with the PAGESIZE parameter is smaller than what the report was designed for, your report fails with a ”REP–1212” error.

Method

The execution method your concurrent program uses appears here.

Valid values are:

Spawned	Your concurrent program is a stand–alone program in C or Pro*C.
Host	Your concurrent program is written in a script for your operating system.
Immediate	Your concurrent program is a subroutine written in C or Pro*C. Immediate programs are linked in with your concurrent manage and must be included in the manager’s program library.
Oracle Reports	Your concurrent program is an Oracle Reports script.
PL/SQL Stored Procedure	Your concurrent program is a stored procedure written in PL/SQL.
SQL*Loader	Your concurrent program is a SQL*Loader program.

SQL*Plus	Your concurrent program is a SQL*Plus or PL/SQL script.
Request Set Stage Function	PL/SQL Stored Function that can be uesd to calculate the completion statuses of request set stages.

You can switch between Spawned and Immediate, overriding the execution method defined in the Concurrent Program Executable window, only if either method appears when the executable is selected and both an execution file name and subroutine name have already been specified in the Concurrent Program Executable window. See: Concurrent Program Executables: page 16 – 6.

Priority

You can assign this program its own priority. The concurrent managers process requests for this program at the priority you assign here.

If you do not assign a priority, the user's profile option Concurrent:Priority sets the request's priority at submission time.

Request

Type

If you want to associate your program with a predefined request type, enter the name of the request type here. The request type can limit which concurrent managers can run your concurrent program.

You can define a concurrent manager to run only certain types of concurrent requests. See: Concurrent Request Types, *Oracle Applications System Administrator's Guide.*

Use in SRS

Check this box to indicate that users can submit a request to run this program from a Standard Request Submission window.

If you check this box, you must register your program parameters, if any, in the Parameters window accessed from the button at the bottom of this window.

Allow Disabled Values

If you check the Use in SRS box, you can also check this box to allow a user to enter disabled or outdated values as parameter values.

Many value sets use special table columns that indicate whether a particular value is enabled (using ENABLED_FLAG, START_DATE_ACTIVE, and END_DATE_ACTIVE columns). These value sets normally allow you to query disabled or outdated values but not enter them in new data. For Standard Request Submission, this means that a user would not normally be allowed to enter disabled values as report parameter values when submitting a report, even if the report is a query–only type report.

Run Alone

Indicate whether your program should run alone relative to all other programs in the same logical database. If the execution of your program interferes with the execution of all other programs in the same logical database (in other words, if your program is incompatible with all programs in its logical database, including itself), it should run alone.

You can enter any specific incompatible programs in the Incompatible Programs windows.

Enable Trace

Turns on SQL tracing when program runs.

Restart on System Failure

Use this option to indicate that this concurrent program should automatically be restarted when the concurrent manager is restored after a system failure.

Output

Format

Select the output format for your concurrent program from the following:

- Text
- HTML
- PDF
- PS (Post Script)

Save

Indicate whether to automatically save the output from this program to an operating system file when it is run. This value becomes the default for all requests submitted for this program. The output of programs with Save set to No is deleted after printing.

If this is a Standard Request Submission program, users can override this value from the Submit Requests window.

Print

Enter Yes or No to indicate whether to allow the concurrent managers to print your program's output to a printer. If you enter No, your concurrent program's output is never sent to the printer.

Columns / Rows

Enter the minimum column and row length for this program's report output. Oracle Applications uses this information to determine which print styles can accommodate your report.

Style

The print style you select depends on your system and printer setup. Print styles include:

- 132 columns and 66 lines (Landscape)
- 180 columns and 66 lines (Landwide)
- 80 columns and 66 lines (Portrait)
- 132 columns and 62 lines (A4)

Your list is limited to those styles that meet your program's columns and row length requirements.

Style Required

If your program requires a specific print style (for example, a checkwriting report), use this check box to enforce that print style.

Printer

If you want to restrict your program's output to a single printer, enter the name of the printer to which you want to send your output. If your program has minimum or maximum columns or rows defined, your list of values is limited to those printers that can support your program's requirements.

Users cannot override your choice of printer from the Submit Requests or Requests windows.

Concurrent Programs Buttons

Use these buttons to open detail windows for program incompatibilities your program parameters.

Copy to... Choose this button to create another concurrent program using the same executable, request and report information. You can elect to copy the incompatibility and parameter details as well.

Incompatibilities Choose this button to open the Incompatible Programs window.

Parameters Choose this button to open the Concurrent Program Parameters window.

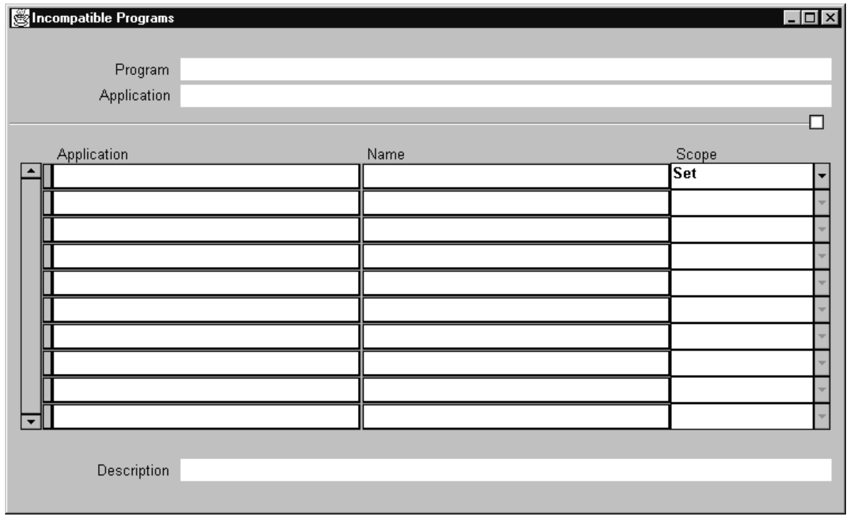
Copy to Window

Create another concurrent program using the same executable, request and report information as the current program. You can optionally copy the incompatibility and parameter details information as well.

See: Incompatible Programs Window: page 16 – 17

A system administrator must use the Administer Concurrent Managers window to have the internal concurrent manager verify the concurrent managers before any changes you make in this zone take effect.

Incompatible Programs Window



Identify programs that should not run simultaneously with your concurrent program because they might interfere with its execution. You can specify your program as being incompatible with itself.

See:

Administer Concurrent Managers
Oracle Applications System Administrator's Guide

Application

Although the default for this field is the application of your concurrent program, you can enter any valid application name.

Name

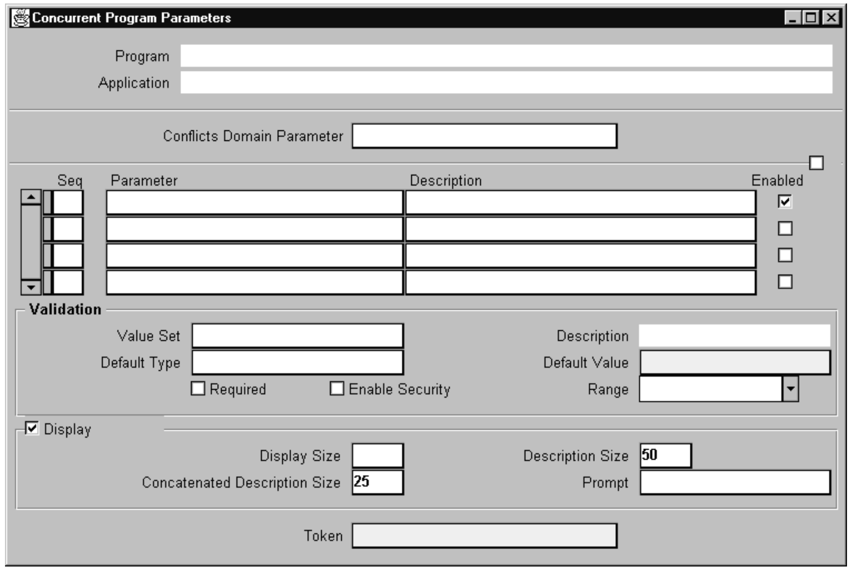
The program name and application you specify must uniquely identify a concurrent program.

Your list displays the user–friendly name of the program, the short name, and the description of the program.

Scope

Enter Set or Program Only to specify whether your concurrent program is incompatible with this program and all its child requests (Set) or only with this program (Program Only).

Concurrent Program Parameters Window



Enter and update the program parameters that you wish to pass to the program executable. Program parameters defined here should match the variables in your execution file.

Sequence

Choose the sequence numbers that specify the order in which your program receives parameter values from the concurrent manager.

Parameter

Enter the parameter name. The value is case insensitive.

Enabled

Disabled parameters do not display at request submission time and are not passed to your execution file.

Argument Detail

You specify information about your parameter almost exactly as you define a flexfield segment.

Validation Information

Value Set

Enter the name of the value set you want your parameter to use for validation. You can only select from independent, table, and non–validated value sets.

The maximum size of your value set is 240 characters.

Default Type

If you want to set a default value for this parameter, identify the type of value you need.

Valid types include:

Constant The default value can be any literal value.

Current Date The default value is the current date in the format DD–MON–YY or DD–MON–YYYY, depending on the length of the segment.

Maximum Size	Date Format
9	DD–MON–YY
11	DD–MON–YYYY

Current Time The default value is the current time or the current date and time, depending on the length of the segment.

Maximum Size	Time Format
5	HH24:MI
8	HH24:MI:SS
15	DD–MON–YY HH24:MI
17	DD–MON–YYYY HH24:MI
18	DD–MON–YY HH24:MI:SS
20	DD–MON–YYYY HH24:MI:SS

Profile The default value is the current value in the user profile option defined in the Default Value field. Use the profile option name, not the end–user name. You do not need to include $PROFILE$.

SQL Statement The default value is determined by the SQL statement you defined in the Default Value field.

Segment — The default value is the value entered in a prior segment of the same parameter window.

If you choose Current Date or Current Time, you skip the next field.

Default Value

You can enter a default value for the parameter. This default value for your parameter automatically appears when you enter your parameter window. You determine whether the default value is a constant or a context–dependent value by choosing the default type.

Your default value should be a valid value for your value set. Otherwise you see an error message when you enter your parameter window on the Run Request window and your default value does not appear.

Valid values for each default type include:

Constant — Enter any literal value for the default value.

Profile — The default value is the current value of the user profile option you specify here. Enter the profile option name, not the end–user name. You do not need to include $PROFILES$ (the Release 9 format).

Segment — The default value is the value entered in a prior segment of the same flexfield window. Enter the name of the segment whose value you want to copy.

SQL Statement — The default value is determined by the SQL statement you enter here. Your SQL statement must return exactly one row and one column in all cases.

Required

If the program executable file requires an argument, you should require it for your concurrent program.

Enable Security

If the value set for this parameter does not allow security rules, then this field is display only. Otherwise you can elect to apply any security rules defined for this value set to affect your parameter list.

Range

Choose either Low or High if you want to validate your parameter value against the value of another parameter in this structure. Parameters with a range of Low must appear before parameters with a range of High (the low parameter must have a lower number than the high parameter). For example, if you plan two parameters named "Start Date" and "End Date," you may want to force users to enter an end date later than the start date. You could assign "Start Date" a range of Low and "End Date" a range of High. In this example, the parameter you name "Start Date" must appear before the parameter you name "End Date."

If you choose Low for one parameter, you must also choose High for another parameter in that structure (and vice versa). Otherwise you cannot commit your changes.

If your value set is of the type Pair, this field is display only. The value defaults to Pair.

Window Information

Display

Indicate whether to display this parameter in the Parameters window when a user submits a request to run the program from the Submit Requests window.

You should provide a default type and value for any non–displayed parameter.

Display Size

Enter the field length in characters for this parameter. The user sees and fills in the field in the Parameters window of the Submit Requests window.

You should ensure that the total of the value set maximum sizes (not the display sizes) for all of your parameters, plus the number of separators you need (number of parameters minus one), does not add up to more than 240. If your program values' concatenated length exceeds 240, you may experience truncation of your data in some forms.

Description Size

Enter the display length in characters for the parameter value description. Your window may show fewer characters of your description than you specify here if there is not enough room (determined by the sum of your longest prompt plus your display size for this parameter plus seven). However, your window does not display more characters of the description than you specify here.

Prompt

A user sees the prompt instead of the parameter name in the Parameters window of the Submit Requests window.

The default is the name of the parameter.

Concatenated Description Size

Enter the display length in characters for the parameter value description. The user sees the parameter value in the Parameter Description field of the Submit Requests and View Requests forms. The Parameter Description field concatenates all the parameter values for the concurrent program.

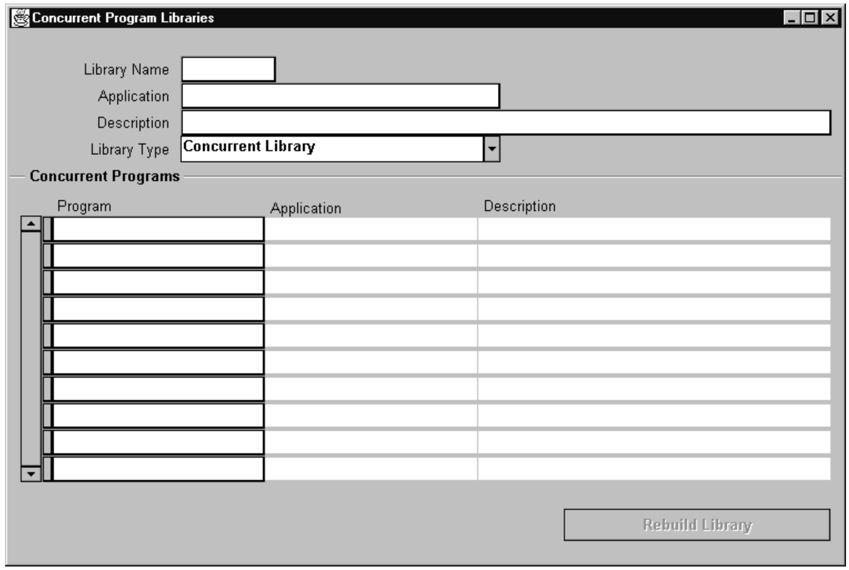
Suggestion: We recommend that you set the Concatenated Description Size for each of your parameters so that the total Concatenated Description Size for your program is 80 or less, since most video screens are 80 characters wide.

Token

For a parameter in an Oracle Reports program, the keyword or parameter appears here. The value is case insensitive. For other types of programs, you can skip this field.

See: Incompatible Programs Window: page 16 – 17

Concurrent Program Libraries Window



Use this window to register program libraries, which are lists of immediate concurrent programs that you wish to link with a concurrent manager. Concurrent managers use the programs in a program library to run their immediate programs. You must register libraries before you can define concurrent managers. You can only include immediate–type concurrent programs in program libraries.

After adding any immediate concurrent program to your library or creating a new library, you must rebuild and relink your library before your changes take effect. After you rebuild and relink your library, the system administrator must restart the concurrent manager using your library.

You can only register program libraries that you have already built at the operating system level.

Prerequisites

- Use the Applications window to register your application with Oracle Application Object Library.
- Use the Concurrent Program Executables window to define your concurrent program executable file.

Concurrent Program Libraries Block

The combination of application name plus library name uniquely identifies your set of programs.

Library Name

This is the same name you gave to your program library file at the operating system. The library name must be eight characters or less.

System administrators can then assign this library name to concurrent managers. Each concurrent manager is linked to one program library and can only run immediate–type programs that use concurrent program executables that are part of that library. A concurrent manager can run other execution type programs that are not in its program library.

Application

The bin directory under this application's top directory should contain the executable program library file.

Library Type

There are two types of program library you can define:

Concurrent Library A library of immediate concurrent programs to link with a concurrent manager.

Transaction Library A library of transaction programs to link with a transaction manager.

Concurrent Programs Block

List the concurrent program executables you have linked to this program library at the operating system level.

Program

Enter the name of an immediate–type concurrent program executable that you linked into your program library at the operating system. This block verifies that the program name and application name you specify uniquely identify a defined concurrent program executable.

Application

This is the application to which your concurrent program belongs.

Rebuild Library

Select this button when you need to rebuild your concurrent library. You should rebuild your library whenever you add new programs to your library.

CHAPTER

17

Coding Oracle Tools Concurrent Programs

This chapter provides an overview of how to code concurrent programs using various Oracle Tools, such as SQL*Plus and SQL*Loader.

The following topics are covered:

- SQL*Plus Programs
- PL/SQL Stored Procedures
- SQL*Loader
- Accepting Input Parameters for Oracle Tools Programs
- Naming Your Oracle Tool Concurrent Program

Oracle Tool Concurrent Programs

Oracle Application Object Library lets you write concurrent programs in SQL*Plus, PL/SQL (if you have PL/SQL installed on your database), SQL*Loader, or Oracle Reports.

Attention: Starting with Release 11, SQL*Report (RPT) is no longer supported.

Concurrent Processing with Oracle Reports (See page 18 – 2)

For SQL*Plus and PL/SQL programs, the concurrent manager logs onto the database, starts your program, automatically spools output to a report output file, and logs off the database when your program is complete. If your program produces report output, you can define your program to have the concurrent manager automatically print the report output file after your program completes. Reports submitted through Standard Request Submission have printing and submission information set at run time.

SQL*PLUS Programs

For SQL*Plus programs, the concurrent manager automatically inserts the following prologue of commands into your SQL*Plus script:

SQL*Plus Prologue

```
SET TERM OFF
SET PAUSE OFF
SET HEADING OFF
SET FEEDBACK OFF
SET VERIFY OFF
SET ECHO OFF
WHENEVER SQLERROR EXIT FAILURE
```

The concurrent manager also inserts a command into your SQL*Plus script to set LINESIZE according to the print style of the script.

If you want your SQL*Plus script to continue after a SQL error, you must insert the following line into your SQL*Plus script:

```
WHENEVER SQLERROR CONTINUE
```

PL/SQL Stored Procedures

PL/SQL stored procedures behave like immediate concurrent programs in that they do not require the concurrent manager to create an independent spawned process.

Concurrent Processing APIs for PL/SQL Procedures(See page 20 – 2)

Beginning with Release 11, concurrent programs using PL/SQL stored procedures can generate log files or output files.

PL/SQL File I/O Processing

Package FND_FILE contains routines which allow PL/SQL stored procedures running as concurrent programs to write to the request log and output files, stored under <PROD_TOP>/log and <PROD_TOP>/out.

Text written by the stored procedures is first kept in temporary files on the database server, and after request completion is copied to the log and out files by the manager running the request. Opening and closing files is handled behind the scenes by the concurrent manager. Every read and write to the temporary files is implicitly flushed to minimize risk of data loss.

The concurrent managers maintain a shared pool of temporary files; when a manager starts up, it attempts to use filenames from the pool. If no filenames exist, the manager creates new temporary log and output files. These two files are cleared after each concurrent request, and then reused for the next request. As a result no more temporary files are created than necessary.

The temporary files are named as follows, where the x's indicate a sequence number, padded to 7 digits:

```
lxxxxxxx.req
oxxxxxxx.req
```

The directory for temporary files must be set in the environment variable APPLPTMP when the managers are started. This directory must also be listed in the UTL_FILE_DIR parameter in init.ora.

To write to these log and output files, simply call the necessary procedures. Opening and closing the files is handled by the concurrent managers. Procedure arguments and exceptions are detailed below.

There are several limitations of these procedures. The temporary files cannot be deleted, but are reduced to 0–length. Deleting them must be handled by the system administrator. FND_FILE can only be used within PL/SQL procedures if the PL/SQL procedure itself is a concurrent program. If the PL/SQL procedure is being called by another concurrent program which is not a PL/SQL procedure FND_FILE package cannot be used. This package is not designed for generic PL/SQL text I/O. It is only used for writing to request log and output files.

Using these APIs may impact your application's performance. Temporary files are first created and then copied over the network to the request log and out files. Moving large files can be slow, and can create considerable network traffic. You may wish to be conservative with the amount of data written from your PL/SQL procedure.

To facilitate debugging and testing from SQL*Plus, you can use the procedure FND_FILE.PUT_NAMES(LOG, OUT, DIR). This function sets the temporary log and out filenames and the temporary directory to the user–specified values. DIR must be a directory to which the database can write. FND_FILE.PUT_NAMES should be called before calling any other FND_FILE function. If this function is not called when using SQL*Plus, FND_FILE will choose a filename from the pool, as described above. FND_FILE.PUT_NAMES works only once per session, and it does nothing if called from a concurrent program. Procedure FND_FILE.CLOSE will close the files in a command–line session. FND_FILE.CLOSE should not be called from a concurrent program; the concurrent manager will handle closing files.

See: FND_FILE page: 20 – 15

SQL*Loader

For SQL*Loader programs, the concurrent manager runs SQLLOAD on the control file specified on the Concurrent Program Executable form. If your program produces report output, you can define your program to have the concurrent manager automatically print the report output file after your program completes.

You can either supply information about the data file in the control file, or pass the full directory path and file name of your data file as an argument. The concurrent manager passes the "data=(full pathname of data file)" token at request run time. Without a data file name, the concurrent manager skips that token and SQL*Loader uses the data file name specified in the control file.

If you port your application to a different operating or hardware system, check the directory and file name of your data file and change the value of your program argument if necessary.

Accepting Input Parameters For Oracle Tool Programs

You should write your program to receive arguments in the same order that you specify when you call your program and pass arguments. Concurrent managers pass the arguments directly to your programs.

In SQL*Plus and PL/SQL programs, you must name your arguments &1, &2, &3, etc. so that you are guaranteed to get the first argument you pass in &1, the second in &2, and so on.

With PL/SQL stored procedures, you should define your arguments as IN parameters.

In SQL*Loader programs, pass the full directory path and file name of your data file as an argument. If you port your application to a different operating or hardware system, check the directory and file name of your data file and change the value of your program argument if necessary.

Oracle Reports Parameters (See page 18 – 4)

Naming Your Oracle Tool Concurrent Program

If your operating system is case–sensitive, the file name of your Oracle Tool concurrent program should always be in uppercase and the extension in lowercase.

Use the *Oracle Applications Installation Manual* for your operating system to determine the correct naming conventions for your Oracle Tool programs.

Oracle Tool Concurrent Programs
Oracle Applications Installation Manual

Suggestion: Effective with Oracle Applications Release 10, you cannot reference the interface program **FDPSGP** when submitting your Oracle Tool concurrent programs. You should define your Oracle Tool concurrent programs to use the appropriate execution methods.

CHAPTER

18

Coding Oracle Reports Concurrent Programs

This chapter tells you about using Oracle Reports with the Oracle Application Object Library concurrent manager. It includes a discussion on integrating custom Oracle Reports programs with Oracle Applications, a list of user exits available for your programs, an example program, and a troubleshooting guide.

The following topics are covered:

- Overview of Oracle Reports Concurrent Programs
- User Exits in Oracle Reports
- Using Dynamic Currency in Oracle Reports
- Oracle Reports Troubleshooting

Oracle Reports

You can write Oracle Reports reports, integrate them with Oracle Application Object Library, and run them as concurrent programs from your forms or though Standard Request Submission.

In this chapter, the Oracle Reports executable file is referred to as **ar25run** or **ar25runb** (for the bitmapped version). The name of your Oracle Reports executable file may vary depending on which version of Oracle Reports you use.

You have the option of using Oracle Application Object Library user exits such as dynamic currency formatting with your Oracle Reports programs.

The concurrent manager can run Oracle Reports either in character mode or in bitmap mode according to your specifications. You control the orientation and page size of your report output.

A troubleshooting guide for your Oracle Reports programs appears at the end of this chapter.

Concurrent Processing with Oracle Reports

Most Oracle Applications reports are launched from the concurrent manager as concurrent processes. In most UNIX systems, the concurrent manager inherits its environment variables from the shell from which it was started; the reports are then run from this environment.

Oracle Reports Integration

For Oracle Reports programs, the concurrent manager runs the executable **ar25run** or **ar25runb** on the report description file. **ar25runb** is used for bitmap reports (including PostScript, HTML, and PDF files). Both executables include Oracle Applications user exits. If your Oracle Reports program produces report output, the concurrent manager can automatically print the report output file after your program completes, provided that your site has the appropriate print drivers defined.

Using PL/SQL Libraries

Immediately before running a report, the concurrent manager dynamically prepends several values onto the environment variable $REPORTS25_PATH, as shown below:

```
REPORTS25_PATH =
$[PROD]_TOP/$APPLPLS:$[PROD]_TOP/$APPLREP
    :$[PROD]_TOP/$APPLREP/LANGDIR
    :$AU_TOP/$APPLPLS:$REPORTS25_PATH
```

This puts the PL/SQL libraries in the `$[PROD]_TOP`/$APPLPLS, any other report objects such as external queries, boiler plate text etc. in `$[PROD]_TOP`/$APPLREP, and sharable libraries in $AU_TOP/$APPLPLS in REPORTS25_PATH before the concurrent manager runs a report. `$[PROD]_TOP` is the application basepath of the application owning the report, and LANGDIR is the directory for a particular language, such as US or FR.

The APPLSYS.env, set at installation, sets REPORTS25_PATH to $AU_TOP/$APPLPLS. This may be modified to include customized libraries.

Oracle Applications Installation Manual

Bitmapped Oracle Reports

If you define a concurrent program with the bitmapped version of Oracle Reports, select PostScript, HTML, or PDF as appropriate from the Output Type poplist in the Define Concurrent Program form.

You can control the orientation of the bitmapped report by passing the ORIENTATION parameter or token. For example, to generate a report with landscape orientation, specify the following option in the Execution Option field:

```
ORIENTATION=LANDSCAPE
```

Do not put spaces before or after the execution options values. The parameters should be separated by only a *single* space. You can also specify an orientation of PORTRAIT.

You can control the dimensions of the generated output with the PAGESIZE parameter. A specified <*width*>x<*height*> in the Execution Options field overrides the values specified in the report definition. For example:

```
ORIENTATION=LANDSCAPE PAGESIZE=8x11.5
```

The units for your width and height are determined by your Oracle Reports definition. You set the units in your Oracle Reports menu under Report => Global Properties => Unit of Measurement.

If the page size you specify with the PAGESIZE parameter is smaller than what the report was designed for, your report fails with a "REP–1212" error.

Oracle Reports Parameters

Though the concurrent manager passes program arguments to your Oracle Reports program using tokens (so that their order does not matter), you should write your program to receive arguments in the same order that you specify when you call your program and pass arguments for easier maintenance.

Your Oracle Reports program parameters should not expect NULL values. The concurrent manager cannot pass a NULL value to your program.

For Oracle Reports programs you have a choice of two implementation methods.

Standard Request Submission

If you choose to make your Oracle Reports program available through Standard Request Submission, you check the Use in SRS check box of the Concurrent Programs form and define your arguments in the Concurrent Program Parameters block. Your program is available for the Submit Request form once you use Oracle System Administration to add your program to the appropriate report security groups.

If you also call your program using FND_REQUEST.SUBMIT_REQUEST from a form other than the Submit Request form, you supply values for your arguments in the order in which you registered them. The concurrent manager automatically adds the tokens you defined when you registered your arguments to the values you supply when you submit the program from the Submit Request form or from FND_REQUEST. The concurrent manager passes tokenized arguments (token1=parameter1, token2=parameter2, etc.) to your Oracle Reports program. In this case, each parameter *value* can be up to 240 characters in length, excluding the length of the associated token.

Non–Standard Request Submission

If you do not wish to make your Oracle Reports program available through Standard Request Submission, you pass tokens to your Oracle Reports program in your FND_REQUEST call from a form. In this case you do not check the Use in SRS check box of the Concurrent Programs form. Note that each argument of the form *TOKEN=parameter* must be a maximum of 240 characters in length, including the token name.

Accessing User Exits and Profile Options

Oracle Application Object Library lets you access user profile information and run user exits from your Oracle Reports program by including the appropriate calls in your program. These Oracle Application Object Library calls also allow your report to access the correct organization (for multiple organizations or "multi–org" installations) automatically.

Call FND SRWINIT and FND SRWEXIT

To access profile values, multiple organizations, or Oracle Applications user exits, and for your program to be used with concurrent processing at all, you must have the first and last user exits called by your Oracle Reports program be FND SRWINIT and FND SRWEXIT.

FND SRWINIT sets your profile option values and allows Oracle Application Object Library user exits to detect that they have been called by a Oracle Reports program. FND SRWEXIT ensures that all the memory allocated for Oracle Application Object Library user exits has been freed up properly. The steps below ensure that your program correctly calls FND SRWINIT and FND SRWEXIT.

Warning: With future releases of Oracle Application Object Library and Oracle Reports, we may provide a simpler set of steps to access FND SRWINIT and FND SRWEXIT. **We reserve the right to discontinue support for these steps.** If you use the steps below to integrate your Oracle Reports programs with Oracle Application Object Library, you should plan to convert to a different set of integration steps in the future.

- Create a lexical parameter P_CONC_REQUEST_ID with the datatype Number. The concurrent manager passes the concurrent request ID to your report using this parameter.
- Call FND SRWINIT in the "Before Report Trigger."

- Call FND SRWEXIT in the ”After Report Trigger.”

Calling Other Oracle Application Object Library User Exits

These integration steps let you call certain Oracle Application Object Library user exits, in addition to FND SRWINIT and FND SRWEXIT, to access user profile values and perform calculations in your Oracle Reports program:

- FND FORMAT_CURRENCY (See page 18 – 10)
- FND FLEXSQL
- FND FLEXIDVAL

Oracle Reports 2.5 Flexfield Support
Oracle Applications Flexfields Guide

Note that you can call many Oracle Applications PL/SQL routines, such as user profiles routines, from your Oracle Reports programs as well as these user exits. In general, you should use PL/SQL routines instead of user exits where possible.

You can test your Oracle Reports program that calls Oracle Applications user exits by running **ar25run**, **ar25runb**, **ar25des**, or **ar25desb** from the operating system.

Definitions of Oracle Reports Executables

ar25run This is the character–mode runtime executable that concurrent managers use to run the reports. If your character mode report calls any applications user exits, use this executable to run your report when you run it interactively.

ar25runb This is the bitmap mode runtime executable that concurrent managers use to run the reports. If your bitmap report calls any Applications user exits and you want to generate a bitmap report output (including PostScript, HTML, and PDF files), use this executable to run the report when you run it interactively. The concurrent manager uses this executable to run the reports if you select PostScript, HTML, or PDF from the Output Type poplist in the Define Concurrent Program form.

ar25desb This is the bitmap designer to build and modify reports which use Oracle Applications user exits.

Using this executable, you can create and modify reports, and test run them from within the designer. Please note that Oracle Reports supports only the bitmap designer. To design reports that will still run under character mode and produce ASCII output seamlessly, design the report in "Character Mode Emulation". In Release 11 of Oracle Applications, most of the Oracle Applications reports are designed in character mode emulation.

☞ **Attention:** The basic bitmap environment for Oracle Reports is not available on all platforms. Therefore, **ar25desb** is not linked by default by the installation process.

If the bitmap environment is available on your platform and the basic (non–applications linked) bitmap designer is available, then you should explicitly relink **ar25desb**.

User Exits Used in Oracle Reports

The user exits available in Oracle Reports are:

- FND SRWINIT
- FND SRWEXIT
- FND FORMAT_CURRENCY (See page 18 – 10)
- FND FLEXIDVAL
- FND FLEXSQL

Oracle Reports 2.5 Flexfield Support
Oracle Applications Flexfields Guide

FND SRWINIT / FND SRWEXIT

FND SRWINIT sets your profile option values and allows Oracle Application Object Library user exits to detect that they have been called by an Oracle Reports program. FND SRWINIT also allows your report to use the correct organization automatically. FND SRWEXIT ensures that all the memory allocated for Oracle Application Object Library user exits has been freed up properly.

FND FLEXIDVAL / FND FLEXSQL

These user exits allow you to use flexfields in your reports. They are documented in the *Oracle Applications Flexfields Guide*.

Oracle Reports 2.5 Flexfield Support
Oracle Applications Flexfields Guide

Using Dynamic Currency in Oracle Reports

Currency formatting support provides a flexible, consistent method to format a numeric value according to its associated currency. The currency value appears with the correct thousands separator and radix character (decimal point) of the user's country. The value appears with positive and negative indicators of the user's choice.

Displayed currency values are never rounded or truncated except when explicitly specified to be scaled. If the formatted value (which includes the thousands separator) does not fit in the output field, then the currency value without the thousands separator is used. If this value is still too large for the output field, then asterisk characters (***) are displayed in the field to alert you of an overflow condition.

You use the same methodology to add support for report regions with one currency type (for example, US dollar amounts) and for report regions with mixed currency types (for example, US dollar and Japanese yen amounts). However, when reporting on mixed currency amounts you include a special argument to indicate that you wish to align all different currency types at a standard point (usually the precision of the currency with the greatest precision). This precision is defined by a profile option or set by the user of the report at execution time.

Currency Formatting Requirements

A report based on a single currency type should display currency amounts aligned along the radix character as illustrated in the following example:

```
  Currency Value  Code
  --------------  ----
      120,300.00  USD
       –4,201.23  USD
or
   120,300.00   USD
    (4,201.23)  USD
or
     120,300.00+  USD
      –4,201.23   USD
```

If the user chooses a negative or a positive indicator such as parentheses that appears at the right of the currency amount, then values are not flushed with the right margin but are shifted to the left to accommodate the indicator.

A mixed currency report should display currency amounts aligned along the radix character (or implied radix for currencies with no precision like JPY).

```
 Currency Value  Code
 --------------  ----
    300.00       USD
    105.250      DNR
  1,000          JPY
–24,000.34       FRF
```

Call the FND FORMAT_CURRENCY user exit to format the Currency Value column. In this mixed currency report, the minimum precision (specified in the MINIMUM_PRECISION token in the user exit) is set to 3.

FND FORMAT_CURRENCY User Exit

This user exit formats the currency amount dynamically depending upon the precision of the actual currency value, the standard precision, whether the value is in a mixed currency region, the user's positive and negative format profile options, and the location (country) of the site. The location of the site determines the thousands separator and radix to use when displaying currency values. An additional profile determines whether the thousands separator is displayed.

Use the Currencies window to set the standard, extended, and minimum precision of a currency.

Defining Currencies
Oracle Applications [Public Sector] General Ledger User's Guide

You obtain the currency value from the database into an Oracle Reports column. Define another Oracle Reports column, a formula column of type CHAR, which executes the FORMAT_CURRENCY user exit to format the currency value. A displayed field has this formula column as its source so that the formatted value is automatically copied into the field for display.

Syntax

```
FND FORMAT_CURRENCY
CODE=":column containing currency code"
DISPLAY_WIDTH="field width for display"
AMOUNT=":source column name"
DISPLAY=":display column name"
[MINIMUM_PRECISION=":P_MIN_PRECISION"]
[PRECISION="{STANDARD|EXTENDED}"]
[DISPLAY_SCALING_FACTOR="":P_SCALING_FACTOR"]
```

Options:

CODE Specify the column which contains the currency code for the amount. The type of this column is CHARACTER.

DISPLAY_WIDTH Specify the width of the field in which you display the formatted amount.

AMOUNT Specify the name of the column which contains the amount retrieved from the database. This amount is of type NUMBER.

DISPLAY Specify the name of the column into which you want to display the formatted values. The type of this column is CHARACTER.

MINIMUM_PRECISION Specify the precision to which to align all currencies used in this report region. You specify the MINIMUM_PRECISION token in mixed currency report regions to ensure all currency values align at the radix character for easy readability. Your user can adjust the report by setting an input parameter when submitting the report to specifically tailor the report output to a desired minimum precision or accept the default which is determined from the profile option CURRENCY:MIXED_PRECISION (Currency:Mixed Currency Precision). Your report submission must pass the value as a report argument. You use P_MIN_PRECISION as the name of this lexical.

PRECISION If specified as STANDARD, then standard precision is used. If the value is EXTENDED then the extended precision of the currency is used to format the number.

DISPLAY_ SCALING_ FACTOR Optionally, specify the scaling factor to be applied to the amount in that column. If this token is not specified or is negative no scaling is performed. You use P_SCALING_FACTOR as the name of this lexical parameter.

Attention: Colon ":" is placed in front of column names and lexical parameters for token values. This indicates that the value of that token is retrieved from the column or lexical parameter. If it is omitted the value within double quotes itself is used. For example, CODE=":C_CODE" indicates that currency code should be retrieved from column CODE while CODE="C_CODE" indicated that currency code is C_CODE.

Mixed Currency Reports

Every report with a mixed currency region should allow the user to override the default setting of the mixed currency precision profile option at submission time. Define a report argument that accepts this value.

The default value of this argument should be the profile option CURRENCY:MIXED_PRECISION (Currency:Mixed Currency Precision) so the user does not always have to set a value explicitly.

Example Output

The following graphic illustrates various input values and the currency amounts displayed using the user exit (negative amounts are surrounded by parentheses) assuming the report was run in the United States.

```
Item  Code          Input Number  Output    Field  Notes
----  ----     -----------------  ---------------  -----
 01    USD             123456.76     123,456.76
 02    USD                 156.7         156.70
 03    USD                 12345      12,345.00
 04    BHD            123456.764     123,456.764
 05    JPY              12345676  12,345,676
 06    BHD             12345.768      12,345.768
 07    BHD            -12345.768     (12,345.768)
 08    BHD            123456.768     123,456.768
```

```
  09    BHD           –123456.768    (123,456.768)
  10    BHD            1234567.768   1,234,567.768
  11    BHD           –1234567.768  (1,234,567.768)
  12    BHD           12345678.768  12,345,678.768
  13    BHD          –12345678.768   (12345678.768)   [1]
  14    BHD          123456789.768  123,456,789.768   [2]
  15    BHD         –123456789.768  (123456789.768)
  16    BHD         1234567890.768  1234567890.768
  17    BHD        –1234567890.768  ***************   [3]
  18    BHD       12345678901.768  12345678901.768  [1,2]
  19    BHD      –12345678901.768  ***************   [3]
  20    BHD      123456789012.768  ***************   [3]
  21    USD          123456.765     123,456.765
  22    USD          123456.7654    123,456.7654   [2]
  23    USD          123456.76543   123,456.76543   [2,4]
```

```
Code  Name           Precision
USD   US dollars         2
BHD   Bahraini dinars    3
JPY   Japanese yen       0
```

```
[1] – Thousands indicators are stripped
[2] – Digit occupies space normally reserved for
            positive or negative indicator
[3] – Value cannot fit in the field:  overflow
      condition
[4] – Radix is shifted to the left due to the precision
      of the number exceeding MINIMUM_PRECISION
```

If the precision of the input number is less than the precision of the currency then the number is padded with 0’s to match the precision of the currency.

If the precision of the input number is greater than the precision of the currency then the radix of that number might get misaligned from other currency amounts in the column.

If there is one space allocated on the right for displaying the positive or negative format character (for example the profile option for displaying negative number is set to ”()” or ”<>”) and the current number does not use that space (if the number is positive) then that space is used. If this is not sufficient, then the number is left shifted to display the full precision.

If the formatted number does not fit within the DISPLAY_WIDTH then the user exit strips off all thousands separators so as to fit the number in the allowable DISPLAY_WIDTH. The amount may again be misaligned. If it still does not fit then asterisks are printed in that field to indicate an overflow condition.

Currency values are never rounded or truncated on display by default. However, the values can be scaled by a number if explicitly specified by the DISPLAY_SCALING_FACTOR token.

The tokens MINIMUM_PRECISION=":P_MIN_PRECISION" (where the lexical argument was designated as 3) and DISPLAY_WIDTH="15" apply to all items.

Items 1 through 5 show how various currencies display in a mixed currency region with a MINIMUM_PRECISION of 3. All values align along the radix character.

Items 6 through 20 depict how positive and negative values are displayed as both amounts progressively increase in magnitude (DISPLAY_WIDTH is a constant 15). When the formatted value exceeds DISPLAY_WIDTH the thousands separators are removed as in items 13, 15, 16, and 18. When the unformatted value exceeds DISPLAY_WIDTH asterisks are displayed indicating an overflow as in items 17, 19, and 20. Notice also that the positive value shifts right into the space normally reserved for the negative indicator.

Items 21 through 23 show the effects when displaying a value which exceeds the MINIMUM_PRECISION. Since the negative indicator uses a space on the right, a positive value must exceed MINIMUM_PRECISION+1 before it shifts to the left.

Example Report Using FND FORMAT_CURRENCY

The following report illustrates how various currencies are formatted using the FND FORMAT_CURRENCY user exit for a report which displays mixed currency values. This document explains how you develop such a report.

Information about the radix character and thousands separator are determined from the location of the user. The special display for negative and positive currency values is specified by two profile options. Hence, a report can appear differently depending upon the location of the user and the profile options values set.

The following reports, one run by a user in United States and the other by a user in Germany, depict this difference. In Germany the radix character and thousand separator are switched from the US counterpart. In these reports, both Manama and Seattle had a loss and the negative numbers display in parentheses () or angle brackets <> depending upon the user's preference.

Sample Report Output

Report 1 Run in The United States

Settings include:

- Information from the territory:
  - Thousand Separator: ',' (comma)
  - Radix Character: '.' (decimal)
- Profile option settings:
  - Negative Format: ()
  - Minimum Precision: 3
  - Display Thousands Separator: Yes

```
      Net Income for January 1992
      ---------------------------
Office                Net Income Currency
------          --------------- --------
Boston                12,345.00   USD
Chicago              123,456.76   USD
Manama              (23,456.764) BHD
Isa Town        12,345,678.766  BHD
Seattle             (12,345.50)  USD
Tokyo           12,345,676      JPY
```

Report 2: Run in Germany

Settings include:

- Information from the territory:
  - Thousand Separator: '.' (decimal)
  - Radix Character: ',' (comma)
- Profile option settings:
  - Negative Format: –XXX
  - Minimum Precision: 3
  - Display Thousands Separator: Yes

```
      Net Income for January 1992
      ---------------------------
Office                Net Income Currency
------          --------------- --------
Boston                 12.345,00  USD
Chicago               123.456,76  USD
Manama               –23.456,764 BHD
Isa Town         12.345.678,766 BHD
Seattle               –12.345,50  USD
Tokyo            12.345.676      JPY
```

Procedure

Step 1. Define Your Parameters

First define all the parameters (using the Oracle Reports Parameter Screen). Use these parameters in the user exit calls and SQL statements.

```
Name:           P_CONC_REQUEST_ID
Data Data Type: NUMBER
Width:          15
Initial Value:   0
```

You always create this lexical parameter. ”FND SRWINIT” uses this parameter to retrieve information about this concurrent request.

```
Name:           P_MIN_PRECISION
Data Type:      NUMBER
Width:          2
Initial Value:
```

You reference this lexical parameter in your FND FORMAT_CURRENCY user exit call.

Step 2. Call FND SRWINIT

You always call FND SRWINIT from the Before Report Trigger as follows:

```
SRW.USER_EXIT('FND SRWINIT');
```

This user exit sets up information for use by profile options and other AOL features.

You always call FND SRWEXIT from the After Report Trigger as follows:

```
     SRW.USER_EXIT('FND SRWEXIT');
```

This user exit frees all the memory allocation done in other AOL exits.

Step 3. Create the Currency Code Query

Create a query which selects the currency code and the currency amount from your table. In this case you might use:

```
SELECT OFFICE,
       SUM(AMOUNT)   C_INCOME,
       CURRENCY_CODE C_CURRENCY
```

```
FROM OFFICE_INCOME
WHERE TRANSACTION_DATE = '01/92'
ORDER BY BY OFFICE
```

Step 4. Create a column for the currency call

Create one column (C_NET_INCOME) which contains the user exit (FND FORMAT_CURRENCY) call. This is a formula column which formats the number and displays it. The user exit call looks like the following:

```
SRW.REFERENCE(:C_CURRENCY);
SRW.REFERENCE(:C_INCOME);
SRW.USER_EXIT('FND FORMAT_CURRENCY
     CODE=":C_CURRENCY"
     DISPLAY_WIDTH="15"
     AMOUNT=":C_INCOME"
     DISPLAY=":C_NET_INCOME"
     MINIMUM_PRECISION=":P_MIN_PRECISION"');
RETURN(:C_NET_INCOME);
```

Suggestion: Always reference any source column / parameter which is used as a source for data retrieval in the user exit. This guarantees that this column / parameter will contain the latest value and is achieved by "SRW.REFERENCE" call as shown above.

Here the column name containing currency code is "C_CURRENCY" and the field width of the formatted amount field is 15. The source column is "C_INCOME" and the resulting formatted output is placed in "C_NET_INCOME". The minimum precision of all the currencies used for this report is retrieved from the lexical P_MIN_PRECISION (which in this case is set to 3). At the end of the user exit call remember to reference the column "C_NET_INCOME" by RETURN(:C_NET_INCOME), otherwise the column may not contain the current information.

You do not include the MINIMUM_PRECISION token for single currency reports.

Step 5. Hide the Amount

In Default layout, unselect the amount column (C_INCOME) so that it is not displayed in the report. Do not display this amount because it contains the unformatted database column value. In the layout painter update the boiler plate text for each displayed currency field (which in this case are C_CURRENCY and C_NET_INCOME)

Attention: Repeat steps 4 and 5 for each displayed currency field.

Step 6. Create the title

In the layout painter paint the boiler plate text title as follows moving previous fields and boiler plate text as necessary:

```
Net Income for January 1992
---------------------------
```

Step 7. Define Your Report with Application Object Library

Define your report with Standard Request Submission. Ensure you define an argument P_MIN_PRECISION which defaults to $PROFILE$.MIXED_PRECISION.

The report is now ready to be run.

Summary

A brief summary of the report specifications:

Lexical Parameters:

- P_CONC_REQUEST_ID (required)
- P_MIN_PRECISION (needed for mixed currency reports)

Column Names:

- C_CURRENCY
- C_NET_INCOME

AOL User Exits:

- FND SRWINIT (required)
- FND FORMAT_CURRENCY
- FND SRWEXIT (required)

Oracle Reports Troubleshooting

This section contains tips for troubleshooting common problems and error messages.

Concurrent Request Logs

The first step of your debugging should be to examine the log of concurrent request for obvious errors.

Running from the Operating System

If you cannot determine the cause of problem from the concurrent request log, run the report from the operating system. Use the Oracle Applications linked Oracle Reports executable to run the report. Along with the standard Oracle Reports arguments, run the report with the arguments passed by the concurrent manager. The arguments passed by the concurrent manager can be found in the beginning of the concurrent request log under the title ”Arguments”. For example, the menu report can be run from the operating system as follows:

```
ar25run userid=name/password@database \
 destype=file desname=try.out \
 desformat=$FND_TOP/srw/L batch=yes \
 report=$FND_TOP/srw/FNDMNRMT \
 P_CONC_REQUEST_ID=123456 \
 ROOT_MENU=”Y” \
 MENU_APPL_ID=”0” MENU_ID=”0”
```

The P_CONC_REQUEST_ID value is a request ID for the same program that was submitted through Oracle Applications (the request could either be already completed, or could be a pending request that has been placed on hold). It is important that the request ID be for the same program (and submitted from the appropriate user and responsibility) so that the concurrent manager environment can be mimicked as closely as possible. The last three arguments (ROOT_MENU=”Y” MENU_APPL_ID=”0” MENU_ID=”0”) are arguments to the report that appeared in a concurrent request log.

If you can run the report from the command line, that indicates that there is a problem in the environment from which the concurrent manager was started. Ensure that you start the concurrent managers from the same environment in which you are trying to run the report.

Use r25run in Place of ar25run

If you cannot run **ar25run** as above, run a report without any Oracle Applications user exists using **r25run**. For this debugging step Application Object Library provides a report **$FND_TOP/srw/FNDNOEXT.rdf** (the UNIX path name) which has no user exits.

If this report fails, you may be running Oracle Reports from the wrong environment or your Oracle Reports installation may be incorrect. Contact Oracle Tools Support if **FNDNOEXT.rdf** cannot run.

Run the Print Environment Variable Values Report

The concurrent manager inherits its environment variables from the shell from which it was started and then runs reports using this environment. This environment could be different from that a a user sees logging into the Applications because the concurrent manager may have been started by a different user with different environment settings. Due to this difference, it is sometimes difficult and confusing to determine the cause of errors in running reports.

If you want to examine the value of few variables, you can run ”Prints environment variable values” report to print out the variable as seen by the concurrent manager to see if it is correct. Very often problems such as a problem in compilation or the concurrent managers inability to locate a library happen due to an incorrect REPORTS25_PATH.

Emulate Concurrent Manager Environment

For UNIX platforms, to assist in determining where the problem lies, Oracle Application Object Library ships a program called $FND_TOP/srw/a**r25run.oc**. This program helps you emulate the concurrent manager environment when testing reports from the operating system command line.

This program writes all the environment variables and arguments passed to it in a log file **ar25run.log** (located by default in the $FND_TOP/$APPLLOG directory) Save the **ar25run.oc** source code to a file named **ar25run.oc**, compile it and rename the executable as **ar25run** (referred to as ”new **ar25run**” from now on). Save $FND_TOP/bin/**ar25run** (referred to as ”old **ar25run**” from now on) into some other file and place new **ar25run** into $FND_TOP/bin. For your convenience, compiling and relinking has been incorporated into **fnd.mk** which will, by default build an executable $FND_TOP/bin/**ar25rund**, which you can then rename to **ar25run** and which would be your new **ar25run**.

In the **ar25run.oc** code, you may hardcode the directory path for **ar25run.log** (for example, change **ar25run.log** to ”/dev/fnd/6.0/log/**ar25run.log**”) to direct the log file to a more appropriate directory.

Submit the report from concurrent manager and look at the **ar25run.log**.

Then run the reports from the operating system with the same arguments as shown by **ar25run.log** using old **ar25run**.

If running with the same arguments does not help, emulate the same environment variables which **ar25run.log** shows. This can be done by printing your environment variables from the command line, sorting them and the **ar25run.log** environment variables properly, and then comparing them.

Frequently Asked Questions

Why does my report only fail from the concurrent manager?

This is because the environment from which the concurrent manager launches a report is different from the one you are using when running the report from the command line.

Why does my report show different data?

If your report shows different data when you run it as a standalone report than when you run it from the concurrent manager, you may find that you get different data from the different situations. This is usually due to different (or no) profile option or other values being passed to your report by the concurrent manager. This is especially likely if your report accesses multiple organizations data.

If you have commented out the calls to SRWINIT and SRWEXIT to test your report from a development environment that does not have the Oracle Application Object Library user exits linked in (for example, Microsoft Windows), check that you have re–enabled those calls before trying to run your report from the concurrent manager.

Why do I get the error REP–0713 when I run my report?

Oracle Reports uses a text file called uiprint.txt to hold printer names. If your printer name is not in this file, you can get the REP–0713 error.

My bitmapped report does not print in landscape. Why?

Print styles such as Landscape are associated with printer drivers that send instructions telling the printer how to print text files. However, bitmapped reports are not text files.

Bitmapped reports are output as PostScript files. The PostScript file is a set of instructions telling the printer exactly what to print and how to print it. To get a landscape report, the PostScript file must be generated as landscape.

If you want a landscape bitmapped report, specify this either in the Reports Designer or in the execution options of your concurrent program.

When printing bitmapped reports, a print style is still needed to determine the printer driver used. To avoid confusion, create a special print style for bitmapped reports and make that the required style for all bitmapped reports in the Define Concurrent Programs form.

Why do I get many pages of nonsense when I print my report?

You are looking at the PostScript code. The printer driver you are using caused the printer not to recognize the file as being PostScript. Check your driver. Some initialization strings will cause this problem. Also, do not use the program ”enscript” to do the printing.

What does the ”REP–0065: Virtual Memory System error” mean?

Unfortunately this is not a very informative error message. This could occur due to various reasons. Try the following to isolate the problem:

- Is your **/tmp** directory getting full? By default Oracle Reports uses this directory to write temporary files. These files could be directed to another directory using the environment variable TMPDIR. If you have another large partition on your machine, set TMPDIR to this partition and restart the concurrent manager from that environment.
- Are the failing reports using Page N of M? This can consume a lot of Oracle Reports virtual memory.
- If possible, try running the reports against a smaller database.

CHAPTER

19

Coding C or Pro*C Concurrent Programs

This chapter provides an overview of how to code a concurrent program in C or Pro*C. It provides utility routines you can use in your concurrent programs, along with examples.

The following topics are discussed in this chapter:

- Coding C and Pro*C Concurrent Programs
- Concurrent Processing Pro*C Utilitiy Routines

Coding C and Pro*C Concurrent Programs

This chapter describes writing a concurrent program in C or Pro*C. It includes utilities you can use along with examples of their usage.

Pro*C Concurrent Programs

When writing a program using C or Pro*C, use the Unified Concurrent Processing API templates **EXMAIN.c** and **EXPROG.c** to code your program. See your *Oracle Applications Installation Manual* for the exact location of these templates on your operating system.

Pro*C Concurrent Programs
Oracle Applications Installation Manual

Unified Concurrent Processing API afprcp()

The templates EXMAIN.c and EXPROG.c provide the correct initialization for your system for your programs. You can use concurrent program execution files written using these templates with either the spawned or immediate execution methods.

To code a custom program, copy the files to your own directory and rename them before changing them. We recommend against modifying the original templates.

EXMAIN.c is the main program which performs initialization and calls your subroutine using the **afprcp()** function. **EXPROG.c** is the subroutine which contains your application code.

Replace SUBROUTINE_NAME everywhere it appears in both files with the actual name of your subroutine. You should call **afpend()** at the end of your subroutine to clean up and return from your concurrent program. The utility **afpend()** closes the database connection, frees Application Object Library memory, closes Application Object Library files, and returns the status code you specify. You can specify one of three status codes:

- FDP_SUCCESS
- FDP_ERROR
- FDP_WARNING

The following are examples of **EXMAIN** and **EXPROG**:

EXMAIN.c

```
/*==================================================+
 |  Example MAIN for concurrent programs            |
 |  File is in $FND_TOP/usrxit/EXMAIN.c             |
 +==================================================*/

/*--------------------------------------------------+
 |  Copy this file to make a main for your          |
 |  concurrent program.  Replace SUBROUTINE_NAME    |
 |  everywhere (2 places) with the actual name of   |
 |  your concurrent program subroutine.  (This is   |
 |  the same subroutine name you register with      |
 |  Application Object Library.)                    |
 |                                                  |
 |  Do not add or modify any other statements in    |
 |  this file.                                      |
 +--------------------------------------------------*/

#ifndef AFPUB
#include <afpub.h>
#endif

#ifndef AFCP
#include <afcp.h>
#endif

AFP_FUNCS SUBROUTINE_NAME;


int main(argc, argv)
int argc;
char *argv[];
{
    afsqlopt options;

    return(afprcp(argc, argv, (afsqlopt *)NULL,
           (afpfcn *)SUBROUTINE_NAME));
}
```

EXPROG.c

```
/*==================================================+
 |  Example SUBROUTINE for concurrent programs      |
 |  File is in $FND_TOP/usrxit/EXPROG.c             |
 +==================================================*/

/*--------------------------------------------------+
 |  Copy this file to write a subroutine for your   |
 |  concurrent program.  Replace SUBROUTINE_NAME    |
 |  with the actual name of your concurrent program |
 |  (This is the same subroutine name you register  |
 |  with Application Object Library.)               |
 |                                                  |
 |  Remember to register your subroutine and        |
 |  concurrent program with Application Object      |
 |  Library and to add it to a library if you wish  |
 |  it to be run as an immediate program.           |
 |                                                  |
 |  Add your code where indicated.                  |
 |                                                  |
 |  Call afpend() to exit your subroutine.          |
 +--------------------------------------------------*/

#ifndef AFPUB
#include <afpub.h>
#endif

#ifndef AFCP
#include <afcp.h>
#endif

/*--------------------------------------------------+
 |  Add other include files you need here.          |
 |                                                  |
 |  You will need fddmsg.h if you use Message       |
 |  Dictionary.                                     |
 +--------------------------------------------------*/



boolean SUBROUTINE_NAME(argc, argv, reqinfo)
int  argc;
text  *argv[];
```

```
dvoid *reqinfo;
{

/*
 *  This is the beginning of an example program.
 * If you copied this source to create your program, delete
the lines below.
 */

  int i;
  text buffer[241];

  fdpwrt(AFWRT_LOG | AFWRT_NEWLINE, ”Hello World.”);
  fdpwrt(AFWRT_LOG | AFWRT_NEWLINE, ”Hello World.”);
  fdpwrt(AFWRT_OUT | AFWRT_NEWLINE, ”This is a test! Take
one.”);
  fdpwrt(AFWRT_OUT | AFWRT_NEWLINE, ”This is a test! Take
two.”);
  fdpwrt(AFWRT_OUT | AFWRT_NEWLINE,
”-------------------------”);


  for ( i = 0; i < argc; i++ )
  {
        sprintf(buffer, ”argv[%d]: %s”, i, argv[i]);
        fdpwrt(AFWRT_OUT | AFWRT_NEWLINE, buffer);
  }

/*
 *  This is the end of an example program.
 * If you copied this source to create your program,
 * delete the lines above.
 */



/*---------------------------------------------------+
 |  Add your code here                               |
 +---------------------------------------------------*/

/*---------------------------------------------------+
 |  Finished                                         |
 |                                                   |
 |  Always call afpend() to clean up before          |
```

```
 |  returning from your subroutine.                   |
 |                                                    |
 |   return(afpend(status, reqinfo, message));        |
 |                                                    |
 |  status is FDP_SUCCESS                             |
 |            FDP_ERROR                               |
 |            FDP_WARNING                             |
 |                                                    |
 |  message is a completion message displayed on      |
 |  the View Requests screen when your concurrent     |
 |  program/request completes.  Possible values are   |
 |  ""       for successful completion                |
 |  "text"   for text                                 |
 |  buffer   for text stored in a buffer              |
 |  afdget() for a message stored in Message          |
 |           Dictionary                               |
 +---------------------------------------------------*/


    return(afpend(FDP_SUCCESS, reqinfo, ""));
                 /* For successful completion */
}
```

Accepting Input Parameters

Use the standard argument passing method when you write a Pro*C concurrent program. This method, provided by the Unified Concurrent Processing API, retrieves the parameters you pass to your program, loads them into **arvg[]** and sets **argc**, logs onto the database, and loads the user profile option values.

If you want to make your program available through Standard Request Submission, the first parameter you define is in **argv[1]**. You do not define the parameter in **argv[0]**.

Returning From Your Pro*C Program

When your program completes, you must use Oracle Application Object Library macro **afpend()** to exit and to change the status of your concurrent request.

The macro **afpend()** logs your process off of the database, indicates to the concurrent manager whether your program was successful, and writes a completion message to your concurrent request's log file. Note that you should surround the macros with parentheses. If your program was successful, the last statement should be:

```
return(afpend(FDP_SUCCESS, reqinfo, ""));
```

The concurrent manager uses this macro to display a Completed / Normal phase / status when a user views this concurrent request in the Requests form. If you do not use **afpend()** to exit from your program and use **exit()** instead, the concurrent manager marks the request as Completed / Error.

If your program detects an error and you want to display your own error message text, the last statement should be:

```
return(afpend(FDP_ERROR, reqinfo, "error_message"));
```

Your users see a phase / status of Completed / Error on the Requests form.

If your program completes successfully, but detects some exception that needs attention, you can return a status of "WARNING" to indicate the situation. The final phase / status of your request on the Requests form is then Completed / Warning. Your last statement should be:

```
return(afpend(FDP_WARNING, reqinfo, "error_message"));
```

If your program detects an error and you want to display an error message from Message Dictionary, the last statements should be:

```
afdname(application_short_name, message_name);
return(afpend(FDP_FAILURE, reqinfo, afdget()));
```

You use the Oracle Application Object Library provided C routines **afdget()** and **afdname()** to get the error message you need from Message Dictionary.

The concurrent manager displays this error message in the Completion Text field on Request Detail window of the Requests form.

Implementing Message Dictionary (See page 12 – 5)

Naming Your Execution File

Use the appropriate file naming convention for your operating system as described in your Oracle *Applications Installation Manual*. If your operating system is case–sensitive, your file name should be in uppercase, and some operating systems require file name extensions.

The execution file name should match the compile file name of your copy of the **EXMAIN.c** program.

When you later define your spawned concurrent program executable with Oracle Application Object Library, you define your program with the same name as your file name without an extension as the executable file. For example, on Unix if you name your executable file APPOST, then you must define your concurrent program executable with the execution file field APPOST.

Pro*C Concurrent Programs
(Oracle Applications Installation Manual)

Testing Your Pro*C Program

You can run your concurrent program directly from the operating system without running it through a concurrent manager. Use this method if you want to pass arguments to your program and use default user profile options.

Syntax *PROGRAM orauser/pwd* 0 Y [*parameter1*] [*parameter2*] ...

PROGRAM	The name of your execution file containing your program. This is the name you enter in the Execution File field of the Define Concurrent Program Executable form.
orauser/pwd	The ORACLE username and password that connects you to data that your program uses.
parameter1, 2 ...	Program specific parameters. If a parameter contains spaces or double quotes, you must use a special syntax. Refer to your *Oracle Applications Installation Manual* for the syntax on your operating system. For example in Unix, you can pass a character string using ”This is an example of a \” (double quote)”.

Implementing Standard Request Submission (See page 21 – 8)

Compiling Your Immediate Concurrent Program

Once you compile all the files in your concurrent program, you can leave them as distinct object files, or put them in an object library. You can place your object files or object library in the **lib** directory under your application's top directory. For executables, you can place them in the **bin** directory under their application's top directory.

Header Files Used With Concurrent Programs

Effective with Release 10, Oracle Application Object Library established a new system of C program header files. Your spawned and immediate C programs, as well as any user exits written in C, should follow the following header conventions.

Table 19 – 1 lists the headers used with C API's in Release 11.

Header File	Comments
afpub.h	The primary header file for AOL API's. Include with all C programs accessing AOL routines.
afcp.h	The header file used with concurrent programs using the supplied C API's for concurrent processing. All Pro*C programs used in concurrent processing should include this header file.
afuxit.h	The header used with C API's used in user exits. All Pro*C user exits should include this header file.
afufld.h	The header file containing the get / put field value functions. Use this header file with **<afuxit.h>**.
fddutl.h	The header file used with message dictionary code. All Pro*C programs accessing the message dictionary feature should include this header.
fdpopt.h	The header file used to access profile options. All Pro*C programs accessing profile options should include this header.

Table 19 – 1 (Page 1 of 1)

If you have custom API's that call other header files, ensure you use the appropriate headers for the new standard.

In addition, the macro **bool** is obsolete in Release 11 and should be replaced with **boolean**.

Concurrent Processing Pro*C Utility Routines

This section describes C routines that you can use in your concurrent programs. Some of these routines are optional, and some are required and depend on the type of concurrent program you are writing. This section also includes examples of Pro*C concurrent programs.

Attention: Do not call **fdpscr()**, **fdpwrt()**, or other concurrent manager functions from user exits. The only supported interface is request submission via the PL/SQL stored procedure API, which you can code from your form.

For information on user profile C options **afpoget()** and **afpoput()**, see the User Profiles chapter.

User Profile C Functions (See page 13 – 11)

afpend()

Summary

```
#include <afcp.h>

return(afpend(status, reqinfo, message));
```

Description

Call the function **afpend()** at the end of your subroutines written with the unified concurrent processing API templates. Use afpend to clean up and return from your concurrent program with a status code and completion message. It closes the database connection, frees Application Object Library memory, closes Application Object Library files and returns the specified status code.

Return Value

This function returns TRUE if it is successful, and returns FALSE if an error occurs.

Arguments

status — The status code you want to return. Valid status codes are FDP_SUCCESS, FDP_WARNING AND FDP_ERROR.

reqinfo

message — The completion message displayed on the View Requests screen when your concurrent request completes.

Possible message values are:

”” — No content, for successful completion.

”text”	For text.
buffer	For text stored in a buffer.
afdget()	For a message stored in the Message Dictionary.

Example

```
/*   use afpend to return messages with a success code */
   char errbuf[241];

   if (!submit())
   {
       /* return failure with a message */
       return(afpend(FDP_ERROR, reqinfo,
             ”Failed in submit()”));
   }
   else if (!setprofiles())
   {
       /* return warning with a message */
       return(afpend(FDP_WARNING, reqinfo,
              ”Failed in setprofiles()”));
   }
   else if (!subrequest(argc, argv, reqinfo, errbuf))
   {
       /* return failure with a message */
       return(afpend(FDP_ERROR, reqinfo, errbuf));
   }
   else
   {
       /* Successful completion. */
       return(afpend(FDP_SUCCESS, reqinfo, ””));
   }
```

fdpfrs()

Summary

```
afreqstate fdpfrs (request_id, errbuf);
       text request_id;
       text errbuf;
```

Description

The **fdpfrs()** command returns the status of a specific request id. Us this command with the return status macros ORAF_REQUEST_XXX.

Return Value This function returns the state of the request id passed as an argument.

Arguments

request_id A null terminated string containing the request ID you want to inquire about.

errbuf A character string returned by **fdpfrs()** that contains an error message if **fdpfrs()** fails. You should declare effbuf to be size 241.

Example

```
#ifndef AFPUB
#include <afpub.h>
#endif

#ifndef AFCP
#include <afcp.h>
#endif

boolean check_request_status(request_id, errbuf)
text* request_id;
text* errbuf;

{
afreqstate request_status;

request_status = fdpfrs(request_id, errbuf);

If (ORAF_REQUEST_TEST_FAILURE(request_status) ||
   ORAF_REQUEST_NOT_FOUND(request_status))
   return FALSE;

if (ORAF_REQUEST_COMPLETE(request_status) &&
   (ORAF_REQUEST_NORMAL(request_status))
   return TRUE;

return FALSE;
}
```

fdpscp()

Summary

```
#include <afcp.h>

boolean fdpscp( argc_ptr, argv_ptr, args_method, errbuf)
int *argc_ptr;
char  **argv_ptr[];
text  args_method;
text  *errbuf;
```

Description

This function exists for compatibility with concurrent programs written with prior versions of Oracle Application Object Library. When writing new concurrent programs, use the unified concurrent processing API.

The function **fdpscp()** was called in the first statement of any spawned Pro*C concurrent program. This function retrieves the parameters your program expects, loads them into the argv[] array, and prints a standard header including the run date and time in the log file. It also logs your program onto the database. This function connects your program to the database using the ORACLE ID that is assigned to the application with which your concurrent program is defined.

Return Value

This function returns TRUE if it successfully retrieves all the parameters your concurrent request is called with. Otherwise, it returns FALSE. If this function returns FALSE, your concurrent program should print the contents of errbuf and exit with failure.

Arguments

argc_ptr — A pointer to argc, the first argument to main(). You should call **fdpscp()** using &argc.

argv_ptr — A pointer to argv, the second argument to main(). You should call **fdpscp()** using &argv.

args_method — This parameter is not currently used. You should initialize it to (text)'\0'.

errbuf — A character string returned by **fdpscp()** that contains an error message if **fdpscp()** returns FALSE. You should declare errbuf[] to be size 241.

Example

```
#include  <afcp.h>

/*  This is an example of a Pro*C concurrent program.  This
    sample program prints its input parameter to the
```

```
     log file.    */

routine()

{
   text   args_method = (text)'\0';
   text   errbuf[241];

   if (!fdpscp( &argc, &argv, args_method, errbuf ) ){
     fdpwrt( AFWRT_LOG | AFWRT_NEWLINE,
          "Error calling fdpscp" );
     fdpwrt( AFWRT_LOG | AFWRT_NEWLINE, errbuf );
     return(afpend(FDP_ERROR, reqinfo, "Failed to get
     arguments"));
   }
   if (!fdpwrt(AFWRT_LOG | AFWRT_NEWLINE, argv[1] )) {
     return(afpend(FDP_ERROR, reqinfo, "Failed to write
     arguments"));
   }
   {return(afpend(FDP_SUCCESS, reqinfo, ""));}
}
```

---

fdpscr()

Summary

```
#include <afcp.h>

boolean fdpscr( command, request_id, errbuf )
text  *command;
text  *request_id;
text  *errbuf;
```

Description

The **fdpscr()** function submits a request to run a concurrent program. You can only call this function from a Pro*C concurrent programs. The user profile options of the child request default to those of the parent concurrent program. You must commit after you call this function for your request to become eligible to be run by a concurrent manager. If you perform a rollback without having committed, your request will be lost.

Return Value If **fdpscr()** successfully submits your concurrent request, it returns TRUE. Otherwise, **fdpscr()** returns FALSE.

Arguments

command A character string that contains the parameters to your concurrent program, preceded by the word CONCURRENT. You should use the same command you use when you call a concurrent program from a form, omitting the #FND.

request_id A character string returned by **fdpscr()** that contains the request id that is assigned to your concurrent request. You should declare request_id[] to be size 12.

errbuf A character string returned by **fdpscr()** that contains an error message if **fdpscr()** returns FALSE. You should declare errbuf[] to be size 214.

Example

```
/* Submit request */
  if (!fdpscr( command, request_id, errbuf))
  {
       fdpwrt( AFWRT_LOG | AFWRT_NEWLINE,
                    ”Failed to submit concurrent request”);
       fdpwrt( AFWRT_LOG | AFWRT_NEWLINE, errbuf);

       return(FALSE);
   }
   else  /* Successful completion */
   {

       sprintf(errbuf, ”Concurrent request %s submitted
              successfully”, request_id);
       fdpwrt( AFWRT_LOG | AFWRT_NEWLINE, errbuf);

       return(TRUE);
    }
```

fdpwrt()

Summary

```
#include <fdpwrt.h>

boolean fdpwrt( flag, message)
fdcoflgs  flags
text      *message;
```

Description

The **fdpwrt()** function writes a text string to a standard log or report output file. You do not need to explicitly open and close the file. Oracle Application Object Library names your log or report output file in the standard naming convention as described in your *Oracle Applications Installation Manual*.

Return Value

The function **fdpwrt()** returns FALSE if it cannot write your message to the file that you specify. Otherwise, **fdpwrt()** returns TRUE.

Arguments

flag	A macro you use to specify what file you want to write to. AFWRT_LOG writes to a log file. AFWRT_OUT writes to a report output file. You can choose options to flush the buffer or add a newline character. Use | (bitwise OR) to turn an option on, and use &~ (bitwise AND NOT) to turn an option off. AFWRT_FLUSH flushes the buffer to the file automatically after each call. By default, AFWRT_FLUSH is on for log files and off for report output files. AFWRT_NEWLINE adds a newline character (\n) at the end of the string in the buffer before flushing the buffer. By default, AFWRT_NEWLINE is off.
message	A null–terminated character string.

Example

```
/* Submit request */
  if (!fdpscr( command, request_id, errbuf))
  {
       fdpwrt( AFWRT_LOG | AFWRT_NEWLINE,
                     ”Failed to submit concurrent request”);
       fdpwrt( AFWRT_LOG | AFWRT_NEWLINE, errbuf);
```

```
        return(FALSE);
    }
    else  /* Successful completion */
    {

        sprintf(errbuf, ”Concurrent request %s submitted
               successfully”, request_id);
        fdpwrt( AFWRT_LOG | AFWRT_NEWLINE, errbuf);

        return(TRUE);
     }
```

CHAPTER

20

Concurrent Processing APIs for PL/SQL Procedures

This chapter describes concurrent processing APIs you can use in your PL/SQL procedures. It also includes example PL/SQL code using these concurrent processing APIs.

The following concurrent processing packages are covered:

- FND_REQUEST: Concurrent Program Submission
- FND_CONCURRENT: Information on Submitted Requests
- FND_FILE: PL/SQL File I/O
- FND_PROGRAM: Concurrent Program Loaders
- FND_SET: Request Set Creation

Concurrent Processing APIs for PL/SQL Procedures

This section describes concurrent processing APIs you can use in both client–side and server–side PL/SQL procedures. This section also includes example PL/SQL procedure code using these concurrent processing APIs.

Use the FND_REQUEST package to submit a concurrent program. Use the FND_CONCURRENT package to determine or set information about your concurrent requests.

FND_REQUEST.SET_OPTIONS (Client or Server)

Syntax:

```
function FND_REQUEST.SET_OPTIONS
         (implicit  IN varchar2 default 'NO',
          protected IN varchar2 default 'NO',
          language  IN varchar2 default NULL,
          territory IN varchar2 default NULL)
          return boolean;
```

Description

Optionally call before submitting a concurrent request to set request options. Returns TRUE on successful completion, and FALSE otherwise.

Arguments (input)

implicit

Determines whether to display this concurrent request in the end–user Concurrent Requests form. (All requests are automatically displayed in the System Administrator's privileged Concurrent Requests form, regardless of the value of this argument.) Specify 'NO', 'YES', 'ERROR', or 'WARNING'.

'NO' allows the request to be viewed on the end–user Concurrent Requests form.

'YES' means that the request may be viewed only from the System Administrator's privileged Concurrent Requests form.

'ERROR' causes the request to be displayed in the end user Concurrent Requests form only if it fails.

'WARNING' allows the request to display in the end–user Concurrent Requests form only if it completes with a warning or an error.

protected — Indicates whether this concurrent request is protected against updates made using the Concurrent Requests form. 'YES' means the request is protected against updates; 'NO' means the request is not protected.

language — Indicates the NLS language. If left NULL, defaults to the current language.

territory — Indicates the language territory. If left NULL, defaults to the current language territory.

FND_REQUEST.SET_REPEAT_OPTIONS (Client or Server)

Summary

```
function FND_REQUEST.SET_REPEAT_OPTIONS
          (repeat_time      IN varchar2 default NULL,
           repeat_interval  IN number default NULL,
           repeat_unit      IN varchar2 default 'DAYS',
           repeat_type      IN varchar2 default 'START'
           repeat_end_time  IN varchar2 default NULL)
           return boolean;
```

Description

Optionally call before submitting a concurrent request to set repeat options. Returns TRUE on successful completion, and FALSE otherwise.

Arguments (input)

repeat_time — Time of day to repeat the concurrent request, formatted as HH24:MI or HH24:MI:SS. The only other parameter you may use with repeat_time is repeat_end_time.

repeat_interval — Interval between resubmissions of the request. Use this parameter along with repeat_unit to specify the time between resubmissions. This parameter applies only when repeat_time is NULL.

repeat_unit — The unit of time used along with repeat_interval to specify the time between resubmissions of the request. The available units are 'MINUTES', 'HOURS', 'DAYS', and 'MONTHS'. This parameter applies only when repeat_time is NULL.

repeat_type — Determines whether to apply the resubmission interval from either the 'START' or the 'END' of the request's execution. This parameter applies only when repeat_time is NULL.

repeat_end_time The date and time to stop resubmitting the concurrent request, formatted as either:

'DD–MON–YY HH24:MI:SS'
or
'DD–MON–YYYY HH24:MI:SS'

FND_REQUEST.SET_PRINT_OPTIONS (Client or Server)

Summary

```
function FND_REQUEST.SET_PRINT_OPTIONS
          (printer        IN varchar2 default NULL,
           style          IN varchar2 default NULL,
           copies         IN number default NULL,
           save_output    IN boolean default TRUE,
           print_together IN varchar2 default 'N')
           return boolean;
```

Description Optionally call before submitting a concurrent request to set print options. Returns TRUE on successful completion, and FALSE otherwise.

☞ **Attention:** Some print options that are set at the program level (i.e., using the Concurrent Programs form) cannot be overridden using this procedure. See the following argument descriptions to determine which print options can be overridden.

Arguments (input)

printer Name of the printer to which concurrent request output should be sent. You cannot override this print option if it was already set using the Concurrent Programs form.

style Style used to print request output, for example 'Landscape' or 'Portrait'. (Valid print styles are defined using the Print Styles form.) If the Style option was already set using the Concurrent Programs form, and the Style Required check box is checked, you cannot override this print option.

copies Number of copies of request output to print. You can override this print option even if it was already set using the Concurrent Programs form.

save_output Indicates whether to save the output file. Valid values are TRUE and FALSE. You can override this print option even if it was already set using the Concurrent Programs form.

print_together This parameter applies only to requests that contain sub–requests. 'Y' indicates that output of sub–requests should not be printed until all sub–requests complete. 'N' indicates that the output of each sub–request should be printed as it completes.

FND_REQUEST.SUBMIT_REQUEST (Client or Server)

Summary

```
function FND_REQUEST.SUBMIT_REQUEST
         (application    IN varchar2 default NULL,
          program        IN varchar2 default NULL,
          description    IN varchar2 default NULL,
          start_time     IN varchar2 default NULL,
          sub_request    IN boolean default FALSE
          argument1,
          argument2, ..., argument99,
          argument100) return number;
```

Description Submits a concurrent request for processing by a concurrent manager. If the request completes successfully, this function returns the concurrent request ID; otherwise, it returns 0.

☞ **Attention:** FND_REQUEST must know information about the user and responsibility from which the request is submitted. Therefore, this function only works from concurrent programs or forms within Oracle Applications.

The FND_REQUEST.SUBMIT_REQUEST function returns the concurrent request ID upon successful completion. It is then up to the caller to issue a commit to complete the request submission.

Your code should retrieve and handle the error message generated if there is a submission problem (the concurrent request ID returned is 0). Use FND_MESSAGE.RETRIEVE and FND_MESSAGE.ERROR to retrieve and display the error (if the request is submitted from the client side).

Overview of Message Dictionary (See page 12 – 2)

You must call FND_REQUEST.SET_MODE before calling FND_REQUEST.SUBMIT_REQUEST from a database trigger.

If FND_REQUEST.SUBMIT_REQUEST fails from anywhere but a database trigger, database changes are rolled back up to the point of the function call.

After a call to the FND_REQUEST.SUBMIT_REQUEST function, all setup parameters are reset to their default values.

Arguments (input)

application — Short name of the application associated with the concurrent request to be submitted.

program — Name of the concurrent program for which the request should be submitted.

description — Description of the request that is displayed in the Concurrent Requests form (Optional.)

start_time — Time at which the request should start running, formatted as HH24:MI or HH24:MI:SS (Optional.)

sub_request — Set to TRUE if the request is submitted from another request and should be treated as a sub–request.

☞ **Attention:** Do not use the token SUB_REQUEST if you are submitting requests from within a PL/SQL stored procedure concurrent program.

argument1...100 — Arguments for the concurrent request; up to 100 arguments are permitted. If submitted from Oracle Forms, you must specify all 100 arguments.

FND_REQUEST.SET_MODE (Server)

Summary

```
function FND_REQUEST.SET_MODE
            (db_trigger IN boolean) return boolean;
```

Description

Call this function before calling FND_REQUEST.SUBMIT_REQUEST from a database trigger.

Note that a failure in the database trigger call of FND_REQUEST.SUBMIT_REQUEST does not roll back changes.

Arguments (input)

db_trigger — Set to TRUE if request is submitted from a database trigger.

Example Request Submissions

Example 1

```
/* Submit a request from a form and commit*/
 :parameter.req_id :=
      FND_REQUEST.SUBMIT_REQUEST (
           :blockname.appsname,
           :blockname.program,
           :blockname.description,
           :blockname.start_time,
           :blockname.sub_req = 'Y',
           123, NAME_IN('ORDERS.ORDER_ID'), 'abc',
           chr(0), '', '', '', '', '', '',
           '', '', '', '', '', '', '', '', '', '',
           '', '', '', '', '', '', '', '', '', '',
           '', '', '', '', '', '', '', '', '', '',
           '', '', '', '', '', '', '', '', '', '',
           '', '', '', '', '', '', '', '', '', '',
           '', '', '', '', '', '', '', '', '', '',
           '', '', '', '', '', '', '', '', '', '',
           '', '', '', '', '', '', '', '', '', '',
           '', '', '', '', '', '', '', '', '', '');

IF :parameter.req_id = 0 THEN
    FND_MESSAGE.RETRIEVE;
    FND_MESSAGE.ERROR;
ELSE
  IF :SYSTEM.FORM_STATUS != 'CHANGED' THEN
    IF app_form.quietcommit  THEN
      /*form commits without asking user to save changes*/
     fnd_message.set_name('SQLGL',
           'GL_REQUEST_SUBMITTED');
     fnd_message.set_TOKEN('REQUEST_ID',
            TO_CHAR(:PARAMETER.REQ_ID), FALSE);
     fnd_message.show;
    ELSE
       fnd_message.set_name('FND',
            'CONC–REQUEST SUBMISSION FAILED');
       fnd_message.error;
    END IF;
  ELSE
    DO_KEY('COMMIT_FORM');
    IF :SYSTEM.FORM_STATUS != 'CHANGED' THEN
```

```
        /*commit was successful*/
        fnd_message.set_name('SQLGL',
           'GL_REQUEST_SUBMITTED');
        fnd_message.set_TOKEN('REQUEST_ID',
             TO_CHAR(:PARAMETER.REQ_ID), FALSE);
        fnd_message.show;
    END IF;
  END IF;
END IF;
```

Attention: When calling FND_REQUEST.SUBMIT_REQUEST from Oracle Forms V4.5, you must specify all 100 arguments even though they are defaulted in the function. This is a feature of PL/SQL V1. You should end your actual (in other words, non–NULL) arguments to the request with a chr(0). Thus, a call to FND_REQUEST.SUBMIT_REQUEST from Oracle Forms for a program that takes only three arguments looks like Example 1.

Example 2

```
/* Submit a request where no setup is required */
declare
  req_id number;
begin
  req_id := FND_REQUEST.SUBMIT_REQUEST ('FND',
              'FNDMDGEN', 'Message File Generator',
              '01–NOV–93 00:00:00', FALSE, ...arguments...);
  if (req_id = 0) then
    /* Handle submission error */
    FND_MESSAGE.RETRIEVE;
    FND_MESSAGE.ERROR;
  else
    commit;
  end if;
end;
```

Example 3

```
/* Submit a request from a database trigger */
result := FND_REQUEST.SET_MODE(TRUE);
req_id := FND_REQUEST.SUBMIT_REQUEST (FND',
            'FNDMDGEN', 'Message File  Generator',
```

```
                         '01-NOV-93 00:00:00', FALSE, ...arguments...);
```

Example 4

```
/* Submit a request inserting NULL arguments.
   This call inserts 6 arguments with arguments 1, 3,
   4, and 6 being NULL */
req_id := FND_REQUEST.SUBMIT_REQUEST ('FND',
                        'FNDPROG',
                        'Description of FNDPROG',
                        '01-FEB-94 00:00:00', FALSE,
                        '', 'arg2', '', NULL, arg5, '');
```

Example 5

```
/* Submit a repeating request */
result := FND_REQUEST.SET_REPEAT_OPTIONS ('', 4, 'HOURS',
'END');
req_id := FND_REQUEST.SUBMIT_REQUEST ('CUS',
                          'CUSPOST', 'Custom Posting',
                          '01-APR-94 00:00:00', FALSE,
                          ...arguments...);
```

Attention: You may not want to submit a request if FND_REQUEST.SET_REPEAT_OPTIONS returns failure. Thus, you may wish to test the result of FND_REQUEST.SET_REPEAT_OPTIONS before issuing the call to FND_REQUEST.SUBMIT_REQUEST.

Example 6

```
/* Submit a request for 5 copies of a menu report */
result := FND_REQUEST.SET_PRINT_OPTIONS ('hqunx138',
                                        'Landscape',
                                         5,
                                        'Yes',
                                        FALSE);
req_id := FND_REQUEST.SUBMIT_REQUEST ('FND',
                                'FNDMNRMT',
                                '',
                                '',
                               'N', 0, 101);
```

Example 7

```
/* Submit a protected request that repeats at noon */
result := FND_REQUEST.SET_OPTIONS ('YES');
result := FND_REQUEST.SET_REPEAT_OPTIONS ('12:00');
req_id := FND_REQUEST.SUBMIT_REQUEST ('CUS',
                            'CUSPOST', 'Custom Posting',
                            '01–APR–94 00:00:00', FALSE,
                             ... args ...);
```

FND_CONCURRENT.GET_REQUEST_STATUS (Client or Server)

Summary

```
function FND_CONCURRENT.GET_REQUEST_STATUS
       (request_id      IN OUT number,
        application     IN varchar2 default NULL,
         program        IN varchar2 default NULL,
         phase          OUT varchar2,
         status         OUT varchar2,
         dev_phase      OUT varchar2,
         dev_status     OUT varchar2,
         message        OUT varchar2) return boolean;
```

Description

Returns the status of a concurrent request. If the request has already completed, also returns a completion message.

FND_CONCURRENT.GET_REQUEST_STATUS returns both "user–friendly" (i.e., translatable) phase and status values, as well as "developer" phase and status values that can drive program logic.

Arguments (input)

request_id The request ID of the program to be checked.

application Short name of the application associated with the concurrent program. This parameter is necessary only when the request_id is *not* specified.

program Name of the concurrent program. This parameter is necessary only when the request_id is *not* specified. When application and program are provided, the request ID of the last request for this program is returned in request_id.

Arguments (output)

phase The user friendly request phase from FND_LOOKUPS.

status The user friendly request status from FND_LOOKUPS.

dev_phase The request phase as a constant string that can be used for program logic comparisons.

dev_status The request status as a constant string that can be used for program logic comparisons.

message The completion message supplied if the request has completed.

Example

```
begin

        call_status boolean;
        rphase      varchar2(30);
        rstatus     varchar2(30);
        dphase      varchar2(30);
        dstatus     varchar2(30);
        message     varchar2(240);

        call_status :=

FND_CONCURRENT.GET_REQUEST_STATUS(<Request_ID>, '', '',
                  rphase,rstatus,dphase,dstatus, message);
    end;
```

In the above example, rphase and rstatus receive the same phase and status values as are displayed on the Concurrent Requests form. The completion text of a completed request returns in message.

Any developer who wishes to control the flow of a program based on a request's outcome should use the following values to compare the request's phase and status.

Possible values for dev_phase and dev_status are:

dev_phase	dev_status	Description
PENDING	NORMAL	Request is waiting for the next available manager.
	STANDBY	A constrained request (i.e. incompatible with currently running or actively pending programs) is waiting for the Internal concurrent manager to release it.
	SCHEDULED	Request is scheduled to start at a future time or date.

dev_phase	dev_status	Description
	PAUSED	Child request is waiting for its Parent request to mark it ready to run. For example, a report in a report set that runs sequentially must wait for a prior report to complete.
RUNNING	NORMAL	Request is being processed.
	WAITING	Parent request is waiting for its sub–requests to complete.
	RESUMING	Parent request is waiting to restart after its sub–requests have completed.
	TERMINATING	A user has requested to terminate this running request.
COMPLETE	NORMAL	Request completed successfully.
	ERROR	Request failed to complete successfully.
	WARNING	Request completed with warnings. For example, a report is generated successfully but failed to print.
	CANCELLED	Pending or Inactive request was cancelled.
	TERMINATED	Running request was terminated.
INACTIVE	DISABLED	Concurrent program associated with the request is disabled.
	ON_HOLD	Pending request placed on hold.
	NO_MANAGER	No manager is defined to run the request.
	SUSPENDED	This value is included for upward compatibility. It indicates that a user has paused the request at the OS level.

Table 20 – 1 (Page 2 of 2)

FND_CONCURRENT.WAIT_FOR_REQUEST (Client or Server)

Summary

```
function FND_CONCURRENT.WAIT_FOR_REQUEST
  (request_id IN number default NULL,
          interval   IN number default 60,
          max_wait   IN number default 0,
```

```
phase      OUT varchar2,
status     OUT varchar2,
dev_phase  OUT varchar2,
dev_status OUT varchar2,
message    OUT varchar2) return  boolean;
```

Description Waits for request completion, then returns the request phase/status and completion message to the caller. Goes to sleep between checks for request completion.

Arguments (input)

request_id The request ID of the request to wait on.

interval Number of seconds to wait between checks (i.e., number of seconds to sleep.)

max_wait The maximum time in seconds to wait for the request's completion.

Arguments (output)

phase The user–friendly request phase from the FND_LOOKUPS table.

status The user–friendly request status from the FND_LOOKUPS table.

dev_phase The request phase as a constant string that can be used for program logic comparisons.

dev_status The request status as a constant string that can be used for program logic comparisons.

message The completion message supplied if the request has already completed.

FND_CONCURRENT.SET_COMPLETION_STATUS (Server)

Summary

```
function FND_CONCURRENT.SET_COMPLETION_STATUS
  (status       IN           varchar2,
    message     IN           varchar2) return  boolean;
```

Description Call SET_COMPLETION_STATUS from a concurrent program to set its completion status. The function returns TRUE on success, otherwise FALSE.

Arguments (input)	**status**	The status to set the concurrent program to. Either NORMAL, WARNING, or ERROR.
	message	An optional message.

FND_FILE: PL/SQL File I/O

The FND_FILE stored package contains procedures to write text to log and output files. These procedures can only be used within PL/SQL procedures that are concurrent programs.

For testing and debugging, you can use the procedures FND_FILE.PUT_NAMES and FND_FILE.CLOSE. Note that these two procedures should not be called from a concurrent program.

☞ **Attention:** This package is not designed for generic PL/SQL text I/O, but rather only for writing to request log and output files.

PL/SQL File I/O Processing (See page 17 – 3)

FND_FILE.PUT

Summary

```
procedure FND_FILE.PUT
    (which          IN          NUMBER,
     buff           IN          VARCHAR2);
```

Description

Writes text to a file (without a new line character). Multiple calls to FND_FILE.PUT will produce concatenated text. Typically used with FND_FILE.NEW_LINE.

Arguments (input)

which Log file or output file. Use either FND_FILE.LOG or FND_FILE.OUTPUT.

buff Text to write.

FND_FILE.PUT_LINE

Summary

```
procedure FND_FILE.PUT_LINE
    (which          IN          NUMBER,
     buff           IN          VARCHAR2);
```

Description

Writes a line of text to a file (followed by a new line character). You will use this utility most often.

Arguments (input)

which Log file or output file. Use either FND_FILE.LOG or FND_FILE.OUTPUT.

buff Text to write.

Example Using Message Dictionary to retrieve a message already set up on the server and putting it in the log file (allows the log file to contain a translated message):

```
FND_FILE.PUT_LINE( FND_FILE.LOG, fnd_message.get );
```

Putting a line of text in the log file directly (message cannot be translated because it is hardcoded in English; not recommended):

```
fnd_file.put_line(FND_FILE.LOG,'Warning: Employee '||
                  l_log_employee_name||' ('||
                  l_log_employee_num ||
                       ') does not have a manager.');
```

FND_FILE.NEW_LINE

Summary

```
procedure FND_FILE.NEW_LINE
    (which            IN          NUMBER,
     LINES            IN          NATURAL := 1);
```

Description Writes line terminators (new line characters) to a file.

Arguments (input)

which Log file or output file. Use either FND_FILE.LOG or FND_FILE.OUTPUT.

lines Number of line terminators to write.

Example To write two new line characters:

```
fnd_file.new_line(FND_FILE.LOG,2);
```

FND_FILE.PUT_NAMES

Summary

```
procedure FND_FILE.PUT_NAMES
    (p_log            IN          VARCHAR2,
     p_out            IN          VARCHAR2,
    (p_dir            IN          VARCHAR2);
```

Description Sets the temporary log and out filenames and the temp directory to the user–specified values. DIR must be a directory to which the database can write. FND_FILE.PUT_NAMES should be called before calling any other FND_FILE function, and only once per session.

☞ **Attention:** FND_FILE.PUT_NAMES is meant for testing and debugging from SQL*Plus; it does nothing if called from a concurrent program.

Arguments (input)

p_log	Temporary log filename.
p_out	Temporary output filename.
p_dir	Temporary directory name.

Example

```
BEGIN
   fnd_file.put_names('test.log', 'test.out',
     '/local/db/8.0.4/db-temp-dir/');
   fnd_file.put_line(fnd_file.output,'Called stored
     procedure');
   /* Some logic here... */
   fnd_file.put_line(fnd_file.output, 'Reached point A');
   /* More logic, etc... */
   fnd_file.close;
END;
```

FND_FILE.CLOSE

Summary `procedure FND_FILE.CLOSE;`

Description Closes open files.

☞ **Attention:** Use FND_FILE.CLOSE only in command lines sessions. FND_FILE.CLOSE should not be called from a concurrent program.

Example

```
BEGIN
   fnd_file.put_names('test.log', 'test.out',
     '/local/db/8.0.4/db-temp-dir/');
   fnd_file.put_line(fnd_file.output,'Called stored
     procedure');
   /* Some logic here... */
   fnd_file.put_line(fnd_file.output, 'Reached point A');
   /* More logic, etc... */
```

```
   fnd_file.close;
END;
```

FND_PROGRAM: Concurrent Program Loaders

The FND_PROGRAM package includes procedures for creating concurrent program executables, concurrent programs with parameters and incompatibility rules, request sets, and request groups. The FND_PROGRAM package also contains functions you can use to check for the existence of concurrent programs, executables, parameters, and incompatibility rules.

The arguments passed to the procedures correspond to the fields in the Oracle Application Object Library Release 11 forms, with minor exceptions. In general, first enter the parameters to these procedures into the forms for validation and debugging.

If an error is detected, ORA–06501: PL/SQL: internal error is raised. The error message can be retrieved by a call to the function fnd_program.message().

Some errors are not trapped by the package, notably ”duplicate value on index”.

Note that an exception is raised if bad foreign key information is provided. For example, delete_program() does not fail if the program does not exist, but does fail if given a bad application name.

FND_PROGRAM.MESSAGE

Summary

```
function FND_PROGRAM.MESSAGE return VARCHAR2;
```

Description

Use the message function to return an error message. Messages are set when any validation (program) errors occur.

FND_PROGRAM.EXECUTABLE

Summary

```
procedure FND_PROGRAM.EXECUTABLE
   (executable          IN          VARCHAR2,
    application         IN          VARCHAR2,
    description         IN          VARCHAR2 DEFAULT NULL,
    execution_method    IN          VARCHAR2,
    execution_file_name IN          VARCHAR2 DEFAULT NULL,
    subroutine_name     IN          VARCHAR2 DEFAULT NULL,
    icon_name           IN          VARCHAR2 DEFAULT NULL,
```

```
language_code       IN          VARCHAR2 DEFAULT 'US');
```

Description Define a concurrent program executable. This procedure corresponds to the "Concurrent Program Executable" window accessible from the System Administrator and Application Developer responsibilities.

Arguments (input)

executable Name of executable (for example, 'FNDSCRMT').

application The short name of the executable's application, e.g. 'FND'.

description Optional description of the executable.

execution_ method The type of program this executable uses. Possible values are 'Host', 'Immediate', 'Oracle Reports', 'PL/SQL Stored Procedure', 'Spawned', 'SQL*Loader', 'SQL*Plus'. The SQL*Report, FlexRPT, FlexSQL, and Immediate execution methods are currently not supported.

execution_ file_name The operating system name of the file. Required for all but Immediate programs. This file name should not include spaces or periods unless the file is a PL/SQL stored procedure.

subroutine_name Used only by Immediate programs. Cannot contain spaces or periods.

icon_name Reserved for future use by internal developers only. Specify NULL.

language_code Language code for the name and description, e.g. 'US'.

FND_PROGRAM.DELETE_EXECUTABLE

Summary

```
procedure FND_PROGRAM.DELETE_EXECUTABLE
   (executable         IN          varchar2,
   application         IN          varchar2);
```

Description Delete a concurrent program executable. An executable that is assigned to a concurrent program cannot be deleted.

Arguments (input)

executable The short name of the executable to delete.

application The short name of the executable's application, e.g. 'FND'.

FND_PROGRAM.REGISTER

Summary

```
procedure FND_PROGRAM.REGISTER
      (program                  IN VARCHAR2,
      application               IN VARCHAR2,
      enabled                   IN VARCHAR2,
      short_name                IN VARCHAR2,
      description               IN VARCHAR2 DEFAULT NULL,
      executable_name           IN VARCHAR2,
      executable_application    IN VARCHAR2,
      execution_options         IN VARCHAR2 DEFAULT NULL,
      priority                  IN NUMBER   DEFAULT NULL,
      save_output               IN VARCHAR2 DEFAULT 'Y',
      print                     IN VARCHAR2 DEFAULT 'Y',
      cols                      IN NUMBER   DEFAULT NULL,
      rows                      IN NUMBER   DEFAULT NULL,
      style                     IN VARCHAR2 DEFAULT NULL,
      style_required            IN VARCHAR2 DEFAULT 'N',
      printer                   IN VARCHAR2 DEFAULT NULL,
      request_type              IN VARCHAR2 DEFAULT NULL,
      request_type_application IN VARCHAR2 DEFAULT NULL,
      use_in_srs                IN VARCHAR2 DEFAULT 'N',
      allow_disabled_values     IN VARCHAR2 DEFAULT 'N',
      run_alone                 IN VARCHAR2 DEFAULT 'N',
      output_type               IN VARCHAR2 DEFAULT 'TEXT',
      enable_trace              IN VARCHAR2 DEFAULT 'N',
      restart                   IN VARCHAR2 DEFAULT 'Y',
      nls_compliant             IN VARCHAR2 DEFAULT 'N',
      icon_name                 IN VARCHAR2 DEFAULT NULL,
      language_code             IN VARCHAR2 DEFAULT 'US');
```

Description Use this procedure to define a concurrent program. This procedure corresponds to the "Concurrent Program" window accessible from the System Administrator and Application Developer responsibilities.

Arguments (input)

program The user–visible program name, e.g. 'Menu Report'.

application	The short name of the application that owns the program. The program application determines the Oracle user name used by the program.
enabled	Specify either ”Y” or ”N”.
short_name	The internal developer program name.
description	An optional description of the program.
executable_name	The short name of the registered concurrent program executable.
executable_ application	The short name of the application under which the executable is registered.
execution_ options	Any special option string, used by certain executables such as Oracle Reports.
priority	An optional program level priority.
save_output	Indicate with ”Y” or ”N” whether to save the output.
print	Allow printing by specifying ”Y”, otherwise ”N”.
cols	The page width of report columns.
rows	The page length of report rows.
style	The default print style name.
style_required	Specify whether to allow changing the default print style from the Submit Requests window.
printer	Force output to the specified printer.
request_type	A user–defined request type.
request_type_ application	The short name of the application owning the request type.
use_in_srs	Specify ”Y” to allow users to submit the program from the Submit Requests window, otherwise ”N”.
allow_ disabled_values	Specify ”Y” to allow parameters based on outdated value sets to validate anyway. Specify ”N” to require current values.
run_alone	Program must have the whole system to itself. (”Y” or ”N”)
output_type	The type of output generated by the concurrent program. Either ”HTML”, ”PS”, ”TEXT” or ”PDF”.

enable_trace Specify "Y" if you want to always enable SQL trace for this program, "N" if not.

nls_compliant Reserved for future use by internal developers only. Use "N".

icon_name Reserved for future use by internal developers only. Use NULL.

language_code Language code for the name and description.

FND_PROGRAM.DELETE_PROGRAM

Summary

```
procedure FND_PROGRAM.DELETE_PROGRAM

  (program_short_name   IN           varchar2,
    application         IN           varchar2);
```

Description Use this procedure to delete a concurrent program. All references to the program are deleted as well.

Arguments (input)

program_short_name The short name used as the developer name of the concurrent program.

application The application that owns the concurrent program.

FND_PROGRAM.PARAMETER

Summary

```
procedure FND_PROGRAM.PARAMETER

 (program_short_name          IN     VARCHAR2,
   application                IN     VARCHAR2,
   sequence                   IN     NUMBER,
   parameter                  IN     VARCHAR2,
   description                IN     VARCHAR2 DEFAULT NULL,
   enabled                    IN     VARCHAR2 DEFAULT 'Y',
   value_set                  IN     VARCHAR2,
   default_type               IN     VARCHAR2 DEFAULT NULL,
   default_value              IN     VARCHAR2 DEFAULT NULL,
   required                   IN     VARCHAR2 DEFAULT 'N',
   enable_security            IN     VARCHAR2 DEFAULT 'N',
   range                      IN     VARCHAR2 DEFAULT NULL,
   display                    IN     VARCHAR2 DEFAULT 'Y',
```

```
display_size                IN    NUMBER,
description_size            IN    NUMBER,
concatenated_description_size IN NUMBER,
prompt                      IN    VARCHAR2 DEFAULT NULL,
token                       IN    VARCHAR2 DEFAULT NULL);
```

Description

Creates a new parameter for a specified concurrent program. This procedure corresponds to the "Concurrent Program Parameters" window accessible from the System Administrator and Application Developer responsibilities.

☞ **Attention:** A newly added parameter does not show up in the SRS form until the descriptive flexfields are compiled. The program $FND_TOP/$APPLBIN/fdfcmp compiles the descriptive flexfields.

Arguments (input)

program_short_ name	The short name used as the developer name of the concurrent program.
application	The short name of the application that owns the concurrent program.
sequence	The parameter sequence number that determines the order of the parameters.
parameter	The parameter name.
description	An optional parameter description.
enabled	"Y" for enabled parameters; "N" for disabled parameters.
value_set	The value set to use with this parameter.
default_type	An optional default type. Possible values are 'Constant', 'Profile', 'SQL Statement', or 'Segment'.
default_value	Only required if the default_type is not NULL.
required	"Y" for required parameters, "N" for optional ones.
enable_security	"Y" enables value security if the value set permits it. "N" prevents value security from operating on this parameter.
range	Optionally specify "High", "Low", or "Pair".
display	"Y" to display the parameter, "N" to hide it.
display_size	The length of the item in the parameter window.

description_size The length of the item's description in the parameter window.

concatenated_ description_size The Length of the description in the concatenated parameters field.

prompt The item prompt in the parameter window.

token The Oracle Reports token (only used with Oracle Reports programs).

FND_PROGRAM.DELETE_PARAMETER

Summary

```
procedure FND_PROGRAM.DELETE_PARAMETER
  (program_short_name  IN           varchar2,
    application        IN           varchar2
    parameter          IN           varchar2);
```

Description Call this procedure to remove a parameter from a concurrent program.

Arguments (input)

program_short_ name The short name used as the developer name of the concurrent program.

application The application that owns the concurrent program.

parameter The parameter to delete.

FND_PROGRAM.INCOMPATIBILITY

Summary

```
procedure FND_PROGRAM.INCOMPATIBILITY
  (program_short_name  IN     VARCHAR2,
   application         IN     VARCHAR2
   inc_prog_short_name IN     VARCHAR2,
   inc_prog_applicationIN     VARCHAR2,
   scope               IN     VARCHAR2 DEFAULT 'Set');
```

Description Use this procedure to register an incompatibility for a specified concurrent program. This procedure corresponds to the "Incompatible Programs" window accessible from the System Administrator and Application Developer responsibilities.

Arguments (input)

program_short_ name The short name used as the developer name of the concurrent program.

application	The short name of the application that owns the concurrent program
inc_prog_ short_name	The short name of the incompatible program.
inc_prog_ application	Application that owns the incompatible program.
scope	Either ”Set” or ”Program Only”

FND_PROGRAM.DELETE_INCOMPATIBILITY

Summary

```
procedure FND_PROGRAM.DELETE_INCOMPATIBILITY
  (program_short_name          IN      VARCHAR2,
     application               IN      VARCHAR2,
     inc_prog_short_name       IN      VARCHAR2,
     inc_prog_application      IN      VARCHAR2);
```

Description Delete a concurrent program incompatibility rule.

Arguments (input)

program_short_ name	The short name used as the developer name of the concurrent program.
application	Application that owns the concurrent program
inc_prog_ short_name	Short name of the incompatible program to delete.
inc_prog_ application	Application that owns the incompatible program.

FND_PROGRAM.REQUEST_GROUP

Summary

```
procedure FND_PROGRAM.REQUEST_GROUP
   (request_group               IN VARCHAR2,
   application                  IN VARCHAR2,
   code                         IN VARCHAR2 DEFAULT NULL,
   description                  IN VARCHAR2 DEFAULT NULL);
```

Description Creates a new request group. This procedure corresponds to the master region of the ”Request Groups” window in the System Administration responsibility.

Arguments (input)

request_group	The name of the request group
application	The application that owns the request group.
code	An optional code for the request group.
description	An optional description of the request group.

FND_PROGRAM.DELETE_GROUP

Summary

```
procedure FND_PROGRAM.DELETE_GROUP
    (group              IN            VARCHAR2,
    application         IN            VARCHAR2);
```

Description Delete a request group.

Arguments (input)

request_group	The name of the request group to delete.
application	The application that owns the request group.

FND_PROGRAM.ADD_TO_GROUP

Summary

```
procedure FND_PROGRAM.ADD_TO_GROUP
  (program_short_name          IN     VARCHAR2,
    program_application        IN     VARCHAR2,
    request_group              IN     VARCHAR2,
    group_application          IN     VARCHAR2);
```

Description Use this procedure to add a concurrent program to a request group. This procedure corresponds to the ”Requests” region in the ”Request Groups” window in System Administration.

Arguments (input)

program_short_name	The short name used as the developer name of the concurrent program.
program_application	The application that owns the concurrent program.
request_group	The request group to which to add the concurrent program.
group_application	The application that owns the request group.

FND_PROGRAM.REMOVE_FROM_GROUP

Summary

```
procedure FND_PROGRAM.REMOVE_FROM_GROUP
   (program_short_name  IN          VARCHAR2,
    program_application IN          VARCHAR2,
    request_group       IN          VARCHAR2,
    group_application   IN          VARCHAR2);
```

Description

Use this procedure to remove a concurrent program from a request group.

Arguments (input)

program_short_name — The short name used as the developer name of the concurrent program.

program_application — The application that owns the concurrent program.

request_group — The request group from which to delete the concurrent program.

group_application — The application that owns the request group.

FND_PROGRAM.PROGRAM_EXISTS

Summary

```
function FND_PROGRAM.PROGRAM_EXISTS
   (program     IN          VARCHAR2,
   application  IN          VARCHAR2)
return boolean;
```

Description

Return TRUE if a concurrent program exists.

Arguments (input)

program — The short name of the program

application — Application short name of the program.

FND_PROGRAM.PARAMETER_EXISTS

Summary

```
function FND_PROGRAM.PARAMETER_EXISTS
   (program_short_name IN    VARCHAR2,
   application         IN    VARCHAR2,
```

```
   parameter          IN     VARCHAR2)
return boolean;
```

Description Return TRUE if a program parameter exists.

Arguments (input)

program	The short name of the program
application	Application short name of the program.
parameter	Name of the parameter.

FND_PROGRAM.INCOMPATIBILITY_EXISTS

Summary

```
function FND_PROGRAM.INCOMPATIBILITY_EXISTS
   (program_short_name IN     VARCHAR2,
   application        IN     VARCHAR2,
   inc_prog_short_name IN     VARCHAR2,
   inc_prog_applicationIN     VARCHAR2)
return boolean;
```

Description Return TRUE if a program incompatibility exists.

Arguments (input)

program	The short name of the first program
application	Application short name of the program.
inc_prog_short_ name	Short name of the incompatible program.
inc_prog_ applicatoin	Application short name of the incompatible program.

FND_PROGRAM.EXECUTABLE_EXISTS

Summary

```
function FND_PROGRAM.EXECUTABLE_EXISTS
   (executable_short_name     IN     VARCHAR2,
   application               IN     VARCHAR2)
return boolean;
```

Description Return TRUE if program executable exists.

Arguments (input)

program — The name of the executable.

application — Application short name of the executable..

FND_PROGRAM.REQUEST_GROUP_EXISTS

Summary

```
function FND_PROGRAM.REQUEST_GROUP_EXISTS
   (request_group      IN     VARCHAR2,
   application   IN     VARCHAR2)
return boolean;
```

Description Return TRUE if requst group exists.

Arguments (input)

program — The name of the executable.

application — Application short name of the request group..

FND_PROGRAM.PROGRAM_IN_GROUP

Summary

```
function FND_PROGRAM.INCOMPATIBILITY_EXISTS
   (program_short_name IN     VARCHAR2,
   application         IN     VARCHAR2,
   request_group       IN     VARCHAR2,
   group_application   IN     VARCHAR2)
return boolean;
```

Description Return TRUE if a program is in a request group.

Arguments (input)

program — The short name of the first program

application — Application short name of the program.

request_group — Name of the request group.

group_ application — Application short name of the request group.

FND_SET: Request Set Loaders

The FND_SET package includes procedures for creating concurrent program request sets, adding programs to a request set, deleting programs from a request set. and defining parameters for request sets.

The arguments passed to the procedures correspond to the fields in the Oracle Application Object Library Release 11 forms, with minor exceptions. In general, first enter the parameters to these procedures into the forms for validation and debugging.

If an error is detected, ORA–06501: PL/SQL: internal error is raised. The error message can be retrieved by a call to the function fnd_program.message().

Some errors are not trapped by the package, notably "duplicate value on index".

Note that an exception is raised if bad foreign key information is provided. For example, delete_program() does not fail if the program does not exist, but does fail if given a bad application name.

Overview of Request Sets (See page 22 – 2)

FND_SET.MESSAGE

Summary

```
function FND_SET.MESSAGE return VARCHAR2;
```

Description

Use the message function to return an error message. Messages are set when any validation (program) errors occur.

FND_SET.CREATE_SET

Summary

```
procedure FND_SET.CREATE_SET
    (name                       IN VARCHAR2,
    short_name                  IN VARCHAR2,
    application                 IN VARCHAR2,
    description                 IN VARCHAR2 DEFAULT NULL,
    owner                       IN VARCHAR2 DEFAULT NULL,
    start_date                  IN DATE     DEFAULT SYSDATE,
    end_date                    IN DATE     DEFAULT NULL,
    print_together              IN VARCHAR2 DEFAULT 'N',
```

```
incompatibilities_allowed  IN VARCHAR2 DEFAULT 'N',
language_code              IN VARCHAR2 DEFAULT 'US');
```

Description Use this procedure to register a Request Set. This procedure corresponds to the master region of the "Request Set" window.

Arguments (input)

name The name of the new request set.

short_name The short name for the request set.

application The application that owns the request set.

description An optional description of the set.

owner An optional Oracle Applications user ID identifying the set owner, e.g. SYSADMIN.

start_date The date the set becomes effective.

end_date An optional date on which the set becomes outdated.

print_together Specify "Y" or "N" to indication whether all the reports in a set should print at the same time.

incompatibilities_allowed Specify "Y" or "N" to indicate whether to allow incompatibilities for this set.

language_code Language code for the above data, e.g. "US".

FND_SET.DELETE_SET

Summary

```
procedure FND_SET.DELETE_SET
   (request_set                IN VARCHAR2,
   application                 IN VARCHAR2);
```

Description Delete a request set and references to that set.

Arguments (input)

request_set The short name of the request set to delete.

application The application that owns the request set.

FND_SET.ADD_PROGRAM

Summary

```
procedure FND_SET.ADD_PROGRAM
```

```
(program                        IN VARCHAR2,
program_application             IN VARCHAR2,
request_set                     IN VARCHAR2,
set_application                 IN VARCHAR2,
stage                           IN VARCHAR2,
program_sequence                IN NUMBER,
critical                        IN VARCHAR2 DEFAULT 'Y',
number_of_copies                IN NUMBER   DEFAULT 0,
save_output                     IN VARCHAR2 DEFAULT 'Y',
style                           IN VARCHAR2 DEFAULT NULL,
printer                         IN VARCHAR2 DEFAULT NULL);
```

Description Add a concurrent program to a request set stage. This procedure corresponds to the "Programs" region in the "Stage Requests" window of the "Request Set" form.

Arguments (input)

program_short_ name	The short name used as the developer name of the concurrent program, e.g. 'FNDSCRMT'.
program_ application	The short name of the application that owns the concurrent program.
request_set	The short name of the request set.
set_application	The application that owns the request set.
stage	The short name of the stage.
program_ sequence	The sequence number of this program in the stage. All programs in a stage require a unique sequence number.
critical	Specify 'Y' if this program can affect the stage's outcome, and 'N' if not.
number_of_ copies	An optional default for the number of copies to print.
save_output	Specify 'Y' to allow users to save output, or 'N' if the default is not to save the output.
style	Optionally provide a default print style.
printer	Optionally provide a default printer.

FND_SET.REMOVE_PROGRAM

Summary `procedure FND_SET.REMOVE_PROGRAM`

```
(program_short_name          IN VARCHAR2,
program_application          IN VARCHAR2,
request_set                  IN VARCHAR2,
set_application              IN VARCHAR2,
stage                        IN VARCHAR2,
program_sequence             IN NUMBER);
```

Description Remove a concurrent program from a request set.

Arguments (input)

program_short_ name	The short name used as the developer name of the concurrent program.
program_ application	The short name of the application that owns the concurrent program.
request_set	The short name of the request set.
set_application	The short name of the application that owns the request set.
program_ sequence	The sequence number of this program in the stage. All programs in a stage require a unique sequence number.

FND_SET.PROGRAM_PARAMETER

Summary `procedure FND_SET.PROGRAM_PARAMETER`

```
(program                     IN VARCHAR2,
program_application          IN VARCHAR2,
request_set                  IN VARCHAR2,
set_application              IN VARCHAR2,
stage                        IN VARCHAR2.
program_sequence             IN NUMBER,
parameter                    IN VARCHAR2,
display                      IN VARCHAR2 DEFAULT ’Y’,
modify                       IN VARCHAR2 DEFAULT ’Y’,
shared_parameter             IN VARCHAR2 DEFAULT NULL,
default_type                 IN VARCHAR2 DEFAULT NULL,
default_value                IN VARCHAR2 DEFAULT NULL);
```

Description This procedure registers shared parameter information and the request set level overrides of program parameter attributes. This procedure corresponds to the ”Request Parameters” window of the ”Request Sets” form.

Arguments (input)

program	The short name used as the developer name of the concurrent program.
program_ application	The short name of the application that owns the concurrent program.
request_set	The short name of the request set.
set_application	The short name of the application that owns the request set.
program_ sequence	The sequence number of this program in the stage.
parameter	The name of the program parameter.
display	”Y” to display the parameter, ”N” to hide it.
modify	”Y” to allow users to modify the parameter value, ”N” to prevent it.
shared_parameter	If the parameter uses a shared parameter, enter the shared parameter name here.
default_type	If the parameter uses a default, enter the type here. Valid types are ’Constant’, ’Profile’, ’SQL Statement’, or ’Segment’.
default_value	If the parameter uses a default, enter a value appropriate for the default type here. This argument is required if default_type is not null.

FND_SET.DELETE_PROGRAM_PARAMETER

Summary

```
procedure FND_SET.DELETE_SET_PARAMETER
    program                       IN VARCHAR2,
    program_application           IN VARCHAR2,
    request_set                   IN VARCHAR2 DEFAULT NULL,
    stage                         IN VARCHAR2,
    set_application               IN VARCHAR2,
    program_sequence              IN NUMBER,
    parameter                     IN VARCHAR2);
```

Description

This procedure removes a concurrent program request set parameter from a request set definition.

Arguments (input)	**program**	The short name used as the developer name of the concurrent program.
	program_ application	The short name of the application that owns the concurrent program.
	request_set	The short name of the request set.
	set_application	The short name of the application that owns the request set.
	program_ sequence	The sequence number of this program in the set. All programs in a stage require a unique sequence number.
	parameter	The name of the program parameter to delete.

FND_SET.ADD_STAGE

Summary

```
procedure FND_SET.ADD_STAGE
    (name                        IN VARCHAR2,
    request_set                  IN VARCHAR2,
    set_application              IN VARCHAR2,
    short_name                   IN VARCHAR2,
    description                  IN VARCHAR2 DEFAULT NULL,
    display_sequence             IN NUMBER,
    function_short_name          IN VARCHAR2 DEFAULT 'FNDRSSTE'
    function_application         IN VARCHAR2 DEFAULT 'FND',
    critical                     IN VARCHAR2 DEFAULT 'N',
    incompatibilities_allowed    IN VARCHAR2 DEFAULT 'N',
    start_stage                  IN VARCHAR2 DEFAULT 'N',
    language_code                IN VARCHAR2 DEFAULT 'US');
```

Description Adds a stage to a request set. This procedure corresponds to the ”Stages” window of the ”Request Sets” window.

Arguments (input)	**name**	The name of the stage.
	request_set	The short name of the request set.
	set_application	The application that owns the request set.
	short_name	The stage short (non–translated) name.
	description	Description of the stage.
	function_ short_name	Accept the default, ”FNDRSSTE”, the Standard Stage Evaluation function.

function_ application	Accept the default, ”FND”.
critical	Specify ”Y” if the return value of this stage affects the completion status of the request set, and ”N” if it does not.
start_stage	Specify ”Y” or ”N” to indicate whether this stage is the start stage for the set.
incompatibilities _allowed	Specify ”Y” or ”N” to indicate whether to allow incompatibilities for this stage.
language_code	The language code for the above data.

FND_SET.REMOVE_STAGE

Summary

```
procedure FND_SET.REMOVE_STAGE
    (request_set                IN VARCHAR2,
    set_application             IN VARCHAR2,
    stage                       IN VARCHAR2);
```

Description Use this procedure to delete a stage from a request set.

Arguments (input)

request_set	The short name of the request set.
set_application	The short name of the application that owns the request set.
stage	The short name of the stage to be removed.

FND_SET.LINK_STAGES

Summary

```
procedure FND_SET.LINK_STAGES
    (request_set                IN VARCHAR2,
    set_application             IN VARCHAR2,
    from_stage                  IN VARCHAR2,
    to_stage                    IN VARCHAR2 DEFAULT NULL,
    success                     IN VARCHAR2 DEFAULT ’N’,
    warning                     IN VARCHAR2 DEFAULT ’N’,
    error                       IN VARCHAR2 DEFAULT ’N’);
```

Description Use this procedure to link two stages.

Attention: This procedure updates the specified links. Sets created by FND_SET.CREATE_SET have null links between stages by default.

Arguments (input)

request_set	The short name of the request set.
set_application	The application that owns the request set.
from_stage	The short name of the ”from” stage.
to_stage	The short name of the ”to” stage.
success	Create success link. Specify ’Y’ or ’N’.
warning	Create warning link. Specify ’Y’ or ’N’.
error	Create error link. Specify ’Y’ or ’N’.

FND_SET.INCOMPATIBILITY

Summary

```
procedure FND_SET.INCOMPATIBILITY
  (request_set                IN VARCHAR2,
   application                IN VARCHAR2,
   stage                      IN VARCHAR2 DEFAULT NULL,
   inc_prog                   IN VARCHAR2 DEFAULT NULL
   inc_prog_application       IN VARCHAR2 DEFAULT NULL,
   inc_request_set            IN VARCHAR2 DEFAULT NULL,
   inc_set_application        IN VARCHAR2 DEFAULT NULL,
   inc_stage                  IN VARCHAR2 DEFAULT NULL);
```

Description Use this procedure to register an incompatibility for a set or stage. Examples are given below.

Arguments (input)

request_set	The short name of the request set.
application	The short name of the application that owns the request set.
stage	The short name of the stage (for stage incompatibility).
inc_prog	Short name of the incompatible program.
inc_prog_ application	Application that owns the incompatible program.
inc_request_set	Short name of the incompatible request set.

inc_set_ application The short name of the application that owns the incompatible request set.

inc_stage Short name of the incompatible stage.

Examples

1. Set X is incompatible with program Y:

```
fnd_set.incompatibility(request_set=>'X'
                        application=>'APPX'
                        inc_prog_short_name=>'Y',
                        inc_prog_application=>'APPY');
```

2. Set X is incompatible with set Y:

```
fnd_set.incompatibility(request_set=>'X',
                        application=>'APPX',
                        inc_request_set=>'Y',
                        inc_set_application=>'APPY');
```

3. Set X is incompatible with stage 2 of set Y:

```
fnd_set.incompatibility(request_set=>'X',
                        application=>'APPX',
                        inc_request_set=>'Y',
                        inc_set_application=>'APPY',
                        inc_stage_number=>2);
```

4. Stage 3 of set X is incompatible with program Y:

```
fnd_set.incompatibility(request_set=>'X',
                        application=>'APPX',
                        stage_number=>3,
                        inc_prog_short_name=>'Y',
                        inc_prog_application=>'APPY');
```

FND_SET.DELETE_INCOMPATIBILITY

Summary

```
procedure FND_SET.DELETE_INCOMPATIBILITY
  (request_set                   IN VARCHAR2,
   application                   IN VARCHAR2,
   stage                         IN VARCHAR2 DEFAULT NULL,
   inc_prog                      IN VARCHAR2 DEFAULT NULL
```

```
inc_prog_application       IN VARCHAR2 DEFAULT NULL,
inc_request_set            IN VARCHAR2 DEFAULT NULL,
inc_set_application        IN VARCHAR2 DEFAULT NULL,
inc_stage                  IN VARCHAR2 DEFAULT NULL);
```

Description Use this procedure to delete a request set incompatibility rule.

Arguments (input)

request_set The short name of the request set.

application The short name of the application that owns the request set.

stage The short name of the stage (for stage incompatibility).

inc_prog Short name of the incompatible program.

inc_prog_ application Application that owns the incompatible program.

inc_request_set Short name of the incompatible request set.

inc_set_ application The short name of the application that owns the incompatible request set.

inc_stage Short name of the incompatible stage.

FND_SET.ADD_SET_TO_GROUP

Summary

```
procedure FND_SET.ADD_SET_TO_GROUP
   (request_set         IN           VARCHAR2,
    set_application     IN           VARCHAR2,
    request_group       IN           VARCHAR2,
    group_application   IN           VARCHAR2);
```

Description Use this procedure to add a request set to a request group. This procedure corresponds to the "Requests" region in the "Request Groups" window in System Administration.

Arguments (input)

request_set The short name of the request set to add to the request group.

set_application The application that owns the request set.

request_group The request group to which to add the request set.

group_ application The application that owns the request group.

FND_SET.REMOVE_SET_FROM_GROUP

Summary

```
procedure FND_SET.REMOVE_SET_FROM_GROUP
   (request_set         IN            VARCHAR2,
    set_application     IN            VARCHAR2,
    request_group       IN            VARCHAR2,
    group_application   IN            VARCHAR2);
```

Description

Use this procedure to remove a request set from a request group.

Arguments (input)

request_set The short name of the request set to remove from the request group.

set_application The application that owns the request set.

request_group The request group from which to remove the request set.

group_ application The application that owns the request group.

CHAPTER

21 Standard Request Submission

This chapter describes Standard Request Submission in Oracle Application Object Library.

The Overview of Standard Request Submission includes a summary of Standard Request Submission features, definitions of key concepts, and an outline of the steps necessary to add Standard Request Submission to your application.

The following topics are covered:

- Overview of Standard Request Submission
- Implementing Standard Request Submission

Overview of Standard Request Submission

Standard Request Submission provides you with a standard interface for running and monitoring your application's reports. You no longer need to design and maintain special forms to submit reports. Standard Request Submission lets you avoid programming custom validation logic in Oracle Forms when you add a new report to your application.

Standard Request Submission provides you with a single form you use to submit any of your application reports and concurrent programs, as well as another form you use to check on your reports' progress and to review your reports online. Standard Request Submission also lets your users create sets of reports to submit all at once. Standard Request Submission includes an easy–to–use interface you use to add new reports and to specify the parameters to pass to your reports.

Standard Request Submission includes all the features your users need to submit and monitor their reports without using concurrent processing terminology. Although Standard Request Submission is designed with end user reporting in mind, you can use it to submit concurrent programs that do not create output.

To learn about running requests, viewing reports, creating request sets, and other end–user features of Standard Request Submission, see the *Oracle Applications User's Guide*. To learn about administration of request sets, customization of the Submit Requests window, and other system administrator features of Standard Request Submission, see the *Oracle Applications Administrator's Guide*.

Additional Information: *Oracle Applications System Administrator's Guide, Oracle Applications User's Guide*

Basic Application Development Needs

Oracle Application Object Library provides you with the features you need to satisfy the following basic application development needs:

- Provide your users with a standard interface for running and monitoring your application reports and other programs
- Let your users create and run sets of reports
- Let your users view any of their reports on line
- Let your users automatically run reports and request sets at specific time intervals

- Let your users specify whether reports in a set should run sequentially or in parallel
- Let your users specify whether to continue with the next report if one report in a sequential set fails
- Provide your users with a single report that summarizes the completion information about all the reports in a set
- Restrict reports users can run
- Define report parameters that have different types of validation without programming your own validation logic
- Invisibly pass parameters whose values come from your user's environment to your reports

Major Features

Submit Request Form

The Submit Request form is the standard form you and your users use to run reports and other programs. You need not build forms that submit requests to run your reports or program trigger logic to validate report parameters.

With just one simple form to learn and use, your users save time in submitting reports and request sets. Your users can quickly submit as many reports and request sets as they want. Pop–up windows let your users easily choose the information they want to see in their reports.

Automatic Resubmission

Standard Request Submission can automatically resubmit your report or request set periodically. When you submit your report or request set, you can specify the starting date and time, the interval between resubmissions, and whether to measure the interval between the requested submission time or the completion of your request.

Alternately, you may specify a time of day for the daily resubmission of your request. You can also specify an end date and time when your request should cease resubmitting.

Request Sets

You can define sets of reports, then submit an entire set at the same time with just one transaction. Your request sets can include any reports or programs you submit from the Submit Request form. Using request sets, you can submit the same reports regularly without having to specify each report or program every time you run the set.

Users own the request sets they define, and can access their private request sets from any responsibility. Only Oracle System Administrators and owners can update a request set. Users may run request sets they do not own if their report security group includes the request set.

Request Set Options

You can define whether the reports in a request set should run in a particular order. If you specify that the reports in a set should run sequentially, you can control whether a request set continues to run reports in the set or stops immediately if a report in the set ends in an error.

For each report in a set, you can specify a printer for the output, the number of copies, and whether to save the output to an operating system file. Standard Request Submission saves these options so you do not have to specify them every time you run a request set.

Request Set Log File

Oracle Application Object Library produces a single log file that contains the completion status of all reports in a request set. If a report in a request set fails, you can quickly identify it and review the appropriate detailed log file to determine the reason for failure.

Viewing Requests

You and your users can monitor your reports' progress using the View Requests form. After your reports complete, you can view them online through a scrolling pop–up window without the delay of printing out the entire report.

Cross–application Reporting

Your users can use Standard Request Submission to run reports that belong to applications other than the one they are currently using. For Releases 10.7 and 11, all Oracle Applications products typically use the

APPS schema, so cross–application reporting can be greatly simplified. However, to enable cross–application reporting where you have custom schemas and custom applications, or you are using multiple APPS schemas, your ORACLE DBA must ensure that the Submit Request form can access the tables in the report's application needed to validate report parameters. The concurrent manager automatically uses the ORACLE schema associated with the report's application to run the report.

Oracle Applications system administrators define data groups for each responsibility. Data groups contain lists of application names and ORACLE schemas. The responsibility's data group determines which ORACLE schema to use for a given application name.

Definitions

Child Request (Sub–request)

A child request or a sub–request is a request submitted by another concurrent request (a parent request). In the case of Standard Request Submission, when you submit a request set, the request set submits the reports and programs that you have included in the request set. The reports included in the request set become child requests when the request set submits them for concurrent processing.

Parameter

A value you specify when you run a report. For example, if you run an audit report, you might specify the audit date as a parameter when you run the report.

Parent Request

A parent request is a concurrent request that submits other concurrent requests. In the case of Standard Request Submission, a request set is a parent. When you submit a request set, the request set submits the reports or programs that you have included in the request set. A parent request may be sequential or parallel which determines whether the requests it submits run one at a time or all at once.

Program Application

The application with which you register your report in the Concurrent Programs window.

Responsibility Application

The application with which you define your responsibility in the Responsibility form.

Value

What you enter as a parameter. A value can be a date, a name, text, or a number. The Submit Request form provides you with lists of values for most parameters, to ensure you choose valid values.

Value Set

A set of values against which Oracle Application Object Library validates values your end user enters when running your program. You define your value set by specifying validation rules, format constraints and other properties. For example, you could define a value set to contain values that are character strings, validated from a table in your application. You can specify that Oracle Application Object Library use the same value set to validate different report parameters. You can also use value sets that you use in your flexfields to validate your report parameters.

Controlling Access to Your Reports and Programs

Defining Report Submission Security

Your system administrator controls which responsibilities have access to the reports and other programs in your application. You or your system administrator should first create related groups of reports and request sets. When you define a new responsibility, you assign a report security group to that responsibility.

Define Responsibility
Define Report Group
Oracle Applications System Administrator's Guide

Defining Menus

When you or your system administrator define new menus, you should put the Submit Request, View Requests, and Define Request Set functions on the menu of every responsibility that should have access to Standard Request Submission reports. Be sure to define a request group for any responsibility that has access to the Submit Request form.

Menus Window (See page 11 – 26)

Implementing Standard Request Submission

To take advantage of Standard Request Submission, you must:

- Build your report as a concurrent program, from writing the execution logic and placing the execution file in the correct location on your file system to defining a Concurrent Program Executable for your program
- Design the parameter pop–up window your users see on the Submit Requests form
- Define necessary value sets and validation tables
- Define your concurrent program to use Standard Submission and define your report parameters to make use of Standard Request Submission

The following sections provide you with implementation suggestions for the preceding actions.

Developing Reports for Standard Request Submission

You write a concurrent program and define it as a Standard Submission report. You plan your parameter window and identify the value sets you need to validate your parameters. Define any new value sets that Standard Request Submission will use to validate your report parameters. Note that although Standard Request Submission is designed with end user reporting in mind, you can allow your users to use the Submit Requests form to submit any custom concurrent programs.

Writing Your Report or Program

If your report requires parameters, it should expect to receive them in the same order as your users enter them in the pop–up window. For any type of report except a Oracle Reports report, you as the developer have to maintain the same parameter order in both the report and the pop–up window. When your report is an Oracle Reports report, the order is irrelevant because your parameters are passed to the report with parameter names (tokens) attached.

After you finish writing the report, place it in the appropriate place for your platform. For example, in Unix, use the ***sql*** or ***srw*** directories under the appropriate application top directory.

Use the Concurrent Program Executable window to define your report file as an executable file. You'll use this executable to define your concurrent program.

Overview of Concurrent Processing (See page 15 – 2)
Implementing Concurrent Processing (See page 15 – 21)

Concurrent Processing
Oracle Applications Installation Manual

Concurrent Program Executable Window (See page 16 – 6)
Concurrent Programs Window(See page 16 – 10)

Designing the Parameter Pop–up Window

Determine what parameters your report requires. Then determine what order in which your user should enter parameters in the pop–up window on the Submit Requests form. To define the pop–up window, you also need to define one value set for each parameter. Design value sets to limit the user's choices to valid values. You have the option of restricting the list of values for a table–validated parameter based on the values your user entered for earlier parameters. You set up these restrictions by using defining cascading dependencies when defining your value sets.

You may want your report to expect parameter values such as internal ID numbers that are meaningless to your users while the pop–up window takes user–friendly values. You can select the column to use for the ID as well as the user–friendly meaning, description or other columns you want to use. You can define value sets to have independent, dependent, table, special, pair or no validation.

Additional Information: Planning and Defining Values and Value Sets
Oracle Applications Flexfields Guide

Defining Parameter Validation

Validating parameters in a report pop–up window is very similar to validating segments in a flexfield. You create values sets for your values, decide whether to provide a list of values for your users, and specify any security rules for your values.

Defining Value Sets

Typically, when you write a report or other concurrent program, you want to pass parameters that have specific data types and values. Before you can define and use your report with Oracle Application Object Library, you need to specify the value sets of your parameters. Use the Value Sets window to define a value set for each of your report parameters. When you define a value set, you specify format options as well as other attributes such as whether a value is mandatory and the maximum size of a value. Your value set can have Validation Type of Table, Independent, Dependent, Special, Pair or None.

You can define a value set to validate from a table in your application, such as a lookup table. Make sure the maximum size of your value set is large enough to accommodate your validation data. Once you define a value set, Oracle Application Object Library can use it to validate parameters you pass to your report.

If you already have value sets defined for your key or descriptive flexfields, you can use these to validate your concurrent program parameters. Note that if you share value sets with flexfields, flexfield value security can affect the report parameter values your users can choose. You should specify for each parameter whether you want to enable security.

With Special and Pair value sets you can pass flexfield combinations as parameters to your report. Or you can call other user exits from your Special value sets.

Value Sets Window
Oracle Applications Flexfields Guide

Standard Request Submission
Oracle Applications Flexfields Guide

Defining Values for Value Sets

After you register your report parameters, each report parameter references a value set. If you are using independent or dependent value sets, you can enter values into each corresponding value set using the Segment Values form.

You can easily identify your value sets by using the Segment Values form. You select the program and parameter for which you want to define values using the Find window.

Segment Values Window
Oracle Applications Flexfields Guide

Defining Your Report or Other Program

You must define your report as a concurrent program with Oracle Application Object Library before your users can run it from the Submit Requests form or an application form. Use the Concurrent Programs form to register your report. Define your report just like any other concurrent program, including defining a concurrent program executable. To indicate that your users can use the Submit Requests form to run the program, simply check the Use in SRS check box of the Concurrent Programs form.

Registering Your Parameters

If your report requires parameters, press the Parameters button to get to the Parameters block to define your report parameters.

While you are registering your report parameters, you are also defining the structure of a pop–up window that pops up when your users submit the report in the Submit Requests form. Enter your report parameters in the sequence you want them to appear in the pop–up window and in the order in which the report expects them. Standard Request Submission passes arguments to your report in the sequence you specify. Please keep in mind that what your users enter in the pop–up window and what Standard Request Submission passes to your report can be different if you have specified different Value and Meaning columns for your table–validated parameters.

Make sure you enable all parameters. You specify the value set that identifies valid values, whether the parameter requires a value, whether security is enabled and a default value, if any. Specify if the parameter should display to the user. If you want to link two values in a High_Low relationship, choose High or Low in the Range field. Low values must come before high ones.

The Request Set window accessible from the Oracle System Administration menu also allows you to selectively display the parameters of a report.

Parameter Defaults

You decide whether your users enter a value for a parameter or whether the parameter is passed behind the scenes to your report by checking or unchecking the Display check box. If this is a parameter your users enter, then you must define a prompt for the parameter. You can specify a default type and value for the parameter.

If your parameter is displayed, your users can override the default value you specify. If your parameter is non–displayed, your report receives the value you specify as the default value. Use non–displayed parameters to pass hidden values from your users' environment, such as SET_OF_BOOKS_ID, to your report.

Concurrent Programs Window (See page 16 – 10)

Cross–application Reporting

You can use the cross–application reporting capabilities of Standard Request Submission to run a report belonging to an application other than the application associated with the user's current responsibility.

Method to Determine Which ORACLE ID to Use

When you submit a report using Standard Request Submission, your concurrent manager uses a different method from previous releases to decide which ORACLE schema to use to process your request. The concurrent manager accesses the information recorded by the AutoInstall process to detect what products you have at your site and the products' inter–dependencies.

AutoInstall and the Oracle Applications system administrator set up data groups containing list of Application Name / ORACLE schema pairs. Each responsibility has an assigned data group. When you run a concurrent program from the Submit Requests form, the application name of your program is matched with its associated ORACLE schema in that responsibility's data group.

Defining Data Groups
Oracle Applications System Administrator's Guide

Accessing Another Application's Validation Tables

If you are using the cross–application reporting capabilities of Standard Request Submission to run a report, the Submit Requests form uses the ORACLE schema of the report's application to validate the report parameters. The application name is matched with an ORACLE schema through the responsibility's data group. Your database administrator should make sure that the ORACLE schema that the Submit Requests form uses to validate your report parameters has all the necessary grants, synonyms, and database privileges to access the validation tables that your report uses.

For Releases 10.7 and 11, all Oracle Applications products typically use the APPS schema, so cross–application reporting can be greatly simplified. However, to enable cross–application reporting where you have custom schemas and custom applications, or where you are using multiple APPS schemas, your ORACLE DBA must ensure that the Submit Request form can access the tables in the report's application needed to validate report parameters. The concurrent manager automatically uses the ORACLE schema associated with the report's application to run the report.

For example, suppose you want to run an Oracle Payables report using the Submit Requests form in an Oracle Purchasing responsibility. The parameters of the Oracle Payables report are validated against tables in the Oracle Payables ORACLE schema. The data group assigned to the Oracle Purchasing responsibility contains a listing of the ORACLE schema associated with Oracle Payables (which might be APPS). The report runs in that ORACLE schema.

If you submit a custom application report using a responsibility associated with a different application, you and your system administrator need to provide the concurrent manager with the correct ORACLE schema to use. You should include your custom applications in the data groups of any responsibility using your custom reports.

Oracle Applications Data Model Architecture
Oracle Applications Installation Manual

CHAPTER

22 Request Sets

This chapter provides an overview of Request Sets in Oracle Application Object Library. It includes an overview of request sets and an outline of the steps necessary to implement requests sets in your application.

- Overview of Request Sets

Overview of Request Sets

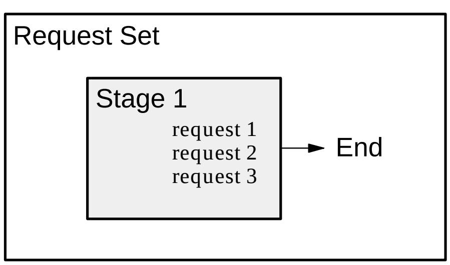
Request sets allow you to submit several requests together using multiple execution paths. A request set is a collection of reports and / or programs that are grouped together. You can thus submit the reports and / or programs in a request set all at once using a single transaction.

Request sets have been completely redesigned in Release 11 to support multiple execution paths. Request sets can now be defined to submit requests depending on the completion statuses of previously submitted requests in the set. For example, if a certain request were to fail, a set could submit a cleanup request, rather than continuing along its normal execution path. A set could submit one request or another based on the number of lines posted by earlier requests. A single set can now execute certain groups of requests in parallel while executing other requests serially.

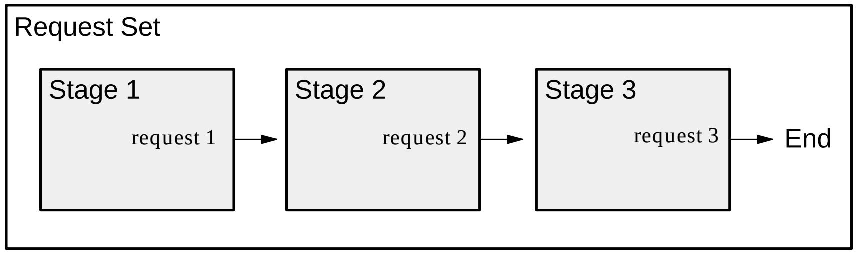
Additional Information: *Oracle Applications User's Guide, Oracle Applications System Administrator's Guide*

Sets, Stages, and Requests

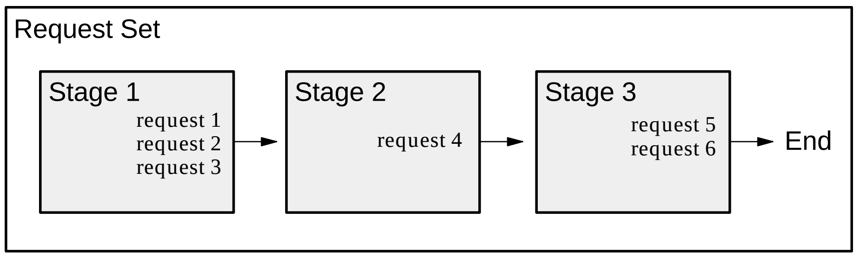
Release 11 request sets are divided into stages. A stage is a component of a request set used to group requests within the set. Each stage contains one or more concurrent requests. All of the requests in a given stage are executed in parallel. The request set executes one stage at a time, following links from stage to stage. The links that are followed depend on the completion statuses of the individual stages. A stage completes only when all of the requests contained in the stage have completed. A set moves on to the next stage only when the previous stage has completed. For example, the analog of a Release 10 "Parallel" request set would be a set with a single stage that contains all of the requests.



The analog of a Release 10 "Sequential" request set would be a set with many stages and one request in each stage.



The concurrent manager allows only one stage in a request set to run at a time. When one stage is complete the following stage is submitted. A stage is not considered to be complete until all of the requests in the stage are complete. One advantage of using stages is the ability to run several requests in parallel and then move sequentially to the next stage. This allows for a more versatile and efficient request set.



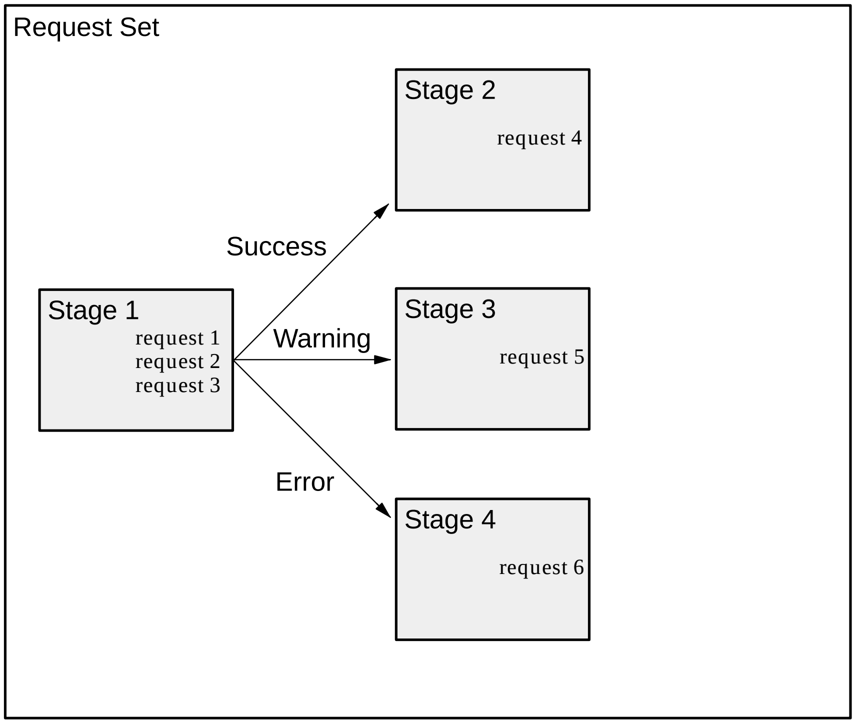
Stage Completion Statuses

Like request sets and concurrent requests, stages can complete with different statuses. Each stage might complete with Success, Warning, or Error. The completion status of a stage can affect branching within a set, and determine the completion status of the entire set. The completion status of a stage is calculated by a PL/SQL function. When you define a request set stage, you can choose the PL/SQL function that will calculate the completion status of that stage. Oracle Application Object Library provides a standard PL/SQL function to compute the completion status of a stage from the completion statuses of the requests within that stage. Other functions may be available as well.

Request Set Execution and Stage Links

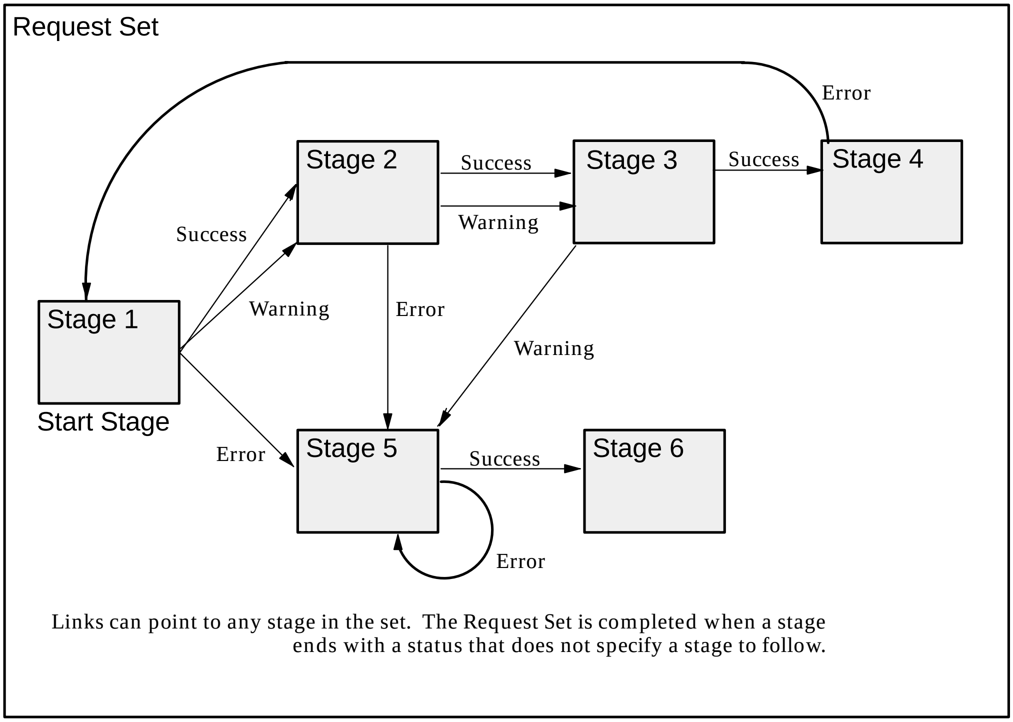
Each request set has one stage designated as the "start stage." The start stage is always the first stage of a request set to be executed. When you define a set, you specify the possible execution paths from the start stage by linking stages together. Each stage can have one link for each of its three possible completion statuses. When a stage completes with a given status, the link associated with that status specifies the next stage to be executed by the set. If no link is specified for that status, then the set has completed.

For example, the following request set always begins execution with Stage 1. If Stage 1 were to complete with the status Warning, then the Warning Link would be followed, and Stage 3 would be executed. After stage 3 completes, the set will end, since there are no links that may be followed.



There are no restrictions on linking stages within a set. Any stage may be linked into any other stage, including itself. Loops are allowed. Two or more links can point to the same stage.

Figure 22 – 1



Stage Functions

As discussed above under Stage Completion Statuses, the completion status of a stage is computed by a PL/SQL function. The function can use information about the requests in a stage when calculating the status of the stage. For example, the Standard Stage Evaluation function uses the completion statuses of the requests in a stage to calculate the completion status of that stage.. For each stage in your set, you can choose a function from the list of registered functions. You can also specify which concurrent requests in the stage will provide information to be used in the function's calculation. Most stages will use the Standard Stage Evaluation function, but other functions are available.

The Standard Stage Evaluation Function

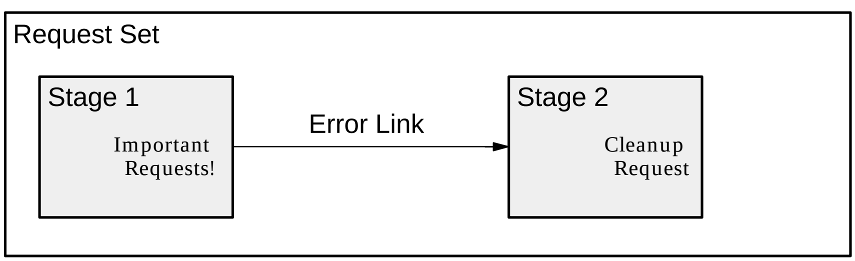
Any stage may use the Standard Stage Evaluation function provided by Oracle Application Object Library. This function computes the stage completion status from the completion statuses of the specified requests in the set. The Standard Stage Evaluation function will return Success if all of the requests completed with Success. The function will return Error if one or more requests completed with Error. Finally, the function will return Warning if one or more requests completed with Warning, and no requests completed with Error.

Request Set Completion Status

When a stage completes with a status for which there is no link defined, then the request set exits. When a set exits, its completion status is determined by one of two methods:

- By default, the completion status of the set is the same as the completion status of the last stage executed. For example, if the last stage were to complete with Warning, then the request set would exit with warning status.
- The user can override the default behavior by declaring certain stages to be "critical." If the set executes a critical stage at any time, then when the set exits, the completion status of the set will be equal to the completion status of the most recently executed critical stage. This can be useful if the final stage of the set is a "clean up" stage, and is not considered important to the overall status of the set.

For example, in the following diagram Stage 1 is very important to us, whereas Stage 2 is not. Here we would want to declare Stage 1 to be a "critical" stage, so that if Stage 1 fails, the completion status of the set will be determined by Stage 1, rather than Stage 2.



CHAPTER

23 The TEMPLATE Form

This chapter provides you with information on the contents of the TEMPLATE form you use to start your application forms.

- Overview of the TEMPLATE Form
- Libraries in the TEMPLATE Form
- Special Triggers in the TEMPLATE Form

Overview of the TEMPLATE Form

The TEMPLATE form is the required starting point for all development of new forms. Start developing each new form by copying the TEMPLATE.fmb file, located in $AU_TOP/forms/US (or your language and platform equivalent), to a local directory and renaming it as appropriate.

TEMPLATE contains the following:

- Platform–independent references to object groups in the APPSTAND form (STANDARD_PC_AND_VA, STANDARD_TOOLBAR, and STANDARD_CALENDAR)
- Platform–independent attachments of several libraries (including FNDSQF, APPCORE, and APPDAYPK)
- Several form–level triggers with required code.

  Special Triggers in the TEMPLATE form (See page 23–6)

- Program units that include a spec and a body for the package APP_CUSTOM, which contains default behavior for window opening and closing events. You usually have to modify this code for the specific form under development.

  Controlling Window Behavior (See page 7 – 2)

- The Applications color palette, containing the two colors required by the referenced visual attributes (”canvas” and ”button”), ”pure” colors (such as ”black,” ”white,” ”blue,” and ”red”), and various other colors named after their content of Red, Blue and Green (such as ”r40g70b100”).
- Many referenced objects (from the object groups) that support the Calendar, the toolbar, alternative regions, and the menu. These objects include LOVs, blocks, parameters, and property classes, and so on.
- The TEMPLATE form contains sample objects that show typical items and layout cosmetics. These are provided merely as samples; to remove them entirely from your form, delete the following objects.
  – blocks: BLOCKNAME, DETAILBLOCK
  – window: BLOCKNAME
  – canvas–view: BLOCKNAME

Libraries in the TEMPLATE Form

The TEMPLATE form includes platform–independent attachments of several libraries. Some of these libraries are attached ”directly” to the TEMPLATE (FNDSQF, APPCORE, and APPDAYPK), while the others are attached to these three libraries. However, in Oracle Forms, the different types of attachments are indistinguishable. If more libraries are later attached to any of these libraries, the additional libraries will also appear to be attached directly to TEMPLATE.

The following libraries are all attached to TEMPLATE in Release 11. You may also see additional libraries, particularly if your site uses Oracle Applications in multiple countries or if your site uses Oracle Industry Applications.

Warning: You should not modify any Oracle Applications libraries other than the CUSTOM library, or you could seriously damage your Oracle Applications products.

APPCORE

APPCORE contains the packages and procedures that are required of all forms to support the menu, Toolbar, and other required standard behaviors. Additionally it contains packages that should be called to achieve specific runtime behaviors in accordance with the *Oracle Applications User Interface Standards*, such as the way in which fields are enabled, behaviors of specific types of windows, and the dynamic ’Special’ menu. Finally, it contains various other utilities for exception handling, message levels, and so on. Some APPCORE event routines call routines in the VERT, GLOBE, and CUSTOM libraries (in that order).

Procedures and functions in APPCORE typically have names beginning with ”APP”.

Oracle Applications APIs (See page 26 – 1)

APPDAYPK

APPDAYPK contains the packages that control the Oracle Applications Calendar feature.

The Calendar (See page 9 – 23)

FNDSQF

FNDSQF contains packages and procedures for Message Dictionary, flexfields, profiles, and concurrent processing. It also has various other utilities for navigation, multicurrency, WHO, etc.

Procedures and functions in FNDSQF typically have names beginning with ”FND”.

Message Dictionary APIs for PL/SQL Procedures (See page 12 – 14)
Flexfield Definition Procedures (See page 14 – 30)
FND_PROFILE: User Profile APIs (See page 13 – 8)
Concurrent Processing APIs for PL/SQL Procedures (See page 20 – 2)
FNDSQF Routine APIs (See page 27 – 1)

CUSTOM

The CUSTOM library allows extension of Oracle Applications forms without modification of Oracle Applications code. You can use the CUSTOM library for customizations such as Zoom (such as moving to another form and querying up specific records), enforcing business rules (for example, vendor name must be in uppercase letters), and disabling fields that do not apply for your site.

You write code in the CUSTOM library, within the procedure shells that are provided. All logic must branch based on the form and block for which you want it to run. Oracle Applications sends events to the CUSTOM library. Your custom code can take effect based on these events.

Using the CUSTOM Library (See page 0 – 1)

GLOBE

The GLOBE library allows Oracle Applications developers to incorporate global or regional features into Oracle Applications forms without modification of the base Oracle Applications form. Oracle Applications sends events to the GLOBE library. Regional code can take effect based on these events. The GLOBE library calls routines in the JA, JE, and JL libraries.

VERT

The VERT library allows Oracle Applications developers to incorporate vertical industry features (for automotive, consumer packaged goods, energy, and other industries) into Oracle Applications forms without modification of the base Oracle Applications form. Oracle Applications sends events to the VERT library. Vertical industry code can take effect based on these events. The VERT library calls routines in various other libraries.

JA

The JA library contains code specific to the Asia / Pacific region and is called by the GLOBE library.

JE

The JE library contains code specific to the EMEA (Europe / Middle East / Africa) region and is called by the GLOBE library.

JL

The JL library contains code specific to the Latin America region and is called by the GLOBE library.

Special Triggers in the TEMPLATE form

The TEMPLATE form contains several form–level triggers that must exist in order for other routines to operate properly. Specific rules about modifications you can safely make to these triggers are discussed below.

☞ **Attention:** Under no circumstances may any of these triggers be deleted.

The text within these triggers must remain within the trigger; however, frequently developers need to add text before or after this text. These triggers include:

Standard Forms Triggers:

– KEY–CLRFRM

– KEY–COMMIT

– KEY–DUPREC

– KEY–EDIT

– KEY–EXIT

– KEY–HELP

– KEY–LISTVAL

– KEY–MENU

– ON–ERROR

– POST–FORM

– PRE–FORM

– WHEN–FORM–NAVIGATE (reference)

– WHEN–NEW–BLOCK–INSTANCE

– WHEN–NEW–FORM–INSTANCE

– WHEN–NEW–ITEM–INSTANCE

– WHEN–NEW–RECORD–INSTANCE

– WHEN–WINDOW–CLOSED

– WHEN–WINDOW–RESIZED

User–Named Triggers:

– ACCEPT

– CLOSE_THIS_WINDOW (reference)

– CLOSE_WINDOW

– EXPORT (referenced)
– FOLDER_ACTION
– FOLDER_RETURN_ACTION
– LASTRECORD (reference)
– MENU_TO_APPCORE (reference)
– QUERY_FIND
– STANDARD_ATTACHMENTS (reference)
– ZOOM (reference)

You cannot change triggers that are referenced into the form.

Triggers That Often Require Some Modification

ACCEPT

```
APP_STANDARD.EVENT('ACCEPT');
```

This trigger processes invocation of the "Action, Save and Proceed" menu choice or toolbar button. It saves and moves to the next record of the block specified as the First Navigation Block.

Replace the code in this trigger, or create block–level triggers with execution style 'Override'.

Save and Proceed (See page 10 – 7)

FOLDER_RETURN_ACTION

```
null;
```

This trigger allows customization of specific folder events.

Replace text with specific code needed to process folder actions.

Warning: Oracle Corporation does not support modifications to this trigger except for Oracle Applications internal use.

KEY–DUPREC

```
APP_STANDARD.EVENT('KEY–DUPREC');
```

This trigger disables the default duplicate record functionality of Oracle Forms.

To process the ”Edit, Duplicate Record Above” menu choice properly, code a block–level KEY–DUPREC with execution style ’Override’. This trigger should perform a duplicate_record, then validate or clear fields as needed.

Duplicating Records (See page 7 – 16)

KEY–CLRFRM

```
APP_STANDARD.EVENT('KEY-CLRFRM');
```

This trigger validates the record before attempting to clear the form.

Add any additional code *after* the supplied text. Typically you would add GO_BLOCK calls if you have alternative regions in your form, where the GO_BLOCK calls repopulate your region control poplist after a Clear Form operation.

KEY–MENU

```
APP_STANDARD.EVENT('KEY-MENU');
```

This trigger disables the Block Menu command of Oracle Forms.

To enable operation of Alternative Regions via the keyboard from a specific block, code a block–level KEY–MENU with execution style ’Override’. This trigger should open an LOV with the same choices as the Alternative Region control poplist.

Alternative Regions (See page 7 – 12)

KEY–LISTVAL

```
APP_STANDARD.EVENT('KEY-LISTVAL');
```

This trigger performs flexfield operations or LOV invocation.

Create block– or item–level triggers with execution style ’Override’ on fields using the Calendar, or fields that dynamically invoke flexfields.

The Calendar (See page 9 – 23)

ON–ERROR

```
APP_STANDARD.EVENT('ON-ERROR');
```

This trigger processes all errors, server or client side, using Message Dictionary calls.

To trap specific errors, check for your specific errors before the APP_STANDARD call.

```
declare
   original_mess      varchar2(80);
begin
   IF MESSAGE_CODE = <your message number> THEN
      original_mess := MESSAGE_TYPE||'–'||
         to_char(MESSAGE_CODE)||': '||MESSAGE_TEXT;
    ––– your code handling the error goes here
     message(original_mess);
   ELSE
      APP_STANDARD.EVENT('ON_ERROR');
   END IF
end;
```

Overview of Message Dictionary (See page 12 – 2)
APP_EXCEPTION: Exception Processing APIs (See page 26 – 6)

POST–FORM

```
APP_STANDARD.EVENT('POST–FORM');
```

This trigger is reserved for future use.

Add any additional code *before* the supplied text.

PRE–FORM

```
FND_STANDARD.FORM_INFO('$Revision: <Number>$',
                   '<Form Name>',
                   '<Application Shortname>',
                   '$Date: <YY/MM/DD HH24:MI:SS> $',
                   '$Author: <developer name> $');
APP_STANDARD.EVENT('PRE–FORM');
APP_WINDOW.SET_WINDOW_POSITION('BLOCKNAME',
                   'FIRST_WINDOW');
```

This trigger initializes internal Oracle Applications values and the menu. The values you enter here are shown when you choose "Help, About Oracle Applications" from the Oracle Applications menu.

You must modify the application short name. The application short name controls which application's online help file is accessed when the user presses the window help button on the toolbar. If you leave the application short name as FND, your user will not see any help because Oracle Applications will not be able to construct a valid help target.

The form name is the user form name (form title). This is for your reference only, and is not used elsewhere.

Oracle Corporation uses a source control system that automatically updates the values beginning with ”$”. If you are not using that source control system, you can and should modify those values with your own development information.

You must also modify the APP_WINDOW call to use your own block name (of your first block) instead of BLOCKNAME. Do not modify the string FIRST_WINDOW.

Controlling Window Behavior (See page 7 – 2)

QUERY_FIND

```
APP_STANDARD.EVENT('QUERY_FIND');
```

This trigger issues a default message stating that Query Find is not available.

Replace the code in this trigger, or create block–level triggers with execution style 'Override' when you create a Find window or Row–LOV in your form.

Query Find Windows (See page 8 – 2)

WHEN–NEW–FORM–INSTANCE

```
FDRCSID('$Header: ... $');
APP_STANDARD.EVENT('WHEN-NEW-FORM-INSTANCE');

-- app_folder.define_folder_block('template test',
    'folder_block', 'prompt_block', 'stacked_canvas',
    'window', 'disabled functions');
-- app_folder.event('VERIFY');
```

The APP_STANDARD.EVENT call in this trigger supports the query–only mode invoked by FND_FUNCTION.EXECUTE. The FDRCSID call supports the Oracle Applications source control system. The APP_FOLDER calls are for Oracle Applications internal use only. Custom forms do not require either the FDRCSID or the APP_FOLDER calls, but it does no harm to leave them in the trigger.

Add any additional code *before* the supplied text.

Warning: Oracle Corporation does not support modifications to the APP_FOLDER calls in this trigger except for Oracle Applications internal use.

WHEN–NEW–RECORD–INSTANCE

```
APP_STANDARD.EVENT('WHEN-NEW-RECORD-INSTANCE');
```

This trigger manages the state of the Oracle Applications menu and toolbar.

Create block–level triggers as needed, with execution style 'Before'.

Synchronizing (See page 10 – 7)

WHEN–NEW–BLOCK–INSTANCE

```
APP_STANDARD.EVENT('WHEN-NEW-BLOCK-INSTANCE');
```

This trigger manages the state of the Oracle Applications menu and toolbar.

Create block–level triggers as needed, with execution style 'Before'.

Synchronizing (See page 10 – 7)

WHEN–NEW–ITEM–INSTANCE

```
APP_STANDARD.EVENT('WHEN-NEW-ITEM-INSTANCE');
```

This trigger manages the state of the Oracle Applications menu and toolbar.

If you add a flexfields routine call, you should add it before the APP_STANDARD.EVENT call. In general, you should not add any other code to this trigger, as such code would affect every item in the form and could hurt your form performance.

Create block–or item–level triggers as needed, with execution style 'Before'.

Synchronizing (See page 10 – 7)

Triggers That Cannot Be Modified

Oracle Applications does not support the modification of these form–level triggers in any way.

CLOSE_THIS_WINDOW

This trigger invokes APP_CUSTOM.CLOSE_WINDOW from the menu Action–>Close Window.

CLOSE_WINDOW

```
APP_CUSTOM.CLOSE_WINDOW(:SYSTEM.EVENT_WINDOW);
```

This trigger processes all window close events. Code that processes the close window events must reside in the APP_CUSTOM.CLOSE_WINDOW package.

Controlling Window Behavior (See page 7 – 2)

EXPORT

```
app_standard.event('EXPORT');
```

This trigger processes invocation of the "Action, Export" menu choice.

FOLDER_ACTION

```
app_folder.event(:global.folder_action);
```

This trigger processes invocation of entries on the Folder menu.

KEY–COMMIT

```
APP_STANDARD.EVENT('KEY–COMMIT');
```

This trigger processes commits in normal or called forms.

Warning: Oracle Corporation strongly recommends against the use of called forms. This procedure supports them for backward compatibility only.

KEY–EDIT

```
APP_STANDARD.EVENT('KEY–EDIT');
```

This trigger performs flexfield operations, or Calendar or Editor invocation.

KEY–EXIT

```
APP_STANDARD.EVENT('KEY–EXIT');
```

This trigger processes Close events, and leaves enter–query mode.

KEY–HELP

```
APP_STANDARD.EVENT('KEY–HELP');
```

This trigger invokes the Window Help system.

LASTRECORD

```
APP_STANDARD.EVENT('LASTRECORD');
```

This trigger processes the menu event Go–>Last Record.

MENU_TO_APPCORE

```
APP_STANDARD.EVENT(:global.menu_to_appcore);
```

This trigger supports the Special menu.

STANDARD_ATTACHMENTS

```
atchmt_api.invoke;
```

This trigger processes invocation of the Attachments menu entry or toolbar button.

WHEN–WINDOW–CLOSED

```
execute_trigger('CLOSE_WINDOW');
```

This trigger centralizes window close events from the Oracle Applications or Window Manager menu.

WHEN–FORM–NAVIGATE

You cannot modify this referenced trigger. It enables certain standard behaviors, such as normalizing a minimized form when it is navigated to.

To make use of this form event, populate a global variable called GLOBAL.WHEN_FORM_NAVIGATE with the name of a user–named trigger. Usually you populate this global immediately before issuing a GO_FORM.

Passing Instructions to a Form (See page 7 – 19)

ZOOM

```
appcore_custom.event('ZOOM');
```

This trigger processes invocation of the "Action, Zoom" menu choice or toolbar button.

CHAPTER

24 Customization Standards

This chapter provides you with standards for building custom application components that integrate with Oracle Applications. Using these guidelines, you reduce the administrative effort to build and maintain custom components.

Because all Oracle Applications products are built using Oracle Application Object Library, you should use Oracle Application Object Library, with other Oracle tools, to customize Oracle Applications. You should be familiar with the concepts presented in the relevant chapters of this guide.

The following topics are covered:

- Overview of Customizing Oracle Applications
- Customization By Extension
- Customization by Modification
- Oracle Applications Database Customization
- Oracle Applications Upgrades and Patches

Overview of Customizing Oracle Applications

There are several different ways you might want to customize Oracle Applications. Some of the most common types of customizations include:

- Changing forms
  - appearance
  - validation logic
  - behavior
- Changing reports or programs
  - appearance
  - logic
- Database customizations
  - adding read–only schemas
  - augment logic with database triggers
- Integrating third party or custom software
  - relinking programs

Basic Business Needs

These suggestions provide you with the ability to satisfy the following basic business needs. You should be able to:

- Enhance Oracle Applications to meet your needs by:
  - Developing new components
  - Modifying existing components
- Improve the Oracle Applications upgrade process by:
  - Preserving custom components across upgrades
  - Simplifying the upgrade of custom components

Definitions

Customization By Extension

Develop new components for existing Oracle Applications and develop new applications using the development features of Oracle Application Object Library.

Customization by extension can be as simple as copying an existing Oracle Applications component to a custom application directory and modifying the copy.

Customization By Modification

Also known as ”customization in place”. Modify existing Oracle Applications components to meet your specific requirements. Modifications range from changing boilerplate text to altering validation logic.

Attention: Oracle Applications recommends that you avoid doing customization by modification. You should do customization by extension instead. You should be aware that modifications are often not preserved during an upgrade or patch process.

Component

A module of code (such as forms, reports, or SQL*Plus scripts) or an Oracle Application Object Library object (such as menus, responsibilities, and messages), or a database object (such as tables, views, packages, or functions).

Custom Application

A custom application is a non–Oracle Applications application that is registered with Oracle Application Object Library and typically has (at least) its own directory structure and other components.

Overview of Building an Application (See page 1 – 14)
Overview of Setting Up Your Application Framework (See page 2 – 2)

Database Object

A table, index, view, sequence, database trigger, package, grant, or synonym.

Application Short Name

A short reference name for your application that contains no spaces. You use an application short name when you request a concurrent process from a form, call Message Dictionary routines, and so on.

Overview of Setting Up Your Application Framework (See page 2 – 2)

Application Basepath

The name of an environment variable that translates into the top directory of your application's directory tree. Oracle Application Object Library searches specific directories beneath the basepath for your application's executable files, including form files.

Overview of Setting Up Your Application Framework (See page 2 – 2)

Application Directory Structure

The hierarchy of directories for an application. The Oracle Applications directory structures are created when you install or upgrade Oracle Applications. You create the directory structure for a custom application. Consult your Installation Manual for the necessary directories for your platform.

Overview of Setting Up Your Application Framework (See page 2 – 2)
Development Environment
Oracle Application Object Library Reference Material
Oracle Applications Installation Manual

Applications Environment File

Defines the application basepath environment variables for your Oracle Applications. The environment file also defines other environment variables used by Oracle Application Object Library. This file is used by applications system administrators to configure users' environments.

Modify Environment File
Oracle Applications Installation Manual

Determining Your Needs

Follow these steps to determine your customization needs:

- Determine the specific requirement that is not met by Oracle Applications
- Try to meet this requirement by altering the definition parameters of the standard applications, as described in your implementation and user manuals. Often, you can eliminate the need for customization by altering the application configuration (such as setting up a descriptive flexfield)
- Document the requirement that you wish to meet by customization
- Determine if you can meet this requirement by extension (adding a new component) or if you must modify an existing component

Whenever possible, you should customize by extension *rather than by modification*. By doing so, you eliminate the risk of overwriting or losing a required piece of application logic or validation in the existing components.

You may customize by modification, but we strongly discourage this. These modifications introduce a level of risk, and are not supported by Oracle Support Services, nor the Applications Division. A simple change in form navigation may eliminate a validation check resulting in data integrity problems, and may be lost during an upgrade or patch.

If you must modify an Oracle Applications form, your first choice should be to determine whether you can accomplish your modification using the CUSTOM library. You can use the CUSTOM library to modify the behavior of Oracle Applications forms without making invasive changes to the forms code, making your modifications less difficult to upgrade. Modifications you can do using the CUSTOM library include hiding fields, opening other forms from the Zoom button, enforcing business rules, adding choices to the Special menu, and so on.

Using the CUSTOM Library (See page 0 – 1)

Customization By Extension

You separate your application extensions from Oracle Applications components for easy identification and to reduce the risk of loss during an upgrade or patch. To keep new components separate, you implement a custom application and make it the owner of the new components.

You may implement one custom application that owns all custom components, or many custom applications that own custom components. For example, you may want to define a custom general ledger application for extensions to Oracle General Ledger, and a custom payables application for extensions to Oracle Payables. Use many custom applications if you will create more than a few components for each Oracle Application. Use a single custom application if you will only create a few components for all Oracle Applications.

Follow these steps for each custom application you wish to implement:

- Define your custom application
- Create your custom application directory structure
- Add your custom application to your Applications Environment File
- Add new components to your custom application
- Document your new components

Suggestion: Use a revision control system to help you keep track of your changes and custom components.

Overview of Building an Application (See page 1 – 14)
Overview of Setting Up Your Application Framework (See page 2 – 2)

Documenting New Components

You should document at least the following for each new component:

- Purpose
- Input parameters (for reports and programs)
- Sample output (for reports and programs)
- Processing logic

- Database objects used and type of access (select, update, insert, delete)

You thoroughly document each extension to simplify each of the following tasks:

- Component modification due to changing business requirements
- Component modification due to Oracle Applications upgrades or patches
- Identification of obsolete extensions following an Oracle Applications upgrade or patch

If your custom component is a modified copy of an Oracle Applications component, you should list the component in the file **applcust.txt**. This file, located in the $APPL_TOP/admin directory (or platform equivalent), provides a single location for a brief listing of customizations. Oracle Applications uses this file during patch processes (for Release 11.0 and later) to generate warning messages that customizations are being overwritten or may need to be replaced after the patch. Also, you can use the list to help determine the scope of changes that may be needed to customizations after an upgrade or patch.

The **applcust.txt** file provides a place to list the original file name and location, the destination file name and location (the customized file), and a brief comment. You do not need to list components that are not customizations of Oracle Applications components (that is, you do not need to list components of a custom application that did not start with Oracle Applications files).

Defining Your Custom Application

Use the Applications window to register your custom application. Use an intuitive application name and short name for your custom application; relate the names to the Oracle Application being extended if appropriate. For example: extensions to Oracle General Ledger could belong to a custom application named Custom General Ledger with an application shortname XXGL.

Although your short name can be up to 50 characters, we recommend that you use only three to five for ease in maintaining your application and in calling routines that use your short name.

Oracle reserves all three to four character codes starting with the letter O, the letters CP, and the letter E, as well as all names currently used by

Oracle Applications products (query all applications in the Applications window). To reduce the risk that your custom application short name could conflict with a future Oracle Applications short name, we recommend that your custom application short name begins with "XX".

Register Your Application (See page 2 – 3)

Applications Window (See page 2 – 6)

Creating Your Custom Application Directory Structure

Use the appropriate operating system commands to create your application directory structure. You should define a release number as part of the application basepath to allow for multiple versions of your custom application in the future. If your custom application corresponds to a single Oracle Application, use the release number of that Oracle Application. If you use one custom application for multiple Oracle Applications, use the Product Family release number. Note that for Release 11, all products use the Product Family release number.

Development Environment
Oracle Application Object Library Reference Material
Oracle Applications Installation Manual

Modifying Your Applications Environment File

Modify your applications environment file to define an environment variable for your application basepath.

Development Environment
Oracle Application Object Library Reference Material
Oracle Applications Installation Manual

Adding New Components

Each time you develop a new component, place it in the correct subdirectory for the appropriate custom application.

Development Environment
Oracle Application Object Library Reference Material
Oracle Applications Installation Manual

Adding a Form

You build new forms using Oracle Application Object Library with Oracle Forms to ensure seamless integration with Oracle Applications. You must start with the TEMPLATE form and follow the Oracle Applications form development standards described in this manual to ensure user interface consistency between Oracle Applications and your extensions.

Overview of Form Development Steps (See page 1 – 16)

If your extension is a modified Oracle Applications form, you copy the original form and make your modifications to the copy.

Move your completed (or modified) form to the proper directory for your custom application and register the form with Oracle Application Object Library, associating it with your custom application.

Forms Window (See page 11 – 20)

After you have registered your form, you can add it to an existing menu (see Modifying an Existing Menu) or call it from a new menu (see Adding Menus and Responsibilities). You can also connect your form to an Oracle Applications form using Zoom in the CUSTOM library.

Using the CUSTOM Library (See page 0 – 1)

Adding a Report or Concurrent Program

You can write concurrent programs (which include both reports and programs) in Oracle Reports, SQL*Plus, PL/SQL, C, Pro*C, and (on some operating systems) shell scripts. You can run your concurrent programs using the Concurrent Processing features of Oracle Application Object Library to provide the same standard scheduling, prioritization, and specialization features found in Oracle Applications.

You must be familiar with the Concurrent Processing chapter in this manual before writing your concurrent program.

Concurrent Processing (See page NO TAG)

Concurrent Programs (See page 16 – 10)

Adding a New Report Submission Form

Oracle Application Object Library provides you with a Standard Request Submission interface for running and monitoring your application's reports and other programs. You no longer need to design and maintain special forms to submit concurrent programs. Use the Submit Request window to submit reports. If you create custom menus in your application, be sure that the Submit Request window is on your menu.

Overview of Standard Request Submission (See page 21 – 2)

If you want to submit your report or program from a custom form, Oracle Application Object Library provides a standard routine to submit a concurrent program to the concurrent manager from within an Oracle Forms trigger. A custom report submission form should provide validation for each parameter value a user can specify.

Adding Online Help

For Release 11, Oracle Applications provides online help in HTML format. You can easily extend Oracle Applications online help following the guidelines in the *Oracle Applications System Administrator's Guide.* If you extend the online help you will need to repropagate your custom file after upgrading.

Overview of Oracle Applications Help for HTML
Help File Architecture
Customizing Oracle Applications Help
Oracle Applications System Administrator's Guide

If you have built a custom application with custom forms and you want to create context–sensitive online help for your custom application, you can build a help system for your application.

Building Online Help for Custom Applications (See page 24 – 26)

Adding Menus

You can define new menus to call your new forms and Oracle Application menus and forms. We recommend that you always enter your custom application short name as part of the (hidden) menu name when you define new menus to help clarify which menus are yours and to help avoid collision with future Oracle Applications menu names. During an upgrade, all menus created by Oracle Applications are deleted, even if you have modified them or created all new menu options. Patches may also affect Oracle Applications menus.

Overview of Menus and Function Security (See page 11 – 2)

Menus Window (See page 11 – 26)

Adding Responsibilities

You can define new responsibilities using the Responsibilities window in the Oracle Applications System Administrator responsibility. You should create new responsibilities for your custom menus and forms. You can associate these custom responsibilities with your custom application or an Oracle Application. Your custom responsibility is preserved across upgrades, regardless of which application it is associated with.

Responsibilities Window
Oracle Applications System Administrator's Guide

Adding Message Dictionary Messages

You can define your own messages in Message Dictionary. Always associate new messages with your custom application (use your own application name when you define the message). During an upgrade, all custom messages associated with an Oracle Application (and not your custom application) are deleted.

Messages Window (See page 12 – 27)

Customization By Modification

You modify Oracle Application components only when you cannot meet a requirement using Oracle Application features and customization by extension is not an option. You must not modify any component without a thorough understanding of the processing logic and underlying database structure for that component. **Modifications to Oracle Applications components introduce a level of risk, and are not supported by Oracle Support Services, nor the Applications Division.**

If possible, copy the component to be modified into a custom application and follow the guidelines for customization by extension. If you cannot define the modified component in a custom application, you should preserve a copy of the original. An example of a component that cannot be defined in a custom application is a report that is submitted from an Oracle Applications report launch form other than the Submit Request window. When the request to run the report is executed, the concurrent manager expects the report to be in the appropriate Oracle Applications directory.

Documenting Modifications

You should list each component that you modify in the file **applcust.txt**. This file, located in the $APPL_TOP/admin directory (or platform equivalent), provides a single location for a brief listing of customizations. Oracle Applications uses this file during patch processes (for Release 11.0 and later) to generate warning messages that customizations are being overwritten or may need to be replaced after the patch. Also, you can use the list to help determine the scope of changes that may be needed to customizations after an upgrade or patch. The **applcust.txt** file provides a place to list the original file name and location and a brief comment. For customization files that are copies of Oracle Applications files (customization by extension), you also include the destination file name and location (the customized file).

In addition to your list in **applcust.txt**, you should also document at least the following for each component modification:

- Purpose of the modification
- Name of files changed
- Changes to input parameters (for reports and programs)
- Sample output (for reports and programs)

- Changes to processing and validation logic
- Changes to database objects used and type of access (select, update, insert, delete)

Suggestion: Use a revision control system to help you keep track of your changes and custom components.

You thoroughly document each modification to simplify each of the following tasks:

- Further modification due to changing business requirements
- Identification of obsolete modifications following an Oracle Applications upgrade
- Recreating modifications to upgraded Oracle Applications components

Preserving Original Files

You should avoid customizing Oracle Applications files "in place". Always make a copy of the file and modify the copy, preferably in a custom application directory.

Before customization, place a copy of the unmodified Oracle Application component in a directory separate from your Oracle Applications for safekeeping (if you no longer need the modification, you can retrieve the original component). For example, on a UNIX system you create a subdirectory named **orig** beneath your Oracle Applications installation directory (the directory denoted by the $APPL_TOP environment variable, which typically includes a release number designation). The directory **orig** contains application directory structures for each modified component. For example, if you modify Oracle General Ledger Release 11 form GLXSSMHR, you would copy the original versions of GLXSSMHR.fmb and GLXSSMHR.fmx into the directory $APPL_TOP/orig/gl/forms/<language>.

You need to create the application directories only for modified components. For example, if you do not modify any Oracle General Ledger Release 11 Oracle Reports reports, you do not need to create the directory $APPL_TOP/orig/gl/reports/<language>.

You do not need to copy components into the **orig** directory if you copy them into a custom application directory for modification and do not modify the original component in the Oracle Applications directory.

Modifying an Existing Form

Whenever possible, confine your modification of form behavior to the CUSTOM library. If you must modify the form itself, use the following guidelines.

Using the CUSTOM Library (See page 0 – 1)

Oracle Forms .fmb files are provided for all Oracle Applications. For Release 11, all Oracle Applications forms are located in the $AU_TOP/<language> directory. Copy the Oracle Applications form to a custom application for modifications. Follow these steps, using Oracle Forms and Oracle Application Object Library, to modify a form:

1. Identify the file
2. Copy the file to a custom application and rename the file
3 Preserve the original file
4. Make the modifications
5. Comment the form
6. Generate the form
7. Register the customization in **applcust.txt**
8. Document your customization

Identifying the file

Once you select a particular Oracle Applications form for modification, you must identify the underlying form file. You take the following steps to find the file:

- Select Help–>About Oracle Applications from the menu. Scroll to the Form Information section. The form name, followed by .fmb, is the form source file to be modified.
- The first two or three characters of the form file name are the application short name. The file should be located under the forms/[language_code] directory for that application.

Making modifications

Form modifications fall into three categories: cosmetic, navigational, and functional. Cosmetic changes to screen boilerplate text or to the display characteristics of fields will not impact form processing. However, reordering fields on a form, or altering field attributes between ”non–displayed” and ”displayed” (which has the effect of reordering fields) are modifications that should be avoided unless you thoroughly analyze the navigation and trigger firing sequence associated with those fields and ensure that no logic or validation changes will result. You should not remove or disable any form validation logic. You may add validation logic, such as permitting only specific formats or ranges for field entries (though this is best done in the CUSTOM library instead of the form itself).

Commenting the form

Oracle Applications forms provide a routine in the PRE–FORM trigger to document the date and author of form modifications. Oracle Forms also provides the ability to enter form–level comments. You should make use of both of the these features when you modify a form.

The Oracle Applications FND_STANDARD.FORM_INFO routine in the PRE–FORM trigger looks like:

```
FND_STANDARD.FORM_INFO('$Revision: <Number>$',
                       '<Form Name>',
                       '<Application Shortname>',
                       '$Date: <YY/MM/DD HH24:MI:SS> $',
                       '$Author: <developer name> $');
```

You should change the Form Name and Application Shortname arguments to reflect your new file name and custom application. However, this particular application short name affects which online help file the user sees after choosing help for the current window. If you want the user to see the original Oracle Applications online help for the original form, you should keep the original application short name. The online help is the only feature affected by this instance of the application short name.

Each time you modify the form you should update the Date argument (DATE_LAST_MODIFIED) value to the current date and the Author (LAST_MODIFIED_BY) value to indicate who made the modifications. If the Date entry does not exist, add it. You must not update the revision number, as it identifies the original version of the form on which your modified form is based. The date and revision appear in the Help –> About Oracle Applications window.

You should also add a new entry to the form level comments each time you modify the form. Format your form level comments as follows:

```
Developer       Date          Description
--------------- ----------    ------------------------------
```

J. Smith	12–SEP–97	Changed boilerplate in Type region
R. Brown	16–SEP–98	Added ENTRY_STATUS field with LOV to Type region

Modifying an Existing Report

Oracle Reports .rdf files are provided for all Oracle Applications. You should copy the Oracle Application report to a custom application for modification. Follow these steps, using Oracle Reports and Oracle Application Object Library, to modify a report:

1. Identify the file
2. Make the modifications
3. Comment the report
4. Create a concurrent program using your file

Identifying the file

Once you select a particular Oracle Applications report for modification, you must identify the underlying report file. You take the following steps to find the file:

- Query the concurrent program name using the Concurrent Programs window (using either the Application Developer responsibility or the System Administration responsibility). Use the Program field as the query criteria (the program name is the same descriptive name that appears when you submit a request to run that program). Write down the contents of the Name field in the Executable region.
- Navigate to the Concurrent Program Executable window using the Application Developer responsibility. Query the executable using the executable name you obtained from the Concurrent Programs window.
- You identify the report type from the Execution Method field. The file name is the Execution File Name with the appropriate suffix.

Execution	Extension	Subdirectory
SQL*Plus	.sql	sql
Oracle Reports Program	.rdf	reports
SQL*Loader (control file)	.ctl	bin

Table 24 – 1 (Page 1 of 1)

- You should be able to find the report file in the appropriate subdirectory in the directory structure of the application that owns the executable file. However, Oracle Applications programs listed with Spawned or Immediate execution styles are typically C programs, so these programs would not be available for modification. Programs listed with an execution style of PL/SQL Stored Procedure are stored in the database.

Making modifications

You may modify reports that do not alter data without risk of affecting your applications. You should have a thorough understanding of the underlying data structures before modification to ensure your customized report produces valid information (see your Oracle Application's database design manual.) Before you modify a report that alters data you should have a thorough understanding of the report's processing logic. You may add to the validation logic, but you should not remove or disable any validation logic.

Commenting the report

In SQL*Plus and PL/SQL reports, create a comments section to record changes using remark statements and add a comment for each modification you make:

SQL*Plus and PL/SQL:

```
REM   Developer   Date          Description
REM   ----------- ----------    ---------------------
REM   J. Smith    12–SEP–91     Changed column 1 heading
REM   R. Brown    16–SEP–91     Added subtotal row
```

When you change an existing line(s) of code, comment out the original line(s) and include detailed comments about the change in front of the new line of code, the developer's name, and the date. For example:

SQL*Plus and PL/SQL:

```
REM  B. Cleaver  11–OCT–91
REM  Column entered_dr format 99,999.99 heading ’Dr’
REM  Expanded column width and changed heading
Column entered_dr format 9,999,999.99 heading ’Debit Amount’
```

In Oracle Reports reports, add comments using the Report screen. Structure the comments as follows:

```
Developer        Date          Description
-----------            ---------     --------------------
J. Smith         12–SEP–91     Changed column 1 heading
R. Brown         16–SEP–91     Added subtotal row
```

Create a Concurrent Program Using Your File

Define a concurrent program executable, concurrent programs using that executable, and assign the program to report security groups.

Overview of Standard Request Submission (See page 21 – 2)

Modifying an Existing C Program

Oracle does not ship Oracle Applications C source code. You can add new C and Pro*C programs to Oracle Applications.

Modifying an Existing PL/SQL Stored Procedure

Modifying Oracle Applications PL/SQL stored procedures may seriously damage the operation of Oracle Applications. We recommend that you use Customization by Extension to add new stored procedures or database triggers to accomplish your goals, or use the CUSTOM library if possible. **Never alter Oracle Application Object Library objects. Only alter Object Library data using Object Library forms or Object Library programs or supported Object Library APIs**. If you do modify Oracle Applications stored procedures, you may need to reapply your changes each time you upgrade or patch Oracle Applications. Also, if you have a problem that requires assistance from Oracle WorldWide Support, you may need to reverse your changes to help isolate the problem.

Modifying Existing Online Help

For Release 11, Oracle Applications provides online help in HTML format. You can easily modify and extend online help following the guidelines in the *Oracle Applications System Administrator's Guide.* If you modify existing online help you will need to redo your modifications after upgrading. If you extend the online help you will need to repropagate your custom file after upgrading.

Overview of Oracle Applications Help for HTML
Help File Architecture
Customizing Oracle Applications Help
Oracle Applications System Administrator's Guide

Modifying Existing Message Dictionary Messages

You should not modify existing Message Dictionary messages. Use Customization by Extension to add new messages associated with your custom application. You can create new messages under an existing Oracle Application, but they will be deleted or overwritten during an upgrade, and you will have to redo them. New Oracle Application messages that duplicate your message names (associated with Oracle Applications products) will overwrite your messages during an upgrade. Custom messages associated with your custom application are preserved.

If you must alter the existing messages, thoroughly document the changes to recreate them after each upgrade or patch.

Modifying Existing Menus and Responsibilities

You should not modify existing menus and responsibilities. Instead, you should define new menus and responsibilities as described in the Customization by Extension section.

You can use a function security report to help duplicate an existing menu and then modify the copy. You can call Oracle Application menus and sub–menus from your custom application responsibilities and menus. There are exceptional cases when a reference to an Oracle Application menu will become invalid following an upgrade. This occurs when the form, menu, or sub–menu becomes obsolete or its ID changes. You protect against this by running function security reports before upgrading so you have a record of what your menu should call.

Oracle Applications Database Customization

You may alter your applications database by adding objects. You can also modify existing objects, but we strongly discourage this approach. **Any changes made to Oracle Applications database objects are not supported by WorldWide Support or the Applications Division and may conflict with future releases of Oracle Applications.**

Always make a full backup of your database before doing any customization.

Manipulating Oracle Applications Data

We strongly recommend that you do not manipulate Oracle Applications data in any way other than using Oracle Applications. Oracle Applications tables are interrelated. When you use Oracle Applications, any changes made to the data in the Oracle Applications tables are validated, and any relationships are maintained. When you modify Oracle Applications data using SQL*Plus or customized applications components, you are at risk of violating the audit ability and potentially destroying the integrity of your data. You must be aware of the potential damaging problems that improper customization may cause.

Modifying Oracle Application Object Library Database Objects and Data

Never alter Oracle Application Object Library objects. Only alter Object Library data using Object Library forms or Object Library programs or supported Object Libr4ary APIs. Oracle Application Object Library is a data–driven system with complex interrelationships between tables. Any changes you make to Oracle Application Object Library's underlying data may destroy the integrity of your data and the functionality of the application.

Documenting Database Modifications

Define all custom database objects and modifications to existing database objects using SQL*Plus scripts. Place these SQL*Plus scripts in your custom application admin/sql directory. These may include object creation scripts and grant, synonym, view, and object modification scripts. You then use these scripts to rebuild objects following an upgrade or patch and to migrate changes to other Oracle Applications installations.

Defining Your Custom Application ORACLE ID

When you create new database objects, associate them with your custom application. Define an ORACLE schema (ORACLE ID) specifically for your custom application objects. Use your custom application's application short name as the ORACLE schema name for easy identification. You must register your ORACLE schema with Oracle Application Object Library.

Oracle Users Window
Oracle Applications System Administrator's Guide

You must use grants and synonyms to allow other ORACLE schemas to access your custom objects and to allow your custom ORACLE ID access to Oracle Applications objects. When you define a responsibility, you assign a data group to the responsibility. When you use that Responsibility, you connect to its ORACLE schema. Typically, most responsibilities connect to the APPS schema, and you grant privileges on your custom tables to the APPS schema. For information on integrating your custom schema with the APPS schema, see the *Oracle Applications Installation Manual* for your platform.

For example, if you define a custom general ledger application, with a corresponding XXGL ORACLE schema that owns several custom tables, you have two options (Oracle General Ledger is installed in the GL ORACLE schema). If you have relatively few custom tables, and they do not require more security than the Oracle Applications tables, the recommended approach (”tightly integrated with APPS schema”) is:

- Grant privilege on the custom tables from XXGL to APPS
- Create synonyms for XXGL tables in APPS
- Define custom responsibilities that use the APPS ORACLE schema

If you wish to have additional security for the custom tables beyond the security for the Oracle Applications tables, use the following approach instead:

- Grant privilege on the Oracle Application objects from APPS to XXGL
- Create synonyms for GL tables in XXGL to the APPS object of the same name
- Define custom responsibilities using the XXGL schema

Note: Oracle Applications share data among applications. The correct privileges and synonyms are automatically created during an upgrade. Your custom ORACLE schema may need privileges from Oracle Applications other than GL.

Defining Custom Views

If your new or modified code accesses Oracle Applications data, use views to insulate your code from changes in the standard applications database structure. Define views in the APPS schema. If an Oracle Application object definition changes, you may only need to alter the view, rather than altering code.

Adding New Objects

Because Oracle Applications are developed using the ORACLE RDBMS, you can easily extend the database by adding objects and relating them to existing Oracle Applications objects. Use standard naming conventions for the new objects (see Naming Standards), and place the new objects in your custom ORACLE schema or the APPS schema as appropriate. New tables that contain columns used by flexfields or Oracle Alert must be registered with Oracle Application Object Library.

Table Registration API (See page 3 – 11)

Oracle Application Object Library runs forms and programs in the ORACLE schema associated with your current responsibility (usually APPS). Any database objects that need to be accessed must have grants provided to and synonyms created in the ORACLE schema used.

Modifying Oracle Applications Database Objects

You only modify an Oracle Applications database object when you cannot satisfy your needs using flexfields or new database objects. Never drop an object, including columns from tables. If absolutely necessary, alter tables by adding new columns that are defined as null allowed. Always export the table structure before alteration. When upgrading or patching Oracle Applications you will have to ensure that the modified objects will not affect or be affected by the upgrade (see Oracle Applications Upgrades).

Oracle Applications Upgrades and Patches

By following these standards you minimize the impact of an Oracle Applications upgrade or patch on your customizations. During the upgrade process you need to perform specific tasks to preserve your customizations. Many of these tasks are detailed in the *Oracle Applications Installation Manual*. You should review the manual for your specific platform and release, including any release updates, thoroughly before performing an upgrade. A patch may affect customizations less than a full upgrade, but should be given similar attention.

Check Database Modifications

If you have altered Oracle Application database objects, you should unload the new versions of your Oracle Applications from tape (or unload the patch) and review all the scripts in the drivers before upgrading or patching. You must determine if your modifications will affect these scripts. If your modifications will impact the scripts, you must reverse the modifications, run the upgrade or patch, and then reapply the modifications.

If changes to the Oracle Applications database structure affect any views you have created, you will need to modify the views after the upgrade completes. If your customized components access Oracle Applications tables directly, you will need to alter your components if the underlying Oracle Applications data structures change.

Identify Obsolete Customizations

Review each customization and determine if the new release of Oracle Applications satisfies the need that the customization met. If the customization is no longer needed, archive the changes and do not migrate them to the new release.

Migrate Customizations

All changes that are not obsolete must be migrated to the new release. You must migrate all components that were modified in the Oracle Applications directory structure, and you must migrate all components in your custom application directories.

To migrate customized components that you modified in the Oracle Application directory, you must first determine if the unmodified component changed between releases. Compare the original version of the prior release component (in the orig directory) to the new version of the component. If they are different, you have to apply the customizations to the new component (follow the guidelines for modifying an existing component). If the component did not change between releases, you create a copy of the new release of the component in the appropriate orig directory and copy the modified component from the previous release directory to the new release directory.

To migrate your custom applications code, document your modifications to your applications environment file before upgrading. After the upgrade process creates your new applications environment file, you modify the new file.

You can also use Oracle Application Object Library loaders and APIs to extract custom Oracle Application Object Library objects, upgrade, then use the extracted loader scripts to reapply your customizations.

Rerun Grant and Synonym Scripts

After determining which modifications are still valid upon upgrading, you should rerun all of the appropriate grant and synonym scripts for your customizations. These should all be located in the admin/sql directories of your custom applications. This is necessary because the upgrade process may lose grants by dropping and recreating an object for which you had previously granted access to your custom application.

Test All Customizations

As the last step of upgrade confirmation, you should test all of your customized components to ensure they work properly and that no changes to Oracle Applications have affected your modifications.

Building Online Help for Custom Applications

There are three main ways to provide custom online help for your application:

- Customize Oracle Applications help files (not recommended because it will be overwritten upon upgrading or patching, and it could damage the Oracle Applications help)
- Customize custom help files provided by Oracle Applications (recommended for adding custom help to augment existing Oracle Applications help for particular Oracle Applications products)
- Build a custom help system for your application (recommended if you have built completely custom forms or have customized copies of Oracle Applications forms)

For information on customizing the custom help files provided by Oracle Applications, see the *Oracle Applications System Administrator's Guide.* The following sections cover building a custom help system for your custom forms.

How the Help System Works

The Oracle Applications context–sensitive online help system for Release 11 provides context–sensitive help at a window–level granularity (that is, different help for each window in the application) and for individual Standard Request Submission reports and programs. Here is how the context–sensitivity works:

- The user presses the Window Help button or selects Help–>Window from the menu.
- Oracle Applications constructs a target name for the window based on:
  - the name of the form (such as POXACCWO)
  - the name of the window (such as HEADERS)
- Oracle Applications searches for a help link file based on:
  - the directory path specified in the profile option HELP_BASE_URL (set when Oracle Applications help is installed; HELP_BASE_URL is an environment variable in Release 10.7 NCA)
  - the current language

– the application short name specified in the FND_STANDARD.FORM_INFO routine in the form
– the links subdirectory
– a filename matching the name of the form

- The link file provides the correspondence between the target (such as POXACCWO_HEADERS) and the target in the destination HTML file (such as rcvacc07.htm#POXACCWO.headers), and redirects the browser to open that destination file at the specified target

Prepare Your Forms

Verify that your custom forms refer to your custom application short name in the FND_STANDARD.FORM_INFO routine in the PRE–FORM trigger:

```
FND_STANDARD.FORM_INFO('$Revision: <Number>$',
                   '<Form Name>',
                   '<Application Shortname>',
                   '$Date: <YY/MM/DD HH24:MI:SS> $',
                   '$Author: <developer name> $');
```

If you leave the `Application Shortname` value as FND, your user will not see any help, because Oracle Applications will not be able to construct a valid help target.

Create Help Directory

You must create a subdirectory for your help files using your application short name as the subdirectory name. The subdirectory should be located under the online help directory for Oracle Applications. This is the directory path specified in the profile option HELP_BASE_URL (set when Oracle Applications help is installed). The exact location depends on your installation configuration.

Under your help subdirectory, you must also create another subdirectory called links. The links subdirectory holds the link files.

Create HTML Help Files

Create your online help HTML files using your favorite HTML editor. Your help files can contain any links and information you want. To allow them to be called from your custom forms, you must include HTML target tags of the form near the beginning of the file:

```
<A NAME="form_name.window_name"></A>
```

For example, your help file might contain the target:

```
<A NAME="poxaccwo.headers"></A>
```

You can also create context–sensitive help for your Standard Request Submission reports and programs (and include targets for them in your HTML files). You include tags for your reports using the following syntax:

```
<A NAME="srs.report_shortname"></A>
```

For example, your help file might contain the target:

```
<A NAME="srs.poxacrcr"></A>
```

Suggestion: Both file names (on some platforms) and HTML target names are case sensitive, so you must ensure that the case of the HTML targets you specify in your help files matches the case you specify in the links files. To make this easier, we suggest that your help file names and target names are always lowercase (because Oracle Applications help and links files are automatically generated, the case of our targets and links match automatically).

If you want your help to have the same look and feel as Oracle Applications help, you can copy the contents of one of the custom help directories (such as the pocust subdirectory on the Unix platform). The custom help directories contain stub files for custom help (such as pocust.htm) as well as button graphics files and other files. Place the copies in your application's help directory and modify the copies as appropriate. Be sure to modify any filenames, directory paths, links and targets embedded in the .htm files to fit your own application. For example, the Contents button in the pocust.htm file points to the help contents file for Oracle Purchasing.

Note that you do not need either the registry.txt or the ctxtsens.txt files that you might see in an Oracle Applications help directory; these files are byproducts of the process Oracle Applications uses to generate our HTML help system.

We recommend that you have approximately one HTML help file for every window or report in your application; however, this is not required, and you can organize your HTML files however you want.

Place your help files in the subdirectory that is named after your application short name (that is, not in the links subdirectory).

Create Help Links Files

To allow Oracle Applications to find the correct help file for your form when the user chooses the window help button from the toolbar, you must set up links files for your context–sensitive help. The links files store the correspondence between the target and the help file that contains the target. You must have one links file for each form, and the file must be named after the form (for example, poxaccwo.htm for the POXACCWO form). The links file contains link correspondences for all the windows in the form.

You should copy a links file from the Oracle Applications help and modify your copy (recommended because only the actual correspondence lines need to be modified, but the rest of the file is JavaScript code that is exactly the same for every links file). Here is an example links file (poxaccwo.htm). The link lines that you should modify are shown in bold (note that the variable names are uppercase):

```
<HTML><BODY>
<SCRIPT LANGUAGE='JavaScript'>

function onerror(msg, URL, lineNum) {
        var newloc = '../../notfound.htm?' + queryString
        location.replace(newloc)
        return true
}

var POXACCWO_HEADERS = 'rcvacc07.htm#poxaccwo.headers'
var POXACCWO_LINES = 'rcvacc07.htm#poxaccwo.lines'
var POXACCWO_QUERY_FIND = 'rcvacc07.htm#poxaccwo.query_find'

var queryString =
location.href.substring(location.href.indexOf('?')+1,locatio
n.href.length)
var newloc = '../' + eval(queryString)
location.replace(newloc)
```

```
</SCRIPT>
</BODY></HTML>
```

You can also create context–sensitive help for your Standard Request Submission reports and programs (and include targets for them in your HTML files). You must create a separate links file for your reports and call that file srs.htm. You then list correspondences for all of your SRS reports in that file, as shown in the following example:

```
<HTML><BODY>
<SCRIPT LANGUAGE='JavaScript'>

function onerror(msg, URL, lineNum) {
        var newloc = '../../notfound.htm?' + queryString
        location.replace(newloc)
        return true
}

var SRS_POASLUPG = 'athb03.htm#srs.poaslupg'
var SRS_POCFH = 'cthh07.htm#srs.pocfh'
var SRS_POCISO = 'cthh03.htm#srs.pociso'
...
var SRS_POCISO = 'cthh03.htm#srs.pociso'

var queryString =
location.href.substring(location.href.indexOf('?')+1,locatio
n.href.length)
var newloc = '../' + eval(queryString)
location.replace(newloc)

</SCRIPT>
</BODY></HTML>
```

Modify Library File

You may want to add a link to your custom application help in the Oracle Applications online help library file (library.htm). Note that this file will be overwritten upon upgrade (or possibly upon patching) and you will need to add your link again after upgrading, so save a copy of your modifications for future reference.

Upgrades and Patches

Because installation of the Oracle Applications help system is a simple installation of the entire directory and file structure for Oracle Applications help, your custom help directories will be overwritten upon upgrading or patching of the online help. Before upgrading or patching, you should be sure to preserve a copy of your custom help directory structures in a safe place before the upgrade or patch so you can move them back into position after your upgrade or patch.

☞ **Attention:** The help system mechanism is subject to change for Release 12 or later, and you may need to revise your help system when you upgrade.

CHAPTER

25

Using the CUSTOM Library

This chapter describes the architecture and implementation details for the CUSTOM library. The CUSTOM library allows you to write custom extensions, such as Zooms and business rule logic, to Oracle Applications.

The following topics are covered:

- Customizing Oracle Applications with the CUSTOM Library
- Writing Code for the CUSTOM Library
- Events Passed to the CUSTOM Library
- When to Use the CUSTOM Library
- CUSTOM Library Package Procedures
- Support and Upgrading

Customizing Oracle Applications with the CUSTOM Library

The CUSTOM library allows extension of Oracle Applications without modification of Oracle Applications code. You can use the CUSTOM library for customizations such as Zoom (such as moving to another form and querying up specific records), enforcing business rules (for example, vendor name must be in uppercase letters), and disabling fields that do not apply for your site.

You write code in the CUSTOM library, within the procedure shells that are provided. All logic must branch based on the form and block for which you want it to run. Oracle Applications sends events to the CUSTOM library. Your custom code can take effect based on these events.

Attention: The CUSTOM library is provided for the exclusive use of Oracle Applications customers. The Oracle Applications products do not supply any predefined logic in the CUSTOM library other than the procedure shells described here.

Writing Code for the CUSTOM Library

The CUSTOM library is an Oracle Forms PL/SQL library. It allows you to take full advantage of all the capabilities of the Developer/2000 suite of products, and integrate your code directly with Oracle Applications without making changes to Oracle Applications code.

The CUSTOM library is located in the $AU_TOP/res/plsql directory (or platform equivalent). The CUSTOM library you modify must replace the default CUSTOM library in this directory in order for your code to take effect.

After you write code in the CUSTOM procedures, compile and generate the library using Oracle Forms. Then place this library into $AU_TOP/res/plsql directory (or platform equivalent). Subsequent invocations of Oracle Applications will then run this new code.

Warning: If there is a .plx (compiled code only) for a library, Oracle Forms always uses the .plx over the .pll. A .plx is only created when you generate a library using the Oracle Forms generator (using the parameter COMPILE_ALL set to Yes), not when you compile and save using the Oracle Forms Designer. Therefore, either delete the .plx file (so your code runs directly from the .pll file) or create your own .plx file using the Oracle Forms generator.

The specification of the CUSTOM package in the CUSTOM library cannot be changed in any way. You may add your own packages to the CUSTOM library, but any packages you add to this library must be sequenced after the CUSTOM package. To ensure that your packages remain sequenced after the CUSTOM package even after a conversion from a .pld file, when program units are alphabetized, we recommend you name your packages with characters that come after C (for example, we recommend you name your own packages with names that begin with USER_).

Coding Considerations and Restrictions

Be aware of the open form environment in which Oracle Applications operate. Also, each running form has its own database connection.

The following considerations and restrictions apply to the CUSTOM library *and* any libraries you attach to CUSTOM:

- You cannot use any SQL in the library. However, you can use a record group to issue SELECT statements, and you can use calls to stored procedures for any other DML operations such as updates, inserts, or deletes.
- When a global library (CUSTOM, APPCORE, FNDSQF, VERT, GLOBE and other Oracle Applications libraries, as well as any library you attach to CUSTOM) calls a stored procedure, that same stored procedure cannot then be invoked from a non–global context (such as a form or a non–global library). If you want to call an Oracle Applications stored procedure (for example, FND_REQUEST.SUBMIT_REQUEST) from the CUSTOM library, you should first write your own stored procedure that is a cover routine of the Oracle Applications stored procedure (but has a different name), and you should call your cover routine from CUSTOM instead of the Oracle Applications stored procedures. Oracle Applications stored procedures are typically documented as ”server–side” procedures.
- Values of PL/SQL package variables (variables listed in the package specification) do not change between calls to the library, no matter which form calls the library.
- Startup code (PL/SQL code in a package but not contained in a function or procedure) executes only once, when the package is first called (CUSTOM is loaded only once in an Oracle Applications session, the first time it is called).

- Oracle Forms global variables in your code are visible to all running forms.

Attaching Other Libraries to the CUSTOM Library

You may attach other libraries to the CUSTOM library. However, you cannot attach the APPCORE library to CUSTOM because it would cause a recursion problem (because CUSTOM is attached to APPCORE).

As of Oracle Applications Release 10SC Production 15 and later, the CUSTOM library comes with the FNDSQF library already attached. FNDSQF provides Oracle Applications routines for function security (for opening forms), flexfields, and other utility routines.

Altering Oracle Applications Code

Frequently you need to know the names of blocks and items within Oracle Applications forms for your CUSTOM logic. You should use the Examine feature available on the Help–>Tools menu while running the form of interest; it will give you easy access to all object names. You should not open Oracle Applications forms in the Oracle Forms Designer to learn this information.

You should exercise caution when changing any properties or values of items in the form from which CUSTOM logic is invoked. The CUSTOM library is intended to be a mechanism to augment Oracle code with your own. Using the CUSTOM library to alter Oracle code at runtime may bypass important validation logic and may jeopardize the integrity of your data. You should thoroughly test all logic you add to the CUSTOM library before using it in a production environment.

Following Coding Standards in the CUSTOM library

Within the CUSTOM library, you are free to write almost any code supported by the Developer/2000 toolset, so long as you follow all Oracle Applications coding standards. However, there is an exception: you cannot call any APPCORE routines from CUSTOM. Almost all APPCORE routines start with the prefix ”APP”.

Where you would normally use the Oracle Applications routine APP_ITEM_PROPERTY.SET_PROPERTY, you should use the Oracle Forms built–in SET_ITEM_PROPERTY. See: Setting Properties in the CUSTOM Library: page 25 – 9

If you use Zoom or the CUSTOM library to invoke forms that you have developed, those forms must adhere completely to all of the Oracle Applications coding standards.

Attention: To invoke another form, use the function security routines. Do not use CALL_FORM since the Oracle Applications libraries do not support it.

Events Passed to the CUSTOM Library

The CUSTOM library receives two different kind of events, generic and product–specific. Generic events are common to all the forms in Oracle Applications. These events are:

- WHEN–FORM–NAVIGATE
- WHEN–NEW–FORM–INSTANCE
- WHEN–NEW–BLOCK–INSTANCE
- WHEN–NEW–RECORD–INSTANCE
- WHEN–NEW–ITEM–INSTANCE
- WHEN–VALIDATE–RECORD
- SPECIALn (where n is a number)
- ZOOM
- EXPORT

Logic you code for WHEN–FORM–NAVIGATE, WHEN–NEW–BLOCK–INSTANCE, WHEN–NEW–RECORD–INSTANCE, or WHEN–NEW–ITEM–INSTANCE fires after any existing logic in those triggers for the form, block or item.

Logic you code for WHEN–NEW–FORM–INSTANCE fires during the call to APP_STANDARD.EVENT. That call may be anywhere within existing WHEN–NEW–FORM–INSTANCE logic in the form.

Logic you code for WHEN–VALIDATE–RECORD fires during the call to APP_STANDARD.EVENT or FND_FLEX.EVENT. One of those calls may be within existing WHEN–VALIDATE–RECORD logic in the form or block, depending on how the form was originally coded.

Logic you code for SPECIALn, where n is a number, fires before any logic in the existing SPECIALn trigger (if there is one).

The ZOOM event occurs when the user invokes Zoom from the menu (Action–>Zoom) or the toolbar. The EXPORT event occurs after an export operation is complete (Action–>Export).

The CUSTOM library also receives some product–specific events associated with the business rules of that product (for example, the NAVIGATE event in Oracle Human Resources). Please refer to the *Open Interfaces Manual* for your Oracle Applications product to see what product–specific events, if any, are passed to CUSTOM.

When to Use the CUSTOM Library

There are several main cases for which you can code logic using the CUSTOM library. Each of these cases must be coded differently.

- **Zoom**—The addition of user–invoked logic on a per–block basis. A Zoom typically consists of opening another form and (optionally) passing parameter values to the opened form through the Zoom logic.
- **Logic for generic events**—Augment Oracle Applications logic for certain generic form events such as WHEN–NEW–FORM–INSTANCE or WHEN–VALIDATE–RECORD.
- **Logic for product–specific events**—Augment or replace Oracle Applications logic for certain product–specific events that enforce business rules.
- **Custom entries for the Special menu**—Add entries to the Special menu for Oracle Applications forms, such as an entry that opens a custom form.

Coding Zoom

Zoom allows the addition of user–invoked logic on a per–block basis. For example, you may want to allow access to the Vendors form from within the Enter Purchase Order form while the user is in the PO Header block of that form. You can enable Zoom for just that block, and when the user invokes it, you can open the Vendors form.

Only Oracle Applications customers use the Zoom feature; Oracle Applications products do not ship any predefined Zoom logic. Note that most Zooms that were predefined in the character–mode Oracle

Applications Release 10 and earlier have been incorporated into Oracle Applications forms as buttons or windows for Release 11. In many cases, redesign of forms for Release 11 eliminated the need for predefined Zooms. Also, the native GUI environment allows users to cut–and–paste data between forms directly instead of relying on Release 10 Zooms to copy the data.

Zoom for Release 11 behaves as follows:

- Oracle Applications provides a menu entry and a button on the toolbar for the user to invoke Zoom when available. The button and the menu entry are disabled unless Zoom logic has been defined in the CUSTOM library for that form and block.
- Whenever the cursor changes blocks in the form, the form calls the ZOOM_AVAILABLE function in the CUSTOM library (via APPCORE). If this function returns TRUE, then the Zoom entries on the menu and toolbar are enabled; if it returns FALSE, then they are disabled.
- If the Zoom entries are enabled, then when the user invokes Zoom the form calls the Zoom event code in the CUSTOM library. You write code for this event that branches based on the current form and block.

▶ **To code Zooms into the CUSTOM library:**

1. Add a branch to the CUSTOM.ZOOM_AVAILABLE function that specifies the form and block where you want a user to be able to invoke Zoom. See: CUSTOM.ZOOM_AVAILABLE: page 25 – 13
2. Add a branch to the CUSTOM.EVENT procedure for the ZOOM event.

   Inside that branch, specify the form and block where you want a user to be able to invoke Zoom. Add the logic you want to occur when the user invokes Zoom. See: CUSTOM.EVENT: page 25 – 14

Coding Generic Form Events

You can code logic that fires for a particular form and block at a particular form event. You can code logic for the following events:

- WHEN–FORM–NAVIGATE
- WHEN–NEW–FORM–INSTANCE
- WHEN–NEW–BLOCK–INSTANCE

- WHEN–NEW–RECORD–INSTANCE
- WHEN–NEW–ITEM–INSTANCE
- WHEN–VALIDATE–RECORD
- SPECIALn (where n is a number)
- ZOOM
- EXPORT

Some Oracle Applications forms, such as most Oracle Human Resources forms, may provide additional events that call the CUSTOM library. These additional events are listed in the documentation for the product that owns the form. You can code logic in the CUSTOM library for such events the same way you would code logic for generic form events.

▶ **To code logic for generic form events into the CUSTOM library:**

1. Add a branch to the CUSTOM.EVENT procedure for the particular event you want.

   Inside that branch, specify the form and block where you want your logic to occur. Add the logic you want to occur when that event fires for the form and block you specify. See: CUSTOM.EVENT: page 25 – 14

Coding Product–Specific Events

Please refer to the *Open Interfaces Manual* or *User's Guide* for your Oracle Applications product to see what product–specific events, if any, are passed to CUSTOM.

▶ **To code logic for product–specific events into the CUSTOM library:**

1. Add a branch to the CUSTOM.STYLE function that specifies the execution style (before, after, override, or standard) you want for your product–specific event logic. You can only specify one of the styles supported for that particular product–specific event. See: CUSTOM.STYLE: page 25 – 13
2. Add a branch to the CUSTOM.EVENT procedure for the particular product–specific event you want. See: CUSTOM.EVENT: page 25 – 14

Within that branch, add logic for that specific business function See: CUSTOM.EVENT: page 25 – 14

Adding Custom Entries to the Special Menu

▶ **To code logic for Special menu entries into the CUSTOM library:**

1. Add a branch to the CUSTOM.EVENT procedure for the WHEN–NEW–FORM–INSTANCE event. See: CUSTOM.EVENT: page 25 – 14

   Inside that branch, specify the form and block where you want your logic to occur. Add the logic to set up the Special menu. See: Application–Specific Entries: The Special Menu: page 10 – 7

   You should start with entry number 15 on the Special menu (SPECIAL15) to avoid conflicting with any Special menu entries already provided by the form or by the VERT or GLOBE libraries (vertical and globablization routines) and work backwards towards SPECIAL1.

2. Add a branch to the CUSTOM.EVENT procedure for the particular SPECIALn event you want (where n is a number between 1 and 15). See: CUSTOM.EVENT: page 25 – 14

   Inside that branch, specify the form and block where you want your logic to occur. Add the logic you want to fire from the Special menu. This is the logic that, if you were developing the form yourself, you would put in the user–named SPECIALn trigger. See: Application–Specific Entries: The Special Menu: page 10 – 7

   At the end of your SPECIALn logic, you must include a call to RAISE FORM_TRIGGER_FAILURE; this ensures that any SPECIALn trigger in the form that may already exist (with the same number) does not fire after your logic has fired, or, if there is no corresponding trigger, that your users do not see an error message when the form cannot find such a trigger.

Setting Properties in the CUSTOM Library

Because you cannot call any APPCORE routines from CUSTOM, you must use the Oracle Forms built–in routines SET_ITEM_PROPERTY and GET_ITEM_PROPERTY instead of the Oracle Applications routines APP_ITEM_PROPERTY.SET_PROPERTY and APP_ITEM_PROPERTY.GET_PROPERTY. The use of properties differs slightly between the Oracle Forms and Oracle Applications routines, so here is a list of what properties you should use where they differ.

Substitutes for APP_ITEM_PROPERTY.SET_PROPERTY

ALTERABLE Call set_item_property(*item_name*, ALTERABLE_PLUS, PROPERTY_ON / OFF);

ENTERABLE Call set_item_property(*item_name*, ENTERABLE_PLUS, PROPERTY_ON / OFF);

If the item is a text or list item, you'll need to set the visual attribute for the item as well. If you're turning ENTERABLE on, call display_item(*item_name*, ''); If you're turning ENTERABLE off, call display_item(*item_name*, 'DISABLED_TEXT');

DISPLAYED When turning DISPLAYED on, call set_item_property(item_name, DISPLAYED_PLUS, PROPERTY_ON);

When turning DISPLAYED off, call set_item_property(*item_name*, DISPLAYED, PROPERTY_OFF);

ENABLED If the item is a text or list item, call set_item_property(*item_name*, ENTERABLE_PLUS, PROPERTY_ON / OFF);

If you are turning ENABLED on (and the item is a text or list item), you should also call set_item_property(*item_name*, VISUAL_ATTRIBUTE, 'DATA');

If you are turning ENABLED off (and the item is a text or list item), you should also call set_item_property(*item_name*, VISUAL_ATTRIBUTE, 'DISABLED_TEXT');

If the item is not a text or list item, call set_item_property(*item_name*, ENABLED, PROPERTY_ON / OFF);

If you turn ENABLED on, and the item is not a display, text, or list item, you should also set NAVIGABLE on. If you're turning ENABLED on, and the item is not a display item, text item, list item or button, set UPDATEABLE on.

All other properties Call set_item_property(*item_name*, *property*, PROPERTY_ON / OFF);

Substitutes for APP_ITEM_PROPERTY.GET_PROPERTY

Note that although app_item_property.get_property returns the constants PROPERTY_ON and PROPERTY_OFF, get_item_property returns the strings 'TRUE' and 'FALSE'.

ALTERABLE Call get_item_property(*item_name*, ALTERABLE_PLUS);

ENTERABLE Call get_item_property(*item_name*, ENTERABLE_PLUS);

ENABLED For text or list items, call get_item_property(*item_name*, ENTERABLE_PLUS);

If the item is not a text or list item, call get_item_property(*item_name*, ENABLED);

All other properties Call get_item_property(*item_name*, *property*);

Support and Upgrading

To manage your customizations and handle upgrade considerations follow these guidelines:

Trouble with Forms Operating with the CUSTOM Library

If a form is operating incorrectly, and you have coded CUSTOM library or Zoom logic for it, use the menu to disable the CUSTOM library code temporarily (Help–>Tools–>Custom Code–>Off) so you can determine whether the problem comes from the customizations or Oracle Applications code.

Upgrading

An Oracle Applications upgrade will overwrite any modifications you have made to the CUSTOM library, so you must keep a backup copy of CUSTOM with the changes you make. Replace the default CUSTOM library with your custom version after the upgrade.

Remember, form and block names may change after an upgrade to Oracle Applications. You should test any custom logic that you have defined to confirm that it still operates as intended before using it in a production environment.

CUSTOM Package

The CUSTOM package contains the following functions and procedure:

- CUSTOM.ZOOM_AVAILABLE
- CUSTOM.STYLE
- CUSTOM.EVENT

CUSTOM.ZOOM_AVAILABLE

Summary

```
function custom.zoom_available return BOOLEAN;
```

Description

If Zoom is available for this block, then return TRUE; otherwise return FALSE. Always test for the form and block name. Refer to the SYSTEM variables for form name and block name in your code and branch accordingly. The module name of your form must match the form file name.

By default this routine must return FALSE.

Example Code

The following example enables Zooms in the following places:

Form: FNDSCAUS, Block USER and

Form: FNDCPMCP, Block PROCESS

```
FUNCTION zoom_available RETURN BOOLEAN IS
  form_name  VARCHAR2(30) := NAME_IN('system.current_form');
  block_name VARCHAR2(30) := NAME_IN('system.cursor_block');
BEGIN
  IF (form_name = 'FNDSCAUS' AND block_name = 'USER') OR
     (form_name = 'FNDCPMCP' AND block_name = 'PROCESS')THEN
     RETURN TRUE;
  ELSE
     RETURN FALSE;
  END IF;
END zoom_available;
```

CUSTOM.STYLE

Summary

```
function custom.style(event_name varchar2) return integer;
```

Description This function allows you to determine the execution style for some product–specific events. You can choose to have your code execute before, after, or in place of the code provided in Oracle Applications. See the *Open Interface Manual* for your Oracle Applications product for a list of events that are available through this interface. Note that some product–specific events may not support all execution styles. CUSTOM.STYLE does not affect generic form events or Zoom.

Any event that returns a style other than custom.standard must have corresponding code in custom.event which will be executed at the time specified.

The following package variables should be used as return values:

- custom.before
- custom.after
- custom.override
- custom.standard

By default this routine must return custom.standard (which means that there is no custom event code).

☞ **Attention:** Oracle Corporation reserves the right to pass additional values for event_name to this routine, so all code must be written to branch on the specific event_name passed.

Example Code The following example sets up the MY_PRICING_EVENT event to have the Override execution style.

```
begin
  if event_name = ’MY_PRICING_EVENT’ then
    return custom.override;
  else
    return custom.standard;
  end if;
end style;
```

CUSTOM.EVENT

Summary `procedure custom.event(event_name varchar2);`

Description This procedure allows you to execute your code at specific events. Always test for event name, then for form and block name within that event. Refer to the SYSTEM variables for form name and block name in your code and branch accordingly. The module name of your form must match the form file name.

Always use FND_FUNCTION.EXECUTE to open a new session of a form. Do not use CALL_FORM or OPEN_FORM. The form function must already be defined with Oracle Application Object Library and added to the menu (without a prompt, if you do not want it to appear in the Navigator).

By default, this routine must perform ”null;”.

> **Attention:** Oracle Corporation reserves the right to pass additional values for event_name to this routine, so all code must be written to branch on the specific event_name passed.

Example Code

The following example contains logic for a Zoom, a product–specific event, and a generic form event.

The Zoom event opens a new session of a form and passes parameter values to the new session. The parameters already exist in the form being opened, and the form function has already been defined and added to the menu (without a prompt, so it does not appear in the Navigator).

```
procedure event(event_name varchar2) is

    form_name      varchar2(30) := name_in('system.current_form');
    block_name     varchar2(30) := name_in('system.cursor_block');
    param_to_pass1 varchar2(255);
    param_to_pass2 varchar2(255);
  begin
    if (event_name = 'ZOOM') then
      if (form_name = 'DEMXXEOR' and block_name = 'ORDERS') then
        /* The Zoom event opens a new session of a form and
        passes parameter values to the new session.  The
        parameters already exist in the form being opened:*/
        param_to_pass1 := name_in('ORDERS.order_id');
        param_to_pass2 := name_in('ORDERS.customer_name');
        fnd_function.execute(FUNCTION_NAME=>'DEM_DEMXXEOR',
                             OPEN_FLAG=>'Y',
                             SESSION_FLAG=>'Y',
                             OTHER_PARAMS=>'ORDER_ID="'||
                             param_to_pass1||
                             '" CUSTOMER_NAME="'||
                             param_to_pass2||'"');
         /* all the extra single and double quotes account for
         any spaces that might be in the passed values */
      end if;
```

```
    elsif (event_name = 'MY_PRICING_EVENT') then
       /*For the product-specific event MY_PRICING_EVENT, call a
         custom pricing routine */
      get_custom_pricing('ORDERS.item_id', 'ORDERS.price');

    elsif (event_name = 'WHEN-VALIDATE-RECORD') then
      if (form_name = 'APXVENDR' and block_name = 'VENDOR') then
        /* In the WHEN-VALIDATE-RECORD event, force the value of
           a Vendor Name field to be in uppercase letters */
        copy(upper(name_in('VENDOR.NAME')), 'VENDOR.NAME');
      end if;

    else
      null;
    end if;
  end event;
end custom;
```

CHAPTER

26

APPCORE Routine APIs

This chapter provides you with specifications for calling many Oracle Applications APIs from your PL/SQL procedures. Most routines in the APPCORE library are described in this section. Some APPCORE routines are described in other chapters (for example, the APP_SPECIAL routines are described in the chapter called ”Controlling the Toolbar and the Default Menu”). The routines described in this chapter include:

- APP_COMBO: Combination Block API
- APP_DATE: Date Conversion APIs
- APP_EXCEPTION: Exception Processing APIs
- APP_FIELD: Item Relationship Utilities
- APP_FIND: Query Find Utilities
- APP_ITEM: Individual Item Utilities
- APP_ITEM_PROPERTY: Property Utilities
- APP_RECORD: Record Utilities
- APP_REGION: Region Utilities
- APP_STANDARD Package
- APP_WINDOW: Window Utilities

APP_COMBO: Combination Block API

Use APP_COMBO to control navigation in combination blocks.

APP_COMBO.KEY_PREV_ITEM

Summary `procedure APP_COMBO.KEY_PREV_ITEM;`

Location APPCORE library

Description Call this procedure in the KEY–PREV–ITEM trigger to provide the standard behavior when back–tabbing from the first item in a record. This procedure ensures that the cursor automatically moves to the last item of the previous record.

Combination Blocks (See page 7 – 8)

APP_DATE: Date Conversion APIs

You can use the APP_DATE package utilities to format, convert, or validate dates. In working with dates, remember that you can associate a format to the date as a DATE datatype and a format as a VARCHAR2 datatype. Most of these utilities associate the NLS format to the date value; others will use the "DD–MON–YY" format. For date/time values, referred to as DATETIME here, the utilities append 'HH24:MI:SS' to the regular date format.

APP_DATE.DATE_TO_CHARDATE

Summary

```
function APP_DATE.DATE_TO_CHARDATE(
               dateval date)
               return varchar2;
```

Location APPCORE library

Description This function converts a date value to a character value and applies the expanded NLS format (DD–MON–RRRR in Releases 10.7 and 11). You would use this routine when passing a date value as a token for a message using the Message Dictionary routines.

Arguments (input)

dateval The date to be converted.

Example

```
fnd_message.set_TOKEN('ORDER_DATE',
 app_date.date_to_chardate(:ORDERS.DATE_ORDERED),
 FALSE);
```

APP_DATE.DATE_TO_CHARDT

Summary

```
function APP_DATE.DATE_TO_CHARDT(
               dateval date)
               return varchar2;
```

Location APPCORE library

Description This function converts a DATETIME date value to a character value and applies the expanded NLS format (DD–MON–RRRR HH24:MI:SS in Releases 10.7 and 11). You would use this routine when passing a datetime value as a token for a message using the Message Dictionary routines.

Arguments (input)

dateval The date to be converted.

Example

```
fnd_message.set_TOKEN('ORDER_DATE',
 app_date.date_to_chardt(:ORDERS.DATE_ORDERED),
 FALSE);
```

APP_DATE.DATE_TO_FIELD

Summary

```
procedure APP_DATE.DATE_TO_FIELD(
                    dateval date,
                    field varchar2)
```

Location APPCORE library

Description This procedure copies a date or DATETIME value into a date field. Use this routine when you need to work with a date field in a library. Use this routine instead of `copy(to_char(date,` 'DD–MON–YYYY {HR24:MI:SS}',*'my_block.my_date_field'*`),`

Arguments (input)

dateval The date to be copied.

field The date field to which the date is copied.

Example

```
app_date.date_to_field(my_date_value,
'my_block.my_date_field');
```

APP_DATE.FIELD_TO_DATE

Summary

```
function APP_DATE.FIELD_TO_DATE(
                    field varchar2 default NULL)
                    return date;
```

Location APPCORE library

Description This function converts a date or DATETIME field value to a character value in 'DD–MON–RRRR {HH24:MI:SS}' format. Use this routine when you need to work with a date field in a library. Use this routine instead of `to_date(name_in(`*'my_block.my_date_field'*`),` `'DD–MON–YYYY {HH24:MI:SS}');`

Arguments (input) **field** The date field to be converted.

Example

```
my_date =
app_date.field_to_date('my_block.my_date_field');
```

APP_EXCEPTION: Exception Processing APIs

You should use the APPCORE package APP_EXCEPTION to raise exceptions in the PL/SQL procedures written for your forms.

APP_EXCEPTION.RAISE_EXCEPTION

Summary

```
procedure  APP_EXCEPTION.RAISE_EXCEPTION(
            exception_type varchar2 default null,
            exception_code number   default null,
            exception_text varchar2 default null);
```

Location APPCORE library and database (stored procedure)

Description This procedure stores exception information and raises exception form_trigger_failure.

Arguments

exception_text Additional context information.

exception_type Error prefix that specifies error type (for example, ORA or APP)

exception_code The number that identifies the error.

APP_EXCEPTION.RETRIEVE

Summary

```
procedure  APP_EXCEPTION.RETRIEVE;
```

Location APPCORE library and database (stored procedure)

Description This procedure retrieves exception information from the database.

APP_EXCEPTION.GET_TYPE

Summary

```
function  APP_EXCEPTION.GET_TYPE return varchar2;
```

Location APPCORE library and database (stored function)

Description This function returns the exception type.

APP_EXCEPTION.GET_CODE

Summary `function APP_EXCEPTION.GET_CODE return number;`

Location APPCORE library and database (stored function)

Description This function returns the exception code.

APP_EXCEPTION.GET_TEXT

Summary `function APP_EXCEPTION.GET_TEXT return varchar2;`

Location APPCORE library and database (stored function)

Description This function returns the exception text.

APP_EXCEPTION.RECORD_LOCK_EXCEPTION

Description This is a predefined exception. Call it in an exception handler to handle cases where the record cannot be locked. It is usually used with the APP_EXCEPTION.RECORD_LOCK_ERROR procedure.

APP_EXCEPTION.RECORD_LOCK_ERROR

Summary

```
procedure  APP_EXCEPTION.RECORD_LOCK_ERROR (
           counter  IN   OUT   number);
```

Description This procedure asks the user to continue or cancel an attempt to lock a record. It returns to the calling procedure to try again if the user continues. It displays an ”Unable to reserve record” acknowledgement and raises FORM_TRIGGER_FAILURE if the user cancels.

APP_EXCEPTION.RECORD_LOCK_ERROR only asks the user every two attempts to lock a record (e.g., counter = 2, 4, 6, etc.). The calling procedure should attempt to lock the record in a loop and call RECORD_LOCK_ERROR in a WHEN APP_EXCEPTION.RECORD_LOCK_EXCEPTION exception handler inside the loop. If the user continues, RECORD_LOCK_ERROR returns and the loop repeats. If the user cancels, RECORD_LOCK_ERROR raises FORM_TRIGGER_FAILURE and the loop exits.

Arguments	**counter**	Maintained by RECORD_LOCK_ERROR to count the attempts to lock a record. Calling procedure should initialize to null or 0.

APP_EXCEPTION.DISABLED

Summary

```
procedure  APP_EXCEPTION.DISABLED;
```

Description

This procedure rings the bell. Call this procedure to disable simple functions (typically in a KEY– trigger).

APP_FIELD: Item Relationship Utilities

This section describes utilities you can use to maintain relationships between your form items.

APP_FIELD.CLEAR_FIELDS

Summary

```
procedure  APP_FIELD.CLEAR_FIELDS(
           field1  varchar2,
           field2  varchar2 default NULL,
           field3  varchar2 default NULL,
           field4  varchar2 default NULL,
           field5  varchar2 default NULL,
           field6  varchar2 default NULL,
           field7  varchar2 default NULL,
           field8  varchar2 default NULL,
           field9  varchar2 default NULL,
           field10 varchar2 default NULL);
```

Description This procedure clears up to ten items if the items are not NULL and are not check boxes or required lists.

APP_FIELD.CLEAR_DEPENDENT_FIELDS

Summary

```
procedure  APP_FIELD.CLEAR_DEPENDENT_FIELDS(
          master_field varchar2,
           field1  varchar2,
           field2  varchar2 default NULL,
           field3  varchar2 default NULL,
           field4  varchar2 default NULL,
           field5  varchar2 default NULL,
           field6  varchar2 default NULL,
           field7  varchar2 default NULL,
           field8  varchar2 default NULL,
           field9  varchar2 default NULL,
           field10 varchar2 default NULL);
```

Description This procedure clears up to ten dependent items if the master item is NULL and the dependent items are not NULL and not check boxes or required lists.

Arguments (input)

master_field	Name of master item
field1 ... field10	Name of dependent item(s).

APP_FIELD.SET_DEPENDENT_FIELD

Summary

```
procedure  APP_FIELD.SET_DEPENDENT_FIELD(
           event           varchar2,
           master_field    varchar2,
           dependent_field varchar2
           invalidate      boolean    default  FALSE);

procedure  APP_FIELD.SET_DEPENDENT_FIELD(
           event           varchar2,
           condition       boolean,
           dependent_field varchar2
           invalidate      boolean    default  FALSE);
```

Description This procedure makes an item enterable or not enterable based on whether the master item is NULL or a specified condition is TRUE, and clears the field. The dependent item can be a text item, check box, or poplist.

You typically call this procedure in the following triggers:

- WHEN–VALIDATE–ITEM on the master field
- WHEN–VALIDATE–ITEM on the field(s) the condition is based on or in event INIT on the dependent field
- PRE–RECORD
- POST–QUERY (required only when the dependent item is in a multi–record block)

For examples on using this procedure, see:

Item Relations (See page 9 – 2)
Mutually Inclusive Items with Dependent Items (See page 9 – 13)
Defaults (See page 9 – 18)

Arguments (input)

event — Name of trigger event. If you call this trigger on a master field, pass VALIDATE instead of the trigger name (which may be WHEN–VALIDATE–ITEM, WHEN–CHECKBOX–CHANGED, WHEN–LIST–CHANGED, or WHEN–RADIO–CHANGED, any of which can also be used).

master_field — Name of master item

condition — TRUE when dependent item is to be enabled

dependent_field — Name of dependent item

invalidate — If TRUE, mark the item as invalid instead of clearing the dependent item. Set this flag to TRUE if the dependent item is a required list or option group.

APP_FIELD.SET_EXCLUSIVE_FIELD

Summary

```
procedure  APP_FIELD.SET_EXCLUSIVE_FIELD(
             event          varchar2,
             field1          varchar2,
             field2          varchar2,
             field3          varchar2  default NULL);
```

Description

This procedure coordinates items so that only one item of a set may contain a value. If a value is entered in one of the items, the other items are cleared and made non–NAVIGABLE (users can still mouse into these items). This procedure only covers sets of two or three mutually–exclusive items.

For examples on using this procedure, see:

Mutually Exclusive Items (See page 9 – 10)

Arguments (input)

event — Name of trigger event (WHEN–NEW–RECORD–INSTANCE, PRE–RECORD, or VALIDATE. VALIDATE is generally used in place of WHEN–VALIDATE–ITEM, WHEN–RADIO–CHANGED, WHEN–LIST–CHANGED, or

WHEN–CHECKBOX–CHANGED, any of which can also be used.)

field1 Name of exclusive item (BLOCK.FIELD)

field2 Name of exclusive item (BLOCK.FIELD)

field3 Name of exclusive item (BLOCK.FIELD, optional)

APP_FIELD.SET_INCLUSIVE_FIELD

Summary

```
procedure  APP_FIELD.SET_INCLUSIVE_FIELD(
           event          varchar2,
           field1          varchar2,
           field2          varchar2,
           field3          varchar2  default NULL,
           field4          varchar2  default NULL,
           field5          varchar2  default NULL);
```

Description

This procedure coordinates up to five items so that if any of the items contains a value, then all of the items require a value. If all of the items are NULL, then the items are not required.

For examples on using this procedure, see:

Mutually Inclusive Items (See page 9 – 12)

Arguments (input)

event Name of trigger event (WHEN–NEW–RECORD–INSTANCE, PRE–RECORD, or VALIDATE. VALIDATE is generally used in place of WHEN–VALIDATE–ITEM, WHEN–RADIO–CHANGED, WHEN–LIST–CHANGED, or WHEN–CHECKBOX–CHANGED, any of which can also be used.)

field1 Name of inclusive item

field2 Name of inclusive item

field3 Name of inclusive item (optional)

field4 Name of inclusive item (optional)

field5 Name of inclusive item (optional)

APP_FIELD.SET_REQUIRED_FIELD

Summary

```
procedure  APP_FIELD.SET_REQUIRED_FIELD(
           event          varchar2,
           condition      boolean,
           field1          varchar2,
           field2          varchar2  default NULL,
           field3          varchar2  default NULL,
           field4          varchar2  default NULL,
           field5          varchar2  default NULL);
```

Description

This procedure makes an item required or not required based on whether a specified condition is true.

For examples on using this procedure, see:

Conditionally Mandatory Items (See page 9 – 15)

Arguments (input)

event	Name of trigger event
condition	True when items should be required
field1	Name of item
field2	Name of item
field3	Name of item (optional)
field4	Name of item (optional)
field5	Name of item (optional)

APP_FIND: Query–Find Utilities

Use the following routines to implement the Find Window functionality.

Query Find Windows (See page 8 – 2)

APP_FIND.NEW

Summary

```
procedure  APP_FIND.NEW(
                block_name varchar2);
```

Description

This routine is called by the "New" button in a Find Window to return the user to a new record in the block on which the find is based.

Arguments (input)

block_name — The name of the block the Find Window is based on

APP_FIND.CLEAR

Summary

```
procedure  APP_FIND.CLEAR;
```

Description

This routine is called by the "Clear" button in a Find Window to clear the Find Window.

APP_FIND.CLEAR_DETAIL

Summary

```
procedure  APP_FIND.CLEAR_DETAIL(
                detail_block    varchar2);
```

Description

This routine clears the result block of a find *block* (not a Find window). This action can only be performed from triggers that allow navigation.

Arguments (input)

detail_block — The name of the block to be cleared

Example

```
APP_FIND.CLEAR_DETAIL('MYBLOCK');
```

APP_FIND.FIND

Summary

```
procedure  APP_FIND.FIND(
           block_name    varchar2);
```

Description This routine is called by the "Find" button in a Find Window to execute the Find.

Arguments (input)

block_name The name of the block the Find Window is based on

APP_FIND.QUERY_RANGE

Summary

```
procedure  APP_FIND.QUERY_RANGE(
           low_value    varchar2/date/number,
           high_value   varchar2/date/number,
           db_item_name varchar2);
```

Description This utility constructs the query criteria for ranges in a Find Window. Depending on the datatype of the low and high value, it creates a range of characters, dates, or numbers.

Arguments (input)

low_value The low value of the range

high_value The high value of the range

db_item_name The name of the item in the block that is being queried

APP_FIND.QUERY_FIND

Summary

```
procedure  APP_FIND.QUERY_FIND(
           lov_name varchar2);

procedure APP_FIND.QUERY_FIND(
          block_window    varchar2,
          find_window     varchar2,
          find_block      varchar2);
```

Description These routines invoke either the Row–LOV or the Find Window. Call them from a user–named trigger "QUERY_FIND."

Arguments (input)

lov_name	The name of the Row–LOV
block_window	The name of the window the Find Window is invoked for
find_window	The name of the Find Window
find_block	The name of the block in the Find Window

APP_ITEM: Individual Item Utilities

This section describes utilities for managing your items individually.

APP_ITEM.COPY_DATE

Summary

```
procedure  APP_ITEM.COPY_DATE(
           date_val  varchar2
           item_name varchar2);
```

Description

Use this procedure to copy a hardcoded date value into a field. This routine does the copy in this way:

```
copy(to_char(to_date('01–01–1900', 'DD–MM–YYYY'),
                'DD–MON–YYYY'), 'bar.lamb');
```

date_val A character date, expressed in the format 'DD–MM–YYYY'

item_name The name of the item to copy the value into, expressed as block.item.

APP_ITEM.IS_VALID

Summary

```
procedure  APP_ITEM.IS_VALID(
           val          varchar2
           dtype        varchar2  default 'DATE');

function APP_ITEM.IS_VALID(
           val          varchar2
           dtype        varchar2  default 'DATE')
```

```
return BOOLEAN;
```

Description Use this routine with fields that are of character datatype but contain data in the format of a date, number or integer. The procedure raises an error if the value is not in a valid format for the specified datatype. The function returns TRUE if the value is in the correct format, otherwise FALSE

val Value to be checked

dtype Datatype value should use: DATE, INTEGER, or NUMBER.

APP_ITEM.SIZE_WIDGET

Summary

```
procedure  APP_ITEM.SIZE_WIDGET(
          wid_name varchar2.
          max_width number default 20);
```

Description This procedure reads the current label for the specified widget and resizes the widget to fully show the label (used to ensure proper size for a translated label). It will not make the widget any larger than the maximum width specified, to prevent overlapping or expansion beyond the edge of the screen. Call this procedure only for check boxes in single–record formats, buttons and radio groups.

wid_name Name of the widget in block.field syntax

max_width The maximum size to make the widget, in inches

APP_ITEM_PROPERTY: Property Utilities

These utilities help you control the Oracle Forms and Oracle Applications properties of your items.

APP_ITEM_PROPERTY.GET_PROPERTY

Summary

```
function  APP_ITEM_PROPERTY.GET_PROPERTY(
          item_name varchar2,
          property number)
          return number;

function  APP_ITEM_PROPERTY.GET_PROPERTY(
          item_id item,
          property number)
           return number;
```

Description

This function returns the current setting of an item property. It differs from the Oracle Forms's get_item_property in that it returns PROPERTY_ON or PROPERTY_OFF instead of 'TRUE' or 'FALSE'.

Item Properties (See page 6 – 20)

APP_ITEM_PROPERTY.SET_PROPERTY

Summary

```
procedure  APP_ITEM_PROPERTY.SET_PROPERTY(
           item_name varchar2,
           property varchar2,
           value number);

procedure  APP_ITEM_PROPERTY.SET_PROPERTY(
           item_id item,
           property varchar2,
           value number);
```

Description

This procedure sets the property of an item. You should never use the Oracle Forms built–in SET_ITEM_PROPERTY to set the field properties DISPLAYED, ENABLED, ENTERABLE, ALTERABLE, INSERT_ ALLOWED, UPDATEABLE, NAVIGABLE, REQUIRED, and

ICON_NAME directly. Use APP_ITEM_PROPERTY.SET_PROPERTY instead.

APP_ITEM_PROPERTY.SET_PROPERTY remaps some properties to do other things like change visual attributes. Also, there are some properties that APP_ITEM_PROPERTY provides that native Oracle Forms does not.

Arguments (input)

item_name Name of the item to apply the property attribute to. Specify both the block and item name. You can supply the item_ID instead of the name of the item.

property The property to set

value Either PROPERTY_ON or PROPERTY_OFF, or an icon name (to change the icon property)

Item Properties (See page 6 – 20)

APP_ITEM_PROPERTY.SET_VISUAL_ATTRIBUTE

Summary

```
procedure  APP_ITEM_PROPERTY.SET_VISUAL_ATTRIBUTE(
           item_name varchar2,
           property number
           value number);
```

Description This procedure sets the visual attributes for an item.

Arguments (input)

item_name Name of the item to apply the visual attribute to

property There are two valid item properties: ENABLED and ENTERABLE. ENABLED sets the visual attribute for all records in a multi–record block. ENTERABLE sets the value for the current record only.

value Either PROPERTY_TRUE or PROPERTY_FALSE

APP_RECORD: Record Utilities

Following are utilities that interact with a block at the record level.

APP_RECORD.HIGHLIGHT

Summary

```
procedure  APP_RECORD.HIGHLIGHT(
           value        varchar2/number);
```

Description

This call changes the visual attribute of the current record. To highlight data, pass 'SELECTED_DATA'. To turn off highlighting, pass 'DATA'. You can pass the name of any visual attribute you want to apply.

Suggestion: To improve performance for large blocks with many hidden fields, position all hidden fields at the end of the block, and place a non–database item named "APPCORE_STOP" before the hidden items. When APP_RECORD.HIGHLIGHT reaches this field, it stops highlighting.

Arguments (input)

value The name of the visual attribute you want to apply.

Visual Attributes
Oracle Applications User Interface Standards

APP_RECORD.FOR_ALL_RECORDS

Summary

```
procedure  APP_RECORD.FOR_ALL_RECORDS(
           block_name      varchar2,
           trigger_name     varchar2);

procedure APP_RECORD.FOR_ALL_RECORDS(
           trigger_name     varchar2);
```

Description

This call executes a user–named trigger for every record of the current block or the specified block. IF you specify a block, the GO_BLOCK built–in fires. When finished, the cursor returns to the original record and item.

If the trigger fails, FORM_TRIGGER_FAILURE is raised and the cursor is left in the record in which the failure occurred.

You can pass arguments to the user–named trigger using global variables.

APP_RECORD.FOR_ALL_RECORDS fires once when there are no queried records.

Arguments (input)

block_name The name of the block to navigate to

trigger_name Name of the trigger to execute

APP_RECORD.DELETE_ROW

Summary

```
procedure  APP_RECORD.DELETE_ROW(
          check_delete       BOOLEAN   default FALSE,
          product_name       varchar2  default NULL,
          message_name       varchar2  default NULL);

function  APP_RECORD.DELETE_ROW(
          check_delete       BOOLEAN   default FALSE,
          product_name       varchar2  default NULL,
          message_name       varchar2  default NULL)
          return BOOLEAN;
```

Description

This call provides a generic message to insure that the user really intends to delete the row.

If the function is called, it does not delete the row but returns TRUE if the user responds with a confirmation to delete the record and FALSE otherwise. If you have a complex delete, you can first confirm that the user wants the delete the record.

If the procedure is called, the record is deleted if the user responds affirmatively. You should provide your own message when there is more than one block that allows delete.

Arguments (input)

check_delete Forces block DELETE_ALLOWED to be checked (optional)

product_name The product shortname needed if passing your own message (optional)

message_name The name of the message (optional)

APP_RECORD.VALIDATE_RANGE

Summary

```
procedure  APP_RECORD.VALIDATE_RANGE(

      from_item     varchar2,
      to_item       varchar2,
      range_name    varchar2  default NULL,
      event_name    varchar2  default 'WHEN-BUTTON-PRESSED',
      dtype         varchar2  default 'DATE',
      product_name  varchar2  default NULL,
      message_name  varchar2  default NULL);
```

Description

This call validates a range to assure that the ”from” value is less than the ”to” value. Call this routine from the WHEN–BUTTON–PRESSED trigger of a Find button, or a WHEN–VALIDATE–RECORD trigger (for example).

If the range is invalid, the routine attempts to navigate to the beginning of the range. If you call VALIDATE_RANGE from a trigger that does not allow navigation, then provide a range name to the routine. Include the range name in your default message.

Arguments (input)

from_item — The block.item of the from value

to_item — The block.item of the to value

range_name — Name of the range (optional)

event_name — Trigger name, used to determine if navigation is possible (optional)

dtype — Datatype of the range (NUMBER or DATE, defaults to DATE)

product_name — The product shortname needed if passing your own message, if necessary (optional)

message_name — The name of the message, if necessary (optional)

APP_REGION: Region Utilities

Following are utilities used with alternative regions.

APP_REGION.ALT_REGION

Summary

```
function  APP_REGION.ALT_REGIONS(
              poplist_name varchar2)
                  return BOOLEAN;
```

Description

Takes the values currently in the poplist identified by *poplist_name* and shows them in LOV 'appcore_alt_regions' (referenced in from APPSTAND automatically). If the user selects a row from the LOV, the corresponding poplist value will be updated and TRUE will be returned; otherwise no changes will be made to the poplist and this will return FALSE. Used for keyboard support of alternative region control.

Arguments (input)

poplist_name The control poplist for an alternative region ('block.field' format).

Example

```
if APP_REGION.ALT_REGIONS('CONTROL.LINE_REGIONS')
then
    CONTROL.LINE_REGIONS('WHEN-LIST-CHANGED');
end if;
```

Alternative Regions (See page 7 – 12)

APP_STANDARD Package

APP_STANDARD.APP_VALIDATE

Summary

```
procedure  APP_STANDARD.APP_VALIDATE (scope    NUMBER);
```

Description

This procedure acts similarly to Oracle Form's built–in Validate, except that it navigates to the first item that caused the validation failure, and it provides support for the button standard. Use it instead of the Oracle Form built–in.

Arguments (input)

scope	The scope of the validation. Valid values are DEFAULT_SCOPE, FORM_SCOPE, BLOCK_SCOPE, RECORD_SCOPE, and ITEM_SCOPE.

APP_STANDARD.EVENT

Summary

```
procedure  APP_STANDARD.EVENT (
                         event_name       varchar2);
```

Description

This procedure invokes the standard behavior for the specified event. The TEMPLATE form contains all necessary calls to this trigger.

Arguments (input)

event_name	Name of the event or trigger

Special Triggers in the TEMPLATE form (See page 23–6)

APP_STANDARD.SYNCHRONIZE

Summary

```
procedure  APP_STANDARD.SYNCHRONIZE;
```

Description

Dynamic changes to the form can affect which menu items apply, although the state of the menu items is not re–evaluated automatically. If you make a change that affects which items in the toolbar and menu can be used, call this routine, and it re–evaluates the menu and toolbar. (For example, changing the INSERTABLE property of a block, changing the status of a record, etc.)

Pulldown Menus and the Toolbar (See page 10 – 2)

APP_STANDARD.PLATFORM

Summary

```
APP_STANDARD.PLATFORM varchar2(30);
```

Description

This package variable stores the name of the value returned by GET_APPLICATION_PROPERTY(USER_INTERFACE). Valid values include ’MACINTOSH’, MSWINDOWS’, MSWINDOWS32’, and ’MOTIF’.

Example

```
if APP_STANDARD.PLATFORM = ’MSWINDOWS’ then
    MDI_height := get_window_property(FORMS_MDI_WINDOW,
                                      HEIGHT);
end if;
```

Global Values (See page 1 – 6)

APP_WINDOW: Window Utilities

The following utilities operate at the window level.

APP_WINDOW.CLOSE_FIRST_WINDOW

Summary

```
procedure  APP_WINDOW.CLOSE_FIRST_WINDOW;
```

Description

This call exits the form. It raises FORM_TRIGGER_FAILURE if it fails.

APP_WINDOW.PROGRESS

Summary

```
procedure  APP_WINDOW.PROGRESS( percent    number);
```

Description

This call manages all aspects of the progress_indicator object. If it is not already visible, the call opens and centers the window. When the percent >= 99.9, the window automatically closes. For any other percent, the progress bar resizes (with a four inch wide maximum).

Arguments (input)

percent A number between 0 and 99.9, expressing the amount competed.

APP_WINDOW.SET_COORDINATION

Summary

```
procedure  APP_WINDOW.SET_COORDINATION(
                event           varchar2,
                coordination    varchar2,
                relation_name   varchar2);
```

Description

This call sets the deferred coordination attribute of a relation to ON or OFF based on the state of the coordination check box. The check box is either ”DEFERRED” or ”IMMEDIATE.”

For a closed window, the relation is always ”DEFERRED.”

When coordination is set to ”DEFERRED,” AutoQuery is turned on.

Master–Detail Relations (See page 7 – 6)

Arguments (input)

event The name of the trigger event (either WHEN–CHECKBOX–CHANGED, WHEN–WINDOW–CLOSED, or OPEN–WINDOW)

coordination IMMEDIATE or DEFERRED. Defaults to IMMEDIATE

relation_name Name of the relation, which is listed in the Oracle Forms Designer under the Block object in the Relation object

APP_WINDOW.SET_WINDOW_POSITION

Summary

```
procedure  APP_WINDOW.SET_WINDOW_POSITION(
          child         varchar2,
          rel           varchar2,
          parent        varchar2    default  NULL);
```

Description

This call positions a window in the following styles:

- CASCADE
- RIGHT
- BELOW
- OVERLAP
- CENTER
- FIRST_WINDOW

If the window was open but obscured, this call raises the windows. If the window was minimized, the call normalizes it.

If system resources are low (especially on MS Windows), a warning message appears.

Non–Modal Windows (See page 5 – 3)

Arguments (input)

child The name of the window to be positioned

rel The style of the window’s position

parent Name of the window to relative to which you want to position the child window

APP_WINDOW.SET_TITLE

Summary

```
procedure  APP_WINDOW.SET_TITLE(
    window_name  varchar2,
    session      varchar2,
    instance1    varchar2 default 'APP_ARGUMENT_NOT_PASSED',
    instance2    varchar2 default 'APP_ARGUMENT_NOT_PASSED',
    instance3    varchar2 default 'APP_ARGUMENT_NOT_PASSED');
```

Description Use this utility to set the title of a window in the standard format.

Non–Modal Windows (See page 5 – 3)

Arguments (input)

window_name The name of the window to set the title for

session General session information (for example, Org, Set of Books), no more than 64 characters

instance[1,2,3] Context information from the master record (optional). The combined length should be no more than 250 characters.

CHAPTER

27

FNDSQF Routine APIs

This chapter provides you with specifications for calling several Oracle Applications APIs from your PL/SQL procedures. Most routines in the FNDSQF library are described in this section. Some FNDSQF routines are described in other chapters (for example, the FND_KEY_FLEX routines are described in the chapter called "Flexfields"). The routines described in this chapter include:

- FND_CURRENCY: Dynamic Currency APIs
- FND_GLOBAL: WHO Column Maintenance
- FND_ORG: Organization APIS
- FND_STANDARD: Standard APIs
- FND_UTILITIES: Utility Routines

FND_CURRENCY: Dynamic Currency APIs

This section describes Dynamic Currency APIs that you can use in your client– and server–side PL/SQL procedures. The Dynamic Currency feature allows different values in arbitrary currencies to be displayed in the same report or form, each shown with appropriate formatting.

FND_CURRENCY.GET_FORMAT_MASK (Client or Server)

Summary

```
function  FND_CURRENCY.GET_FORMAT_MASK(
            currency_code  IN varchar2
            field_length   IN number)
return varchar2;
```

Description This function uses the normal default values to create a format mask.

Arguments (input)

currency_code The currency code for which you wish to retrieve a default format mask

Arguments (input)

field_length The maximum number of characters available to hold the formatted value

Attention: The varchar2 field that receives the format mask should be ten characters longer than the field_length.

This routine uses the following profiles to create the format mask:

- CURRENCY:THOUSANDS_SEPARATOR
- CURRENCY:NEGATIVE_FORMAT
- CURRENCY:POSITIVE_FORMAT

Although the profile for negative format allows three different bracket styles, this routines only uses angle brackets (< >).

Currency Examples

Client–side PL/SQL Example

The ORDER_LINES.AMOUNT field in a form is to be displayed using Dynamic Currency formatting. The format mask is created and passed into the APP_ITEM_PROPERTY.SET_PROPERTY call:

```
APP_ITEM_PROPERTY.SET_PROPERTY('ORDER_LINE.AMOUNT',
                  FORMAT_MASK,
                  FND_CURRENCY.GET_FORMAT_MASK(
                    :ORDER_CURRENCY_CODE,
                     GET_ITEM_PROPERTY(
                                  'ORDER_LINE.AMOUNT',
                                  MAX_LENGTH)));
```

The use of the display group separator, and positive and negative formatting are typically user preferences. Thus these settings are allowed to default from the User Profile system. The precision comes from the stored information for that order's currency code.

Server–side PL/SQL Example

Dynamic currency support is also accessible from server–side PL/SQL. The package interfaces are identical. An example implementation has the following calls:

```
DISPLAYABLE_VALUE := TO_CHAR(AMOUNT,
                        FND_CURRENCY.GET_FORMAT_MASK(
                                     DC_FORMAT_MASK, 30));
```

FND_GLOBAL: WHO Column Maintenance

This section describes Global APIs you can use in your server–side PL/SQL procedures. The server–side package FND_GLOBAL returns the values of system globals, such as the login/signon or "session" type of values. You need to set Who columns for inserts and updates from stored procedures. Although you can use the FND_GLOBAL package for various purposes, setting Who columns is the package's primary use.

You should not use FND_GLOBAL routines in your forms (that is on the client side), as FND_GLOBAL routines are stored procedures in the database and would cause extra roundtrips to the database. On the client side, most of the procedures in the FND_GLOBAL package are replaced by a user profile option with the same (or a similar) name. You should use FND_PROFILE routines in your forms instead.

Tracking Data Changes with WHO (See page 3 – 2)
FND_PROFILE: User Profile APIs (See page 13 – 8)

FND_GLOBAL.USER_ID (Server)

Summary

```
function  FND_GLOBAL.USER_ID
         return number;
```

Description Returns the user ID.

FND_GLOBAL.LOGIN_ID (Server)

Summary

```
function  FND_GLOBAL.LOGIN_ID
          return number;
```

Description Returns the login ID (unique per signon).

FND_GLOBAL.CONC_LOGIN_ID (Server)

Summary

```
function  FND_GLOBAL.CONC_LOGIN_ID
          return number;
```

Description Returns the concurrent program login ID.

FND_GLOBAL.PROG_APPL_ID (Server)

Summary

```
function  FND_GLOBAL.PROG_APPL_ID
          return number;
```

Description Returns the concurrent program application ID.

FND_GLOBAL.CONC_PROGRAM_ID (Server)

Summary

```
function  FND_GLOBAL.CONC_PROGRAM_ID
          return number;
```

Description Returns the concurrent program ID.

FND_GLOBAL.CONC_REQUEST_ID (Server)

Summary

```
function  FND_GLOBAL.CONC_REQUEST_ID
          return number;
```

Description Returns the concurrent request ID.

FND_ORG: Organization APIs

Use this package to set the correct Organization in forms that use organizations.

FND_ORG.CHANGE_LOCAL_ORG

Summary

```
function  FND_ORG.CHANGE_LOCAL_ORG return boolean;
```

Description

Use this function to change the organization of the current form. It returns FALSE if the change is cancelled or fails.

FND_ORG.CHANGE_GLOBAL_ORG

Summary

```
function  FND_ORG.CHANGE_GLOBAL_ORG return boolean;
```

Description

Use this function to change the global organization defaults when opening a new form. It returns FALSE if the change is cancelled or fails.

FND_ORG.CHOOSE_ORG

Summary

```
procedure  FND_ORG.CHOOSE_ORG(
           allow_cancel   IN    boolean    default FALSE);
```

Description

Call this procedure in PRE–FORM to ensure the organization parameters are set. If the local form has no organization parameters passed, the global defaults are used. If the global organization defaults are not set, the procedure opens the organization LOV to force an organization selection.

allow_cancel Allow cancelation of the LOV without forcing a choice. The default is FALSE.

FND_STANDARD: Standard APIs

This section describes utilities you can use to achieve standard functionality in your forms.

FND_STANDARD.FORM_INFO

Summary

```
procedure  FND_STANDARD.FORM_INFO(
           version                  IN varchar2,
           title                    IN  varchar2,
           application_short_name   IN varchar2,
           date_last_modified       IN varchar2,
           last_modified_by         IN varchar2);
```

Description

FND_STANDARD.FORM_INFO provides information about the form. Call it as the first step in your WHEN–NEW–FORM–INSTANCE trigger. The TEMPLATE form provides you with a skeleton call that you must modify.

Special Triggers in the TEMPLATE form (See page 23–6)

FND_STANDARD.SET_WHO

Summary

```
procedure  FND_STANDARD.SET_WHO;
```

Description

SET_WHO loads WHO fields with proper user information. Call in PRE–UPDATE, PRE–INSERT for each block with WHO fields. You do not need to call FND_GLOBAL if you use SET_WHO to populate your WHO fields.

Tracking Data Changes With WHO (See page 3 – 2)
FND_GLOBAL:WHO Column Maintenance (See page 27 – 4)

FND_STANDARD.SYSTEM_DATE

Summary

```
function  FND_STANDARD.SYSTEM_DATE return date;
```

Description

This function behaves exactly like the built–in SYSDATE, only cached for efficiency. You should use it in your Oracle Forms PL/SQL code in place of the built–in SYSDATE.

FND_STANDARD.USER

Summary

```
function  FND_STANDARD.USER return varchar2;
```

Description

This function behaves exactly like the built–in USER, only cached for efficiency. You should use it in your Oracle Forms PL/SQL code in place of the built–in USER.

FND_UTILITIES: Utility Routines

This section describes various utility routines.

FND_UTILITIES.OPEN_URL

Summary

```
procedure  OPEN_URL(URL in varchar2);
```

Description

Invokes the Web browser on the client computer with the supplied URL document address. If a browser is already running, the existing browser is directed to open the supplied URL. You can use this utility to point a Web browser to a specific document from your forms.

Arguments (input)

URL — You can pass either an actual URL string or a *:block.field* reference of a field that contains a URL string.

Example 1

```
FND_UTILITIES.OPEN_URL('http://www.oracle.com/index.html');
```

Example 2

```
FND_UTILITIES.OPEN_URL(:blockname.fieldname);
```

FND_UTILITIES.PARAM_EXISTS

Summary

```
function  PARAM_EXISTS(name varchar2) return boolean;
```

Description

Returns true if the parameter exists in the current form.

Arguments (input)

name — The name of the parameter to search for.

Example

```
if fnd_utilities.param_exists('APP_TRACE_TRIGGER') then
   execute_trigger(name_in('PARAMETER.APP_TRACE_TRIGGER'));
end if;
```

CHAPTER

28 Naming Standards

This chapter provides you with information you need to define all your database and form objects. It provides naming standards and header information for all your objects and files.

The naming standards are grouped into the following sections:

- Database objects
- Form objects
- File standards
- PL/SQL Packages and Procedures
- Reserved Words

Naming Standards and Definitions

In general, make names meaningful and brief. Do not use generic, all–purpose phrases like ”COMMON”, ”MISC”, ”OTHER”, or ”UTILITY” in the name.

Database Objects

Include header information when you create your objects. The header should include the following documentation

- Name
- Purpose
- Arguments

  Arg1 Describe arg1

  Arg2 Describe arg2

- Notes

  1. Special usage notes
  2. Special usage notes

- History

  DD–MON–YY J. Doe Created

Tables

Standard *prod_objects*

prod is the product short name, and *objects* is the name of the objects stored in the table and should be plural. The table name should be 20 characters or less. It can be longer, but you will need to abbreviate it for the table handler package name, which must be 24 characters or less.

Example PO_VENDORS, AS_LOOKUPS

Views

Standard *table*_V or *table_criteria*_V

table is the name of the root table the view is based on. The criteria is a qualifier of the objects shown by the view. Use the criteria qualifier if using table name alone is not unique, the view is based on a join of 2 or more tables, the view contains a WHERE clause, or the view is unusual.

Example PO_VENDORS_CHICAGO_V, AS_LOOKUPS_RANK_V

Triggers

Standard *table*_Ti

table is the name of the table on which the trigger is based, and *i* is a unique id starting at 1.

Example PO_HEADERS_T1

Synonyms

Standard *table*

Your synonym should have the same name as the table or view it is based on.

Using two different names (the synonym name and the underlying object name) is confusing. If you change object names, you should clean up your code, instead of creating synonyms.

Constraints

Primary Key	*table*_PK
Unique	*table*_Ui
Foreign Key	*table*_Fi
Check	Use Message Dictionary message naming standards

Overview of Message Dictionary (See page 12 – 2)

table is the name of the table on which the constraint is created, while *i* is a unique id starting at 1. You should name all of your constraints so that you can enable or disable them easily.

Packages

Standard *prod_module* or *prod_description*

prod is the product short name, *module* is a functional module, and *description* is a one or two word explanation of the purpose. Stored package names should be 24 characters or less. For library packages, your package should be unique within 27 characters. Wrapper packages use a three character prefix. Select a description that helps simplify the names of procedures in the package.

Example OE_SCHEDULE, AOL_FLEXFIELD

Packaged Procedures

Standard *verb_noun*

verb_noun is a brief explanation of the purpose. Do not reuse the product short name or any part of the package name in the procedure name. Remember that you will invoke the procedure as package procedure. For example, if the package name is APP_ORDER_BY, then the procedures should simply be named APPEND and REVERT. Be careful you don't name your package procedure a SQL, PL/SQL, Oracle Forms, or other reserved word, or redefine an Oracle Forms built–in.

Example CALCULATE_PRICE_VARIANCE, TERMINATE_EMPLOYEE

Table Handler Package and Procedures

Package *table*_PKG

table is the name of the table on which the package acts (to insert, update, delete, or lock a record, or to check if a record in another table references a record in this table). The package name should be 24 characters or less.

Example PO_LINES_PKG

Private Packages

Standard *package*_PRIVATE

package is the name of the owning package.

Example APP_ITEM_PROPERTY_PRIVATE

Forms PL/SQL Program Units (Stand–Alone Procedures)

Standard *prod_verb_noun*

prod is the product short name, and *verb_noun* is a brief explanation of the purpose.

Example AP_INITIALIZE_FORM

PL/SQL Variables

Standard *variable* or *X_variable*

variable should be a logical, meaningful, and concise name. Precede variable name with X when the variable is used in a SQL statement, so that there is no possibility of conflicts with database object names or confusion as to which are PL / SQL variables and which are database objects.

Example X_header_id

PL/SQL Global Variables

Standard G_*variable*

variable should be a logical, meaningful, and concise name. Precede variable name with G to distinguish global variables from local variables.

Example G_set_of_books_id

Form Objects

In general, objects that can show multiple items (record groups, LOVs, etc.) should have plural names, while singular objects (modules, blocks) have singular names.

Modules

Standard *file name*

Your form module name should match your form file name. For example, if a form is called POXPOMPO.fmb, make sure the Module Name (visible in the Designer) is POXPOMPO. This is especially important if you reference objects from your form. ZOOM also relies on the Module Name being correct.

Record Groups

Standard *object*

object is the name of the object that the record group contains.

Oracle Forms Global Variables

Standard *prod_variable*

prod is the product short name, and *variable* is the name you would normally give to the variable.

Example PO_SECURITY_LEVEL, MFG_ORGANIZATION

Item

Use logical, meaningful, and concise names. Note that table columns based on LOOKUP_CODES should have a "_CODE" or "_FLAG" suffix, and the displayed meaning item should have the same name but without the suffix.

Mirror Items use the name of the item plus a "_mir" suffix. So if the item in the detail portion is "ename", name the mirror–item "ename_mir".

Blocks

Standard *case_short_name* or *object*

case_short_name is the CASE block short name, and *object* is the name of the objects in the block. The block name should be 14 characters or less.

Example ORDER, LINES

Special Blocks

Block containing toolbar	TOOLBAR
Block containing control items	CONTROL
Block containing display–only, context info	CONTEXT
Blocks to submit concurrent requests	*program* or *report*
Non–database blocks (such as search blocks)	*action* or *action_object*

If the block is shared with other forms, make the block name unique by preceding it with the name of your form.

Canvasses

Standard *object*

object is the name of the object shown on the canvas.

Alternative Region Stacked Canvasses

Standard *block_region*

The region field belong to *block*. *region* describes the fields shown in the region. For example, a block LINES has two alternative regions, one showing price information (base price, discounted price, total price) and the other showing account information. The alternative region stacked canvases are named LINES_PRICE and LINES_ACCOUNT.

Example LINES_DESCRIPTION, LINES_PRICES

Query–Find Canvasses, Windows, and Blocks

Standard QF_*object*

To distinguish windows, blocks and canvasses used for Find Windows, prefix the object name with "QF_".

Windows

Standard *object*

object is the name of the object shown in the window.

LOVs

Standard *object*

object is the name of the object shown in the LOV.

Example ORDER_SALESREPS, LINE_SALESREPS, FREIGHT_CODES

LOV Record Groups

Standard *object* or *object_criteria* *object* is the name of the objects in the record group, usually the same as the basic item or LOV name. *criteria* is a brief description of why specific objects are included in the record group. Use the criteria description only if using object name alone is not unique. Abbreviate the object name or criteria description if *object_criteria* exceeds 30 characters.

Query LOVs and Related Record Groups

Standard QF_*object*

To distinguish between LOVs and record groups used for entry from those used for querying purposes (such as Find Windows), prefix the object name with "QF_". For example, QF_FREIGHT_CODES, QF_DEMAND_CLASSES.

Alternative Region LOVs and Related Record Groups

Standard *block*_REGIONS

To distinguish the LOVs and record groups invoked when pressing KEY–MENU in an alternative region, append _REGIONS to the block name.

Example LINES_REGIONS, ORDERS_REGIONS

Relations

Standard *master_detail*

master is the name of the master block in the relation, and *detail* is the name of the detail block in the relation.

Item and Event Handler Packages and Procedures

Package *block* or *form*

Item Handler Procedure *item*

block is the name of the block owning the items, *form* is the name of the form, and *item* is the name of the item on which the item handler procedure acts.

Combination Block Parameter

Standard *block*_RECORD_COUNT

block is the name of the combination block.

File Standards

All file names must be no greater than 8 chars in length plus a three character extension (FILENAME.ext). The files names must be all caps except for the extension. This will provide a consistent naming scheme across all platforms.

Form Source File Names

Standard *pppggxxx.fmb, pppggxxx.fmx*, or *pppggxxx.fmt*

ppp is the two or three character product short name, *g* is a two–character abbreviation for the functional group, and *xxx* is a three–character abbreviation for the explanation of the purpose.

fmb is the suffix for the source binary file, *fmx* is the suffix for the executable binary file, and *fmt* is the suffix for the source text file. The files reside in $prod/forms/US (or the platform equivalent).

APPSTAND Equivalents

The APPSTAND form provides many standard settings and components that other forms reference in. Particular applications and functional groups may create their own standard form that they use for reference.

The naming convention of these APPSTAND equivalents is:

Standard *PPP*STAND

PPP is the two or three character product short name, such as OE for Order Entry (OESTAND) or GL for General Ledger (GLSTAND).

PL/SQL Packages, Procedures and Source Files

Note that PL/SQL packages and procedures are documented slightly differently: Packages do not have Arguments sections and procedures do not need to have History sections.

Begin all SQL and PL/SQL files with the following lines (after the copyright header):

```
SET VERIFY OFF
WHENEVER SQLERROR EXIT FAILURE ROLLBACK;
```

End all SQL and PL/SQL files with the following lines:

```
COMMIT;
EXIT;
```

PL/SQL Stored Package Source File Names

Standard *pppgxxxs.pls* or *pppggxxb.pls*

ppp is the two or three character product short name, *g* is a one–character abbreviation for the functional group, and *xxx* is a three–character abbreviation for the explanation of the purpose. If you do not need three characters for that purpose, you may use two characters for the functional group. *s* indicate a specification file, and *b* indicates a body file. Each file defines only one package specification or body. The files reside in $prod/install/sql (or the platform equivalent).

Table Handler Package Source File Names

Standard *pppgixxs.pls* and *pppgixxb.pls*

i indicates (table) "interface."

PL/SQL Library File Names

Standard *pppggxxx.pll, pppggxxx.plx,* and *pppggxxx.pld*

The files reside in $prod/plsql and, for run–time, in $au/plsql (or the platform equivalent).

Icon File Names

Standard *ppxxxxxx.ico*, *ppxxxxxx.bmp*

pp is the two–character product short name, and *xxxxxx* is an icon name up to six characters long. The files reside in $TK2_ICON (or the platform equivalent).

Reserved Words

In addition to all words reserved by Oracle8, PL/SQL, and Oracle Forms, do not use any words that begin with the following letters (including the words themselves):

- FOLDER
- CALENDAR
- APPCORE

Glossary

applet See Java applet.

appletviewer A program residing on a client machine that runs and displays a Java applet or application.

Applications database server The machine or machines containing Oracle Applications Release 11 Server code and the Oracle Server holding Oracle Applications data. The Applications database server holds all data and data–intensive programs and processes all SQL requests from the forms –servers and concurrent processing servers. In previous releases, this was known as the Oracle Applications Server.

application server Machines that reside between the client machines and database servers, providing load balancing, business logic, and other functionality. The Oracle Web Application Server handles most of this logic. Also see forms server.

applmgr The login used to install and upgrade Oracle Applications on a server. It owns the server product files.

background process A non–interactive process that runs in an operating system environment and performs a task.

bitmap Definition of a physical bit image on a coordinate plane. A bitmap has a height, width, and vertical and horizontal resolution.

character mode An interface in which users access screen fields and regions through menus or keystrokes. Contrast GUI.

character set A set of encoded binary values that represent the letters, numerals, and punctuation marks of a language, or of a group of languages that use similar written symbols. For example, the WE8ISO8859P1 character set can be used by English and many other languages that use a Latin–based alphabet and Arabic numerals. Terminals and printers handle text data by converting these encoded values to characters. A character set may also be called a codeset.

client A general term for a computer that requires the services, data, or processing of another computer. See client / server architecture.

client/server architecture A computing arrangement in which one or several servers perform database processing for applications that are run on clients. Contrast multi–tier architecture.

command An instruction or request for the system to perform a particular action. An entire command can consist of the command name, parameters, and qualifiers.

concurrency The simultaneous access of the same data by multiple users.

concurrent manager A process manager on the Applications database server that coordinates the concurrent processes generated by users' concurrent requests. See also concurrent processing facility.

concurrent process A task run by a concurrent manager. A concurrent process runs simultaneously with interactive functions and other concurrent processes.

concurrent processing facility An Oracle Applications facility that runs time–consuming, non–interactive tasks in the background.

concurrent processing server A machine on which concurrent processing facilities are run.

concurrent queue A list of concurrent requests awaiting completion. Each concurrent manager has its own queue of pending requests.

concurrent request A request issued to the concurrent processing facility when you submit a non–interactive task, such as running a report.

customization The process of tailoring an Oracle Applications system to the needs of a specific user community.

data dictionary A set of tables and views that contains administrative information about users, data storage, and privileges. It is created and maintained automatically.

database A set of operating system files in which an Oracle Server stores a set of data dictionary tables and user tables.

database instance A running ORACLE system. There is always a one–to–one correspondence between an ORACLE instance and a system global area (SGA).

database object A logical entity created and stored in a database. Tables, views, synonyms, indexes, sequences, and stored procedures are all examples of database objects.

DBA A database administrator responsible for the maintenance of the Oracle Server and the database objects of Oracle Applications.

desktop client A machine on a user's desktop that sends requests for data and then displays the results. In Release 11 (NCA), the desktop client runs the Oracle Forms client Java applet using a Java–enabled web browser or appletviewer, which sends user requests to the forms–server and handles its responses.

environment variable A variable maintained by the UNIX shell that can be referenced by any program running within the shell. Environment variables hold values used by many Oracle programs and utilities.

extension The second part of the full file specification used to indicate the type or purpose of the file. For example, the extension ".sql" indicates a SQL script. See also filename.

filename The name component of a file specification. A filename is assigned by either the user or the system when the file is created. See also extension.

form A logical collection of fields, regions, and zones that appears on a single screen. Oracle Applications forms resemble paper forms used to run a business. You enter data by typing information into the form.

Forms Cartridge Handler An Oracle Web Application Server cartridge that parses a dynamic initial HTML file used for launching the Oracle Forms client Java applet. When a user invokes the initial HTML page from a web browser or appletviewer, the Forms Cartridge Handler reads in the HTML file and substitutes values for items that may differ among users. For example, it can choose the least–loaded forms server to run the Applications forms. The results of parsing the HTML file are then sent to the requesting user's web browser or appletviewer.

Forms Server listener A process that continuously runs on a forms server that handles requests to display Oracle Forms form files. These requests are sent from the Oracle Forms client Java applet running on a desktop client.

forms server A specific type of application server that hosts the Oracle Forms Server engine. This server processes end–user requests by sending messages directly back to the client or by making requests for data to the Applications database server. Data is, in turn, cached on the forms –server and provided to the client as needed.

GUI Graphical User Interface (Also known as a bit–mapped interface). An interface used with personal computers and workstations that allows the user to access fields and regions of the screen with a pointing device, typically a mouse. Contrast character mode.

hypertext A document format that contains links leading to other information or other documents. Also see World Wide Web.

Java applet A program, typically small in size, written in the Java programming language that is downloaded and run by a web browser or appletviewer.

LAN Local Area Network. A limited–distance, high–speed, data communications network that allows various data processing resources to be connected and shared. Contrast WAN.

LOCAL For Windows platforms, this parameter specifies the SQL*Net database alias to use when no communications driver is specified upon loading an Oracle tool.

log in (verb) To perform a sequence of actions that establishes communication with the operating system and sets up default characteristics for the session. Also called signing on.

Megabyte (MB) A unit of memory or disk space equal to 1,048,576 bytes (1024 x 1024). Often rounded to one million bytes.

multiple sets of books See set of books.

Multiple Organization Architecture (Multi–Org) A single installation of any Oracle Applications product to support any number of organizations or different sets of books. The data contained in product schemas is for all organizations, and is partitioned by the ORG_ID column in tables.

multi–tier architecture The underlying architecture of Release 11 (NCA). The architecture consists of desktop clients requesting information from application servers (including forms servers) that mediate connections to the Applications database server. Contrast client / server architecture.

operating system The computer software that performs basic tasks such as allocating memory and allowing computer components to communicate.

ORACLE An Oracle Server database. This generally refers to a database and the objects it contains, not to the Oracle Server executable files.

Oracle Applications System Administrator The person responsible for administering Oracle Applications security and tailoring system operation.

Oracle Installer The program used to install most Oracle products. Oracle Developer / 2000 uses it to install its software; Oracle Applications Release 11 does not.

Oracle Server The database management system sold by Oracle Corporation. The term refers in general to the product executable files and / or the ORACLE databases created through those files.

ORACLE_SID An environment variable that identifies an ORACLE database.

parameter An object of a command. A parameter can be a file specification, a symbol value passed to a command procedure, or a word defined by the operating system.

password An identification word associated with a username. A user must supply a password to access an ORACLE database or an Oracle Applications system.

patch driver A file read by AutoPatch that lists the actions required to apply a patch or release update. Examples of actions include copying a file, generating a form, or running a SQL script.

platform Any individual operating system. Although most Oracle Applications procedures are the same across platforms, some procedures vary. The latter procedures are called platform–specific.

PL/SQL A procedural extension of SQL that provides programming constructs such as blocks, conditionals, and procedures.

product group A set of Oracle Applications products that uses a single installation of Oracle Application Object Library tables.

prompt Words presented on the terminal screen to assist a user’s data entry.

queue A line of items waiting to be processed.

Release 10SC (SmartClient) Provides a graphical user interface to Oracle Applications through the use of Oracle Forms 4.5 run from client machines. Release 10SC is a client/server release supplement to Oracle Applications Release 10.6 or 10.7.

report An organized display of Oracle Applications information. A report can be viewed online or sent to a printer. The content of a report can range from a summary to a complete listing of values.

server A computer in a client/server or multi–tier architecture that handles requests made by client machines or other servers. In a client/server architecture, the server addresses the functions required for concurrent, shared data access to an ORACLE database. In a multi–tier architecture, the role of the server may vary: see application server, forms server, and Applications database server. In some cases, the term "server" may be used to denote a machine or a process running on a machine.

set of books An organization or group of organizations within Oracle Applications that shares a common Accounting Flexfield structure, calendar, and functional currency. You must define at least one set of books for each business location.

SGA System Global Area. Memory that provides communication between all database users and the ORACLE background processes.

short name An abbreviation for an Oracle Applications product (such as gl for Oracle General Ledger) or an Oracle Applications language (such as brapor for Brazilian Portuguese).

shut down (verb) The process of stopping a running instance to make a database unavailable, including closing and dismounting a database if one has been mounted and opened.

SmartClient See Release 10SC.

spawned process A background process initiated by a running program. These include programs run by concurrent managers and SQL*Net listeners.

SQL Structured Query Language. An internationally standard language used to access data in a relational database. The acronym is pronounced "sequel."

SQL*Loader An Oracle Server tool used to load data from operating system files into Oracle Server database tables.

SQL script A file containing SQL statements that can be run through SQL*Plus to perform queries or database administration and installation tasks.

Standard Request Submission A standard interface in Oracle Applications that lets you run and monitor concurrent requests.

stored procedure or function a procedure or function that resides and executes in the Oracle Server. Also called server–side procedure or function.

subdirectory A directory that is contained within another directory.

synonym An alias for a table, view, sequence, or program unit that masks the real name and owner of the object, provides public access to the object, and simplifies SQL statements for database users.

SYS schema One of two standard DBA usernames automatically created with each database (the other is SYSTEM). SYS owns the base data dictionary tables and views.

system administrator See Oracle Applications system administrator.

SYSTEM schema One of two standard usernames automatically created with each database (the other is SYS). The SYSTEM username is the preferred username to use when performing database maintenance.

SYSTEM.DUAL table A necessary table that contains exactly one row. It is used as a "dummy" table to guarantee a known result, such as "true."

table The basic unit of storage in a relational database management system. A table represents entities and relationships, and consists of one or more units of information (rows), each of which contains the same kinds of values (columns).

tablespace A logical portion of an Oracle Server database used to allocate storage for data and to group related logical structures. For example, one tablespace may group all of an application's database objects.

temporary tablespace A tablespace used when a SQL statement requires the creation of temporary segments (for example, the creation of an index).

tier A virtual grouping of one or more servers performing the same set of functions. In Release 11 (NCA), for example, the desktop clients make up the lowest–level tier, the forms servers comprise a middle tier, and the Applications database server and concurrent processing servers form the uppermost tier.

transaction processing option An Oracle Server option for handling a large volume of transactions with a high amount of concurrency.

URL Uniform Resource Locator. An address used to uniquely identify a document on the World Wide Web. An example of a URL is http://www.oracle.com.

user exit A program related to a form. Users invoke it to perform tasks outside the scope of the form.

username A name that grants access to an Oracle Server database schema and defines which database objects the user can manipulate. Every username is associated with a password.

view A custom–tailored presentation of the data in one or more tables. A view can be thought of as a "stored query."

virtual directory Part of a URL that indicates the location of a document on a web server. The web server translates the virtual directory, entered by the user, to a physical location on the machine's file system.

WAN Wide Area Network. A long–distance, low–speed (typically 128 Kbps or slower), data communications network that allows various data processing resources to be connected and shared. Contrast LAN.

web browser A program used to retrieve and display documents on the World Wide Web. Netscape Navigator and Microsoft Internet Explorer are the most common web browsers.

web client A machine on which a user is running a web browser or appletviewer. See also desktop client.

web listener The main component of a web server program that runs as a background process, accepting incoming requests and returning the requested data or document. For example, the Oracle Web Application Server contains the Spyglass Web Listener, but may also work with web listeners from other vendors, such as Microsoft or Netscape.

web server A program that accepts requests to retrieve and display documents on the World Wide Web. The requests are typically sent by a web browser, and may be processed by additional programs (called cartridges in the Oracle Web Application Server), before being passed to the web listener. The term "web server" may be used to denote either this program or the actual machine on which the software is running.

World Wide Web (WWW) A network of machines running web servers that provide access to hypertext documents. The network may consist of machines on the Internet, a corporate intranet, or a combination of both. Also called simply "the Web."

Index

A

B

C

D

G

H

I

J

K

L

M

N

O

P

T

U

V

W

Z

Reader's Comment Form

Oracle® Applications Developer's Guide
Part No. A58187–01

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:

Oracle Applications Documentation Manager
Oracle Corporation
500 Oracle Parkway
Redwood City, CA 94065 U.S.A.
Fax: (650) 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.