Oracle Forms Developer and Oracle Reports Developer

Guidelines for Building Applications

Release 6*i*

January, 2000

Part No. A73073-02

ORACLE

Oracle Forms Developer and Oracle Reports Developer: Guidelines for Building Applications, Release 6*i*

Part No. A73073-02

Contributing Authors: Frederick Bethke, Marcie Caccamo, Ken Chu, Frank Rovitto

Contents

2 Designing Visually Effective Applications

3 Performance Suggestions

4 Designing Multilingual Applications

5 Designing Portable Applications

6 Taking Advantage of Open Architecture

Send Us Your Comments

Forms Developer and Reports Developer: Guidelines for Building Applications

Part No. A73073-02

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 chapter, section, and page number (if available), and email them to `d2kdoc@us.oracle.com`.

Preface

The guidelines in this book are intended to help you fully exploit some of the powerful features available in both Forms Developer and Reports Developer. Whether you've been using these applications for years or are brand new to these products, the concepts and suggestions provided in this book should make it easier for you to complete such tasks as deploying an existing Form or Report application on the Web, designing an effective graphical user interface, or tracking and managing the disparate modules that make up a single application.

How does this book fit in with the accompanying online help? While you can always rely on the online help to explain **how** to accomplish a given task or which options are available to you within a product, this book helps you determine **why** you'd want to choose one option over another and to understand the consequences of each decision. Use this book to help develop your strategy; use the online help for instructions on implementing that strategy.

These guidelines represent the combined experience of our customers, marketing representatives, sales consultants, and the Oracle Applications group. You may want to use these guidelines as the basis for developing your own company standards, or use them simply to augment the standards you already have in place.

Intended Audience

This book is intended for anyone who uses either Forms Developer or Reports Developer. The needs of both novice and advanced users are addressed.

Structure

This book contains the following chapters:

Chapter	Description
Chapter 1, “Managing Your Applications”	Explains how to use the tools currently available with Forms Developer and Reports Developer to set up and manage the development of applications. Topics include: ■ Setting up and administering projects ■ Enabling team development under a variety of networking scenarios ■ Source-controlling projects ■ Exporting projects between platforms Exporting projects to different environments during the application lifecycle
Chapter 2, “Designing Visually Effective Applications”	Presents visual considerations for developing Forms Developer and Reports Developer applications using Form Builder, Report Builder, and Graphics Builder.
Chapter 3, “Performance Suggestions”	Detailed suggestions for improving the performance of the your applications.
Chapter 4, “Designing Multilingual Applications”	Explains how to design multilingual applications.
Chapter 5, “Designing Portable Applications”	Discusses how to develop an application that can be easily ported across Windows 95, Macintosh, and UNIX. Also discusses developing for character mode terminals.

Chapter	Description
Chapter 6, “Taking Advantage of Open Architecture”	Discusses how to use Forms Developer and Reports Developer to: ■ Create applications that include OLE objects and ActiveX controls. ■ Customize your applications with foreign functions. ■ Build and modify applications using the Open API. Run applications against ODBC-compliant data sources.

Notational Conventions

The following conventions are used in this book:

Convention	Meaning
boldface text	Used for emphasis. Also used for button names, labels, and other user interface elements.
italicized text	Used to introduce new terms.
`courier font`	Used for path and file names.
`COURIER CAPS`	Used for: ■ File extensions (`.PLL` or `.FMX`) ■ Environment variables ■ SQL commands ■ Built-ins / package names ■ Executable names

1

Managing Your Applications

One of the most important aspects of application development is managing the modules that make up an application. Large applications can consist of literally thousands of modules, and millions of lines of code. In addition, modules which are important to the project as a whole but which are not compiled into the application itself (such as design specifications, test scripts, and documentation) must also be tracked and maintained.

This chapter explains how to use Forms Developer and Reports Developer to help you manage the application development process.

Section	Description
Section 1.1, "The Software Development Lifecycle: An Overview"	Briefly covers the major milestones of application development and discusses Project Builder within that framework.
Section 1.2, "Managing Project Documents During Design and Development"	Discusses how to manage documents during development of an application.
Section 1.3, "Managing Project Documents During the Test Phase"	Discusses how to ensure that your QA group tests the correct configuration of project documents during the test phase.
Section 1.4, "Managing Project Documents During the Release Phase"	Discusses how to ensure that an installable version of your application is delivered to your customers.
Section 1.5, "Deploying Completed Applications"	Discusses how to turn your own application into one that is installable by the Oracle Installer.

1.1 The Software Development Lifecycle: An Overview

Application development typically occurs in four phases:

- **Design**. The initial specification for the application is developed. This specification can be based on a variety of sources: customer feedback, input of project management or development team members, requests for enhancement, necessary bug fixes, or systems analysis.
- **Develop**. Individual modules are created or modified, possibly incorporating a wide variety of languages, tools, or platforms.
- **Test**. The modules are tested. This generally occurs in two stages: *unit test* and *system test*. Unit test is testing at a modular or functional level; for example, testing UI elements such as menus or buttons. System test tests the integration of major portions of the code; the backend with the UI, for example.
- **Deploy**. The modules are packaged together in an installable form and delivered to customers.

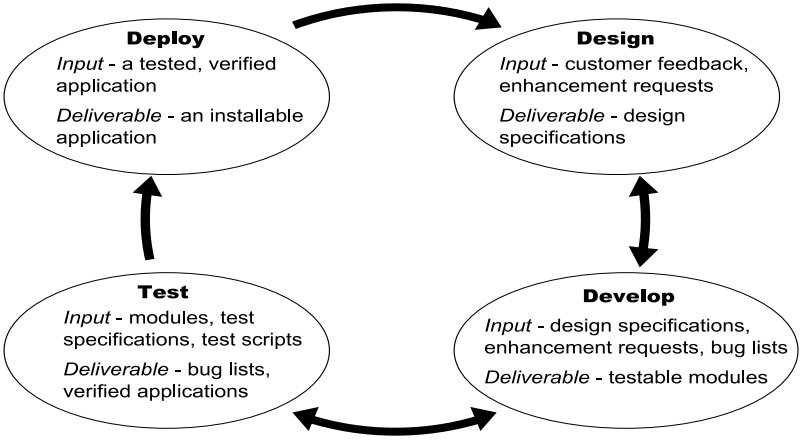


Figure 1–1 The phases of the development lifecycle: input and deliverables

As the application grows in size and complexity, the four phases are repeated iteratively, and the amount of information produced (actual code, bug reports, enhancement requests, etc.) grows. Yet all input and deliverables for all phases must be tracked and maintained to ensure the integrity of the final deliverable: the application your customer installs.

This chapter discusses how to use Forms Developer or Reports Developer to manage your application's code base and maintain version integrity.

1.1.1 Using Project Builder to implement a management strategy

In any development project, management tasks can be split roughly into two categories:

- Project management, which includes allocating the necessary equipment, budget, and person-hours of work necessary to complete the development of the application.
- Software configuration management, which includes assigning modules to developers, determining dependencies among modules, maintaining the code under development, and version control.

Project Builder, a component of both Forms Developer and Reports Developer, enables you to simplify your software configuration management tasks so you and your team can focus on your primary objectives: designing, coding, and testing applications.

1.1.2 About Project Builder

To help simplify your software management tasks, Project Builder provides the means for you to:

- Associate modules with an application or component of an application.
- Automate actions based on file types.
- Create dependencies between modules and indicate how changes cascade; in other words, show which modules need to be recompiled based on changes to other modules.
- Assign default connection strings to modules.
- Designate which modules are to be included in the final install set.
- Share projects and subprojects among team members and port them to different environments.
- Invoke other tools from the Project Builder user interface.

These features are described in detail in Section 1.1.3, "Exploring Project Builder benefits". If you're unfamiliar with Project Builder terminology, however, it's a good idea to read through Section 1.1.2.1, "Understanding Project Builder terminology"

before proceeding. This section defines some basic terms which provide the context for a discussion of Project Builder's features.

1.1.2.1 Understanding Project Builder terminology

Project Builder is based upon the concepts of *projects* and *subprojects*:

- Projects are collections of pointers to the modules and files that are part of your application.
- Subprojects are projects contained within other projects, providing a finer level of organizational granularity. Often the organization of files into subprojects mirrors the organization of files into subdirectories, but this is not a requirement.

In addition to projects and subprojects, these terms are also central to a solid understanding of Project Builder:

- **Types.** A type is the basis of every item, and controls the actions that are available in Project Builder. Project Builder types recognize their associated file types primarily by default extension; for example, `.TXT` for text files. Project Builder predefines types for many commonly used files, such as forms documents (`FMB`), text files, and C source files. You can also use the Type Wizard to define types for other applications.
- **Project items.** The components that make up a project are known as *items*. An item is simply a description of a file that is part of a project. Each item is fully described in the associated Property Palette, which lists the item's type, location in the file system, size, and when it was last modified. The *actions* and *macros* (see below) for the item are also defined.

  It is important to remember that an item is not the file itself; rather, it is a description of the file. So, when you delete an item from a project, you are simply telling Project Builder that the file is no longer part of the project. The file itself is not deleted.
- **Actions.** Actions are command strings that apply to files of a given type; for example, the Edit action for a text item may be the command string that invokes Notepad or WordPad.
- **Macros.** Macros are variables you can use to modify actions. A macro may be either a constant or a simple expression (which, in turn, may contain other constants and/or expressions). For example, Project Builder inserts all the information you've specified for connecting to a database into the `ORACONNECT` macro, which is included in all commands that might require you to connect.

The information in the macro is then inserted into the action so you can log on automatically.

Just as you might use environment variable in scripts or batch files to conveniently modify a script's actions without editing the script itself, so you can use macros to customize actions without having to edit the action themselves. For example, you might define a macro to determine whether to compile your application in Debug mode or Optimized mode. In preparation for building the deployment version of the application, you would simply change one macro definition to switch off Debug, rather than having to find and modify every type whose Build command made use of the Debug flag.

- **Global registry.** The Global Registry contains the pre-defined Project Builder types.
- **User registry**. Each user has a user registry in which to define new types, redefine existing types, and modify or create actions or macros.
- **Project registry file.** The project registry file contains information necessary to track a project, including pointers to modules contained within the project, default connection strings, and a pointer to the "home" directory for the project.

The Project Builder interface provides three tools for manipulating the items that make up a project:

- The Project Navigator furnishes a familiar "navigator" or "explorer" style interface with which you can view the modules in your application. In addition, you can use Project Builder's filtering capabilities to display only the modules you want to see. You can also launch editing tools directly from the Project Navigator.
- The Property Palette enables you to examine and modify the properties of selected items.
- The Launcher, a secondary toolbar, provides another means of accessing development tools. You can even add buttons to the Launcher and associate them with your favorite third-party tools.

Figure 1–2 depicts all three of these tools.

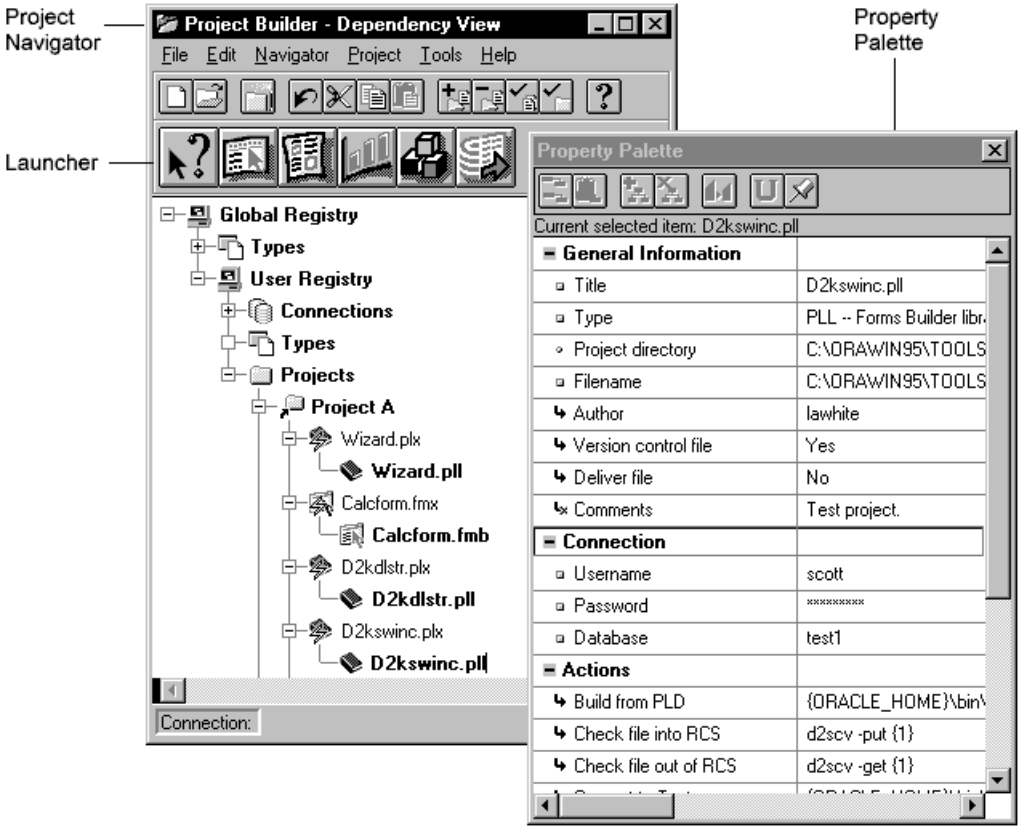


Figure 1–2 The Project Builder user interface

1.1.2.2 How Project Builder affects existing development roles

Certain roles must be filled to keep the application development effort going smoothly. Some, such as project manager, development manager, and team leader, are common roles within development groups and require no definition. However, with Project Builder one new role exists—that of *project administrator.*

A project administrator is charged with creating projects and making them available to developers. The project administrator maintains the Global Registry and modifies it as necessary, exporting the changes to the developers on the team. He or she may

also export the project information to different environments, such as test environments, or other platforms for cross-platform development.

The work the project administrator does when managing projects may affect the roles of the following team members:

- Developers
- Source control administrator
- Testers (QA)
- Releaser

Of course, the precise duties of each team member vary from development group to development group. A team member may also take on more than one role; for example, a team leader may also be a project administrator, or a developer may be in charge of source control.

1.1.3 Exploring Project Builder benefits

Now that you are familiar with basic Project Builder terminology (see Section 1.1.2.1, "Understanding Project Builder terminology"), let's examine the benefits Project Builder provides.

1.1.3.1 Associating modules with an application

You can associate all of the modules in an application with the application itself simply by adding the modules to the same project. This allows you to track a large application as a single entity, determine the dependencies between modules, and so on.

1.1.3.2 Automating actions based on file types

Project Builder ships with an extensive list of types, to which are assigned default actions (such as Open, Edit, or Print). When you select a module, then click the right mouse button, a pop-up menu displays the actions associated with that type. By default, the actions included in a type definition apply to all modules of that type in a project. You can also modify and add to these actions.

Actions are simply command strings. One benefit to defining actions with the actual command strings (besides simplicity, of course) is that an action can be associated conceptually with several different types. For example, editing a Word document requires a different tool than editing a text document, yet conceptually the two edits are very similar. Project Builder can associate an Edit command with many different

types, using a different command string for each. In this way, a single command executes an appropriate action no matter what type of module you're working with.

1.1.3.3 Creating dependencies between modules

Knowing which modules depend on which other modules is necessary to determine when modules need to be recompiled as a result of a change. It's also the key to managing the impact of changes; for example, if a library changes, which forms are now out-of-date?

Project Builder includes the dependencies for module types in their type definitions. Thus, it can recognize dependencies between existing modules in a project. Since it can also track modifications to modules, it automatically recompiles changed modules **and** the modules dependent on them.

In fact, Project Builder can recognize dependencies that do not yet exist within the project and create markers for them. These markers are called *implied items*. For example, suppose your project contains an .FMB file, defined by the Project Builder type "Form Builder document." The "Form Builder executable," or .FMX file, may not exist—you may not have generated it yet. But Project Builder knows the existence of this .FMX file is implied by the existence of the .FMB file, and creates an implied item to mark it.

To determine the existence of an implied item, Project Builder correlates the value of the property **Deliverable Type** for each defined type with the input items, or source, required for the **Build From <type>** action for each defined type. In our example above, the **Deliverable Type** property for the "Form Builder document" type is defined as "Form Builder executable," or .FMX.The **Build From <type>** action defined for a Form Builder executable is "Build From FMB". This means .FMB files are the input items for creating .FMX files, and, conversely, .FMX files are targets for .FMB source.

The chain of implied items can consist of multiple files. For example, suppose you add a C source file to a library file. In this case, Project Builder adds modules of whatever other types are necessary to get a complete path of **Build From <type>** actions from one file type to the other (like an object file).

While Project Builder detects dependencies only between compilable modules and their resultant executables, you can set dependencies manually by adding modules to a project below the item dependent on them. For example, if an .FMB is dependent on a PL/SQL library, you can add the .PLL to the project below the .FMB, and Project Builder will recognize the dependency.

1.1.3.4 Assigning default connection strings to modules

With Project Builder, you can define all of your most-used *connection strings* and store their definitions under the Connections node. You can then assign a connection to a module by dragging the connection from the Connections node and dropping it on the module. When you need to edit that module—for instance, a form—you can select the form in the Project Navigator and choose **Edit** from the pop-up menu. Project Builder automatically opens Form Builder and connects to your database for you.

1.1.3.5 Designating which modules are to be included in the final install set

Project Builder makes it easy to determine and track the modules that will be included in your final install package (for example, .EXE files, .DLL files, and .HLP files).To earmark a file for delivery, set the **Deliver File** property to Yes. When you're ready to create an install package, you can use the Delivery Wizard to package all modules for which the **Deliver File** property is set to Yes into a single unit.

Note: You can set the **Deliver File** property for a type **or** for individual project items.

1.1.3.6 Sharing and porting project and subproject registry files

Project Builder enables you to *export* the information about a project to other team members and to other platforms. Information about types, actions, macros, and project registry files—including all the customizations you've made—can be written to a text-based export file which you can then *import* to other environments and other platforms. This enables cross-platform development and testing.

1.1.3.7 Accessing other product components and third party tools

You can access other tools from the Project Builder user interface through several means:

- Actions, which you access by selecting a module in the Project Navigator and clicking the right mouse button. A pop-up menu displays all the actions associated with the selected item; the actions listed invoke whatever tools are specified in the command strings. You can also double-click an item in the Project Navigator to invoke its default action.
- The **Build**, **Deliver**, and source control actions, which launch whatever tools are associated with them.

- The Launcher toolbar, which launches many components such as Form Builder, Report Builder, and Graphics Builder. You can also add your own buttons to the Launcher toolbar and associate them with your favorite third-party tools.

1.1.3.8 Using source control packages

Both Forms Developer and Reports Developer provide interfaces to these source control packages:

- PVCS from Intersolv
- Clearcase from PureAtria
- Versions, the source control component of StarTeam, from StarBase

You can also use other source control tools by modifying the source control actions provided with Project Builder to point to them.

Since a variety of source control packages are available and can be used with Forms Developer and Reports Developer, specific instructions for source-controlling your projects are beyond the scope of this chapter.However, general guidelines will be provided where applicable.

1.2 Managing Project Documents During Design and Development

Much has been written about the importance of design in the success of an application.Deliverables during the design phase can include design documents and specifications, meeting minutes, UI prototypes, results from customer surveys (if the application is new), user tests and lists of enhancement requests (if the application is to be revised)—all documents that can be added to and tracked within a project.

This means the project administrator for the development effort should be identified early in the design process and begin creating the project immediately. (See Section 1.1.2.2, "How Project Builder affects existing development roles" for information on the role of the project administrator.) This section describes the role of the project administrator and the members of the development team in setting up Project Builder to manage a project during the design and development phase.Specifically, this section addresses:

- Installing Project Builder
- Creating a project
- Working with projects and project documents

- Managing projects and project documents across multiple platforms

Note: The steps involved in accomplishing simple tasks with Project Builder are in the Project Builder online help and are not included in this chapter.

1.2.1 Installing Project Builder

Project Builder is automatically installed to ORACLE_HOME\PJ10.Noteworthy files found in this directory are:

- Global Registry file (TYPESnn.UPD), where nn indicates the national language
- Default user registry files (PJUSERnn.UPD), where nn indicates the national language

Perhaps the most important issue to address during Project Builder installation is how you want to make these various files available to team members.
Section 1.2.1.1, "Installing the project and user registries" discusses your options.

1.2.1.1 Installing the project and user registries

Project Builder depends on native file sharing protocols for its security. This can make project files vulnerable to accidental changes, which is something to keep in mind as you decide how to configure the Global Registry and user registries.
Table 1–1 lists the available options.

Table 1–1 Registry installation options

Option	Pros	Cons	Recommendation
Install Project Builder with the Global Registry on a shared network drive and the user registries on local machines.	If your team is networked, developers can access a single copy of the Global Registry. This ensures that all versions of the Global Registry in use are up-to-date.	If all team members have write access to the Global Registry, it can be accidentally overwritten.	To prevent the Global Registry from being accidentally overwritten, install it in a directory to which only you have write access.

Table 1–1 Registry installation options

Option	Pros	Cons	Recommendation
Install Project Builder with copies of the Global Registry available to each team member, in addition to their own user registries.	You can propagate updates to the Global Registry simply by making a copy of the changed file available to your team members (if they are on the same platform).	The individual Global Registries are not safe from accidental overwrites or deletions.	Use Project Builder's Export facility to propagate changed registry files instead of providing copies. A more rigorous process may help discourage a casual attitude toward registry files.
Install Project Builder with the Global Registry and a single user registry shared among team members.		The types, actions, projects, and project modules are at risk for conflicting modifications.	Don't choose this option. But if you must, have members of your development team edit only the modules, not the project itself.

1.2.2 Creating a project

This section focuses on the creation of a single project for distribution to a team of developers. However, this may not be the best option for your group. If the application being developed is very large or if components are to be split up among team members, you may choose to create several separate, smaller projects, the contents of each determined by the responsibilities of each developer or group of developers.

If you decide to distribute a single project, note that Project Builder projects will accept pointers to modules that do not exist in the specified location. (You can determine whether a module exists by examining its information in the Property Palette; **Time created/modified** and **File size (bytes)** are blank if the module does not exist). This means you can distribute a single large project without requiring all team members to have all modules available.

Creating a project is an ongoing task that requires the participation of both the project administrator as well as each member of the development team. This section describes the responsibilities unique to each role.

1.2.2.1 Creating a project: Project Administrator

As project administrator, your role goes beyond creating a project registry file and deciding what to include in the project. Whether you use the Project Wizard

provided by Project Builder to create the project, or create a project registry file and manually edit the various properties, prior planning is highly recommended before you complete the following tasks:

1. Create the project:
   a. Set up the project's directory structure.
   b. Add modules.
   c. Establish default actions, macros, and connection strings.
   d. Set necessary dependencies manually.
2. Work with the source control administrator to set up a concurrent source control project.
   a. Define new types and edit existing ones.
   b. Customize actions and macros.
   c. Create reusable connections.
3. Make the project available to team members.

The next few topics provide recommendations for completing each of these tasks.

1.2.2.1.1 Step 1: Creating the project

The Project Wizard provides an easy-to-use interface for creating a project. You can also create a new project without the Project Wizard (using the New Project tool on the toolbar) and set project properties in the Property Palette.

At its simplest, a new project is a default project registry file primed with information about the Global Registry, but little else. Project Builder needs a bit more information before it can keep track of *your* project, as discussed in the next few topics.

Step 1a: Set up the project's directory structure

The directory structure of a project can have far-reaching consequences. For example, suppose a project contains modules that are located in a directory that's not a child of the project directory. Now suppose you create actions that search for and modify project modules. How will you find the "orphan" modules? Create alternate actions with hardcoded paths? Not portable. Search from the root? Not efficient.

Recommendations:

- Place modules in the project directory or in a directory that's a child of the project directory (a good choice when adding subprojects).
- As much as possible, organize your projects and subprojects so that they mirror your actual directory structure.

The standard methods for adding modules to a project are the Add Files to Project and Add Directory dialogs. Note that the dialogs always insert the full path unless the module you want to add is in the project directory; then a relative path name is used.

Step 1b: Add modules

Once you have planned the directory structure, you can add modules to the project.

Recommendation: Use subprojects whenever possible to help organize your project. But don't simply group together all forms or all reports. Group the modules into components; for example, you might create a subproject for all the modules in a large form, including .FMB files, .FMX files, PL/SQL libraries, menus, bitmaps, icons, etc. This enables you to more easily create some necessary dependencies not detected by Project Builder.

Step 1c: Establish default actions, macros, and connection strings

This step involves making site-specific edits to actions and macros; for example, changing build actions to use the compilers and compiler options that are standard at your site. If you have not already done so, you can also create connection strings for commonly used databases containing test data or necessary tables.

Step 1d: Set necessary dependencies manually

Project Builder can recognize some dependencies between modules (it knows that .FMX files are built from .FMB files, which are built from .FMT files), but only the dependencies it can deduce by cross-referencing the **Deliverable Type** and the **Build From <type>** actions.

Other dependencies may exist as well: dependencies on PL/SQL libraries, menus, icons, and so on. You can tell Project Builder about these dependencies by creating entries for the modules on which a module is dependent **below** the item for the dependent module, as shown in Figure 1–3, "Manually added dependencies".

Figure 1–3 Manually added dependencies

This figure illustrates NAVWIZ.FMB's dependency upon WIZARD.PLL, NAVIGATE.PLL, and NAVWIZ.MMB.

1.2.2.1.2 Step 2: Work with the source control administrator

After you create your project, you're ready to introduce a source control package. Many third-party source control packages also implement the concept of projects.

Recommendation: Work with your source control administrator to set up a source control project that mirrors your development project in Project Builder.

When setting up a project to source control a complex application, remember to include the non-obvious modules as well. For example, when checking in a form, don't forget menus, PL/SQL libraries, user exits, icons, or special fonts you use. Applications running on Windows may use OCX or ActiveX controls that should be source-controlled as well.

1.2.2.1.3 Step 3: Make the project available to team members

Once you've done the preliminary work of creating the project and establishing source control, it's a good idea to export all project information to a project export file and notify team members of its location. They can then import the project.

It is possible to notify team members of the location of the actual project registry file, but remember that Project Builder uses your operating system's own security features to protect your project modules from being deleted or overwritten. Simple deletes and overwrites are possible. To maintain the integrity of your projects, follow Project Builder's own process for updating projects, and always import and export modifications to the project instead of simply distributing changed registry files.

When you notify your team members of the location of the project export file, you should also notify them of the directory structure you've set up so they can mirror that structure on their development machines. The easiest course for setting up the project is to have all team members map the project location to the same project directory on their machines, since mappings to different project locations would require separate copies of the project registry file with different values for the Project Location: `Q:\myproj, R:\`, etc.

Team members can then check out the modules they have been assigned.

1.2.2.2 Creating a project: Team members

After the project administrator has completed the tasks described in Section 1.2.2.1, "Creating a project: Project Administrator", project team members can fine-tune the work. If you are a project team member, you can expect to:

1. Set up your directory structure and import the project
2. Customize your user registry
    a. Define new types and edit existing ones
    b. Customize actions and macros
    c. Create re-usable connections
3. Check out your assigned modules

1.2.2.2.1 Step 1: Set up your directory structure and import the project

When your project administrator informs you that the project is available, it's time to import the project information and set up your working directories with the modules you've been assigned.

Recommendation: File management is easier if you set up your directory structure to mirror what your project administrator has already created for the project.

1.2.2.2.2 Step 2: Customize your user registry

One of the first things to do when setting up a project is to customize your user registry.

Step 2a: Define new types and edit existing ones

If you want to add modules to your project that are of a type not represented in the Global Registry, you can use the Type Wizard to define your own type in your user registry and assign actions, macros, and so on to it.

In addition, you may want to override a default command or macro for a particular type in the Global Registry. An easy way to accomplish this is to copy the type in the Global Registry, paste it into your user registry, and edit it. Now, all modules of that type in your project will inherit the modifications from the type in the user registry.

Recommendation: Notify your project administrator when you modify a global type by copying it into your user registry and editing it. Such a modification might be useful to the whole team.

Step 2b: Customize actions and macros

While you can customize the actions and macros associated with the types you add to your user registry, it's important to remember that you can modify actions and macros at other points in the Project Builder hierarchy as well. Where you edit the item depends on the extent of the influence you want your change to have.

The following table lists all the locations you might find an action or macro, the scope of that action or macro, and what can override it.

An action or macro assigned to:	Affects:	Unless overridden by:
Global Registry	All items of type(s) to which it is assigned in all user registries and projects beneath the Global Registry.	Actions or macros in a user registry, project, subproject, or item.
User registry	All items of type(s) to which it is assigned in all projects beneath the user registry.	Actions or macros in a project, subproject, or item.
A project	All items of type(s) to which it is assigned in the project.	Actions or macros in a subproject or item.

An action or macro assigned to:	Affects:	Unless overridden by:
A subproject	All items of type(s) to which it is assigned in the subproject.	Actions or macros in an item.
An item	Itself only.	Cannot be overridden.

Step 2c: Create reusable connections

If you have your own set of tables with data you've created for testing purposes, you can add your own connections to the list provided by the project administrator. Once you've created the connections, you can assign a connection to a module by selecting the connection's item in the Project Navigator, dragging it to the project file entries, and dropping it on the item for the module you've chosen. Now, when you select an action that opens a tool requiring a database connection, Project Builder logs on for you.

1.2.2.2.3 Step 3: Check out your assigned modules

Once you have your directory structure in place and the project imported, you can populate your workspace with the modules you've been assigned. The source control commands **Check In**, **Check Out**, and **Source Control Options**, accessible from the **File→Administration** menu, are associated with actions defined for each type. This means you can modify the actions, if necessary, to affect the results of the commands—though this is not recommended for source control.

1.2.3 Working with projects and project documents

When the project enters the development phase, maintaining the integrity of the project becomes increasingly important.

Recommendation: Only the project administrator should make changes to the project that affects multiple team members (such as modifying the Global Registry or adding new subprojects containing new modules).

1.2.3.1 Working with projects: Project Administrator

While the application is in development, as project administrator your role is to maintain and support the project. In addition, you might be in charge of managing development deliverables, or working with a development manager to do so. You might need to:

- Add new modules and dependencies
- Export modifications to the project registry file
- Apply version labels

1.2.3.1.1 Adding new modules and dependencies

Sometimes new modules must be added to a project after its initial creation, and dependencies added manually. The process for doing so is the same as when creating the initial project. For more information, see Section 1.2.2.1.1, "Step 1: Creating the project".

1.2.3.1.2 Exporting modifications to the project registry file

Once you've added the new modules and made the necessary changes, you can export the changes and make them available to members of your team. The process for doing so is the same as when exporting the initial project. For more information, see Section 1.2.2.1.1, "Step 1: Creating the project".

1.2.3.1.3 Applying version labels

Although you can try to keep various revisions synchronized with each other (for example, through a nightly check-in), often development on one module will be completed while another needs bugs fixed or headers changed. Synchronous revisions are generally impractical.

A better method is to synchronize *versions* by applying a symbolic version label to the group of revisions that mark the achievement of a significant milestone. Most major source control tools enable you to apply a symbolic label to a source control project.

1.2.3.2 Working with project documents: Team members

When your project is set up and your modules have been assigned, you can use Project Builder to:

- Edit modules
- Add modules and dependencies manually
- Build your project
- Check modules in and out

1.2.3.2.1 Editing modules

Recommendation: The most efficient way to use Project Builder to edit modules is to customize the actions associated with the types of modules you'll be editing so they invoke the tools you want to use with the options you need. In addition, be sure to associate a connection string with either the individual module or the project. Then you can drag the connection from its location in your user registry and drop it on the module or project item. Once your modules are prepared in this fashion, choosing a pop-up menu item or double-clicking on a project item opens your module in the correct application. If necessary, you'll already be logged on.

You can also use the Launcher to access development tools. The Launcher is shipped with toolbar buttons already set for the Forms Developer or Reports Developer tools, but you can add a third-party tool by creating a button and associating it with an executable.

Note: If you invoke a tool via the Launcher and then open a module, the tool will not be aware of any associated connection strings. You will need to log on to the database manually.

1.2.3.2.2 Adding modules and dependencies manually

See Section 1.2.2.1.1, "Step 1: Creating the project", or contact your project administrator.

1.2.3.2.3 Building your project

The **Build** commands—**Build Selection**, **Build Incremental**, and **Build All**—are available from the **Project** menu. They are also associated with an action—in this case, the **Build From <type>** action.

This means you can select a single command for any different module type and the module will be compiled according to the definition of the **Build From <type>** action—not for that particular type, but for the target you actually want to build.

For example, the **Build From <type>** action for an .FMX file invokes the Form Generator and creates the .FMX file from the corresponding .FMB.What the **Build** command compiles is the .FMB, but how it compiles the .FMB is determined by the action associated with the .FMX that results.

You can modify the results of the **Build** commands by modifying the definition of the **Build From <type>** action for the corresponding target.

Choose **Build Selection** to compile selected modules, or force a compile of all compilable modules by choosing **Build All**. Because Project Builder can detect when modules are out-of-date and need to be recompiled, you can compile only out-of-date modules by selecting the item for the project containing them, then choosing **Build Incremental**.

Note: The **Build** commands are also available from the pop-up menu.

1.2.3.2.4 Checking modules in and out

If modules need conversion for source control (for instance, the source control only works on text and your modules are binary), you can edit the **Check file into RCS** action to automate the conversion to text before check-in.

You can also edit the **Check file out of RCS** action in a similar fashion to convert the text-based source controlled version of the module back to binary.

1.2.4 Managing projects and project documents across multiple platforms

Many applications today run on multiple platforms, with development taking place on a variety of platforms as well. Chapter 5, "Designing Portable Applications" can help you ensure that the application underlying your project is portable.

To ensure that your project is portable, too, Project Builder supports development on several major platforms. To do so, it must ship with a Global Registry that reflects the platform; in other words, the types defined must be found on that platform, and the actions and macros must be written according to the syntax rules of that platform. This means the Global Registry, and, by extension, all user registries and project registry files, are **not** portable.

However, you can export information about a project to a text file and import the text file to another platform, as discussed in Section 1.1.3.6, "Sharing and porting project and subproject registry files".

1.2.4.1 Managing projects across multiple platforms: Project Administrator

If you are the administrator of a project undergoing development on multiple platforms, you can expect to:

- Branch off a source control project to contain the code for the platform
- Export projects and project information to alternate platforms

1.2.4.1.1 Branching off a source control project to contain the code for the platform

Work with your source control administrator to create a branching source control project that enables your team members to isolate the code for the new platform.

1.2.4.1.2 Exporting projects and project information to alternate platforms

Creating an export file for the purpose of distributing a project to another platform is no different from creating an export file to distribute to team members on the same platform. The export file created by Project Builder is a text file, easily transferred to the alternate platform.

1.2.4.2 Managing project documents across multiple platforms: Team members

The role of a team member working on development on an alternate or secondary platform is actually quite similar to the role of a team member developing on the base platform. However, there is one major difference: when you receive a project already created on a different platform, you can expect to:

- Revise customized actions and macros to conform to platform requirements

1.2.4.2.1 Revising customized actions and macros to conform to platform requirements

Equivalent versions of pre-defined actions and macros, where they exist, are provided by Project Builder for all supported platforms. However, if some actions have been customized or new actions created, you will either need to edit the actions to make them work on the new platform or create equivalent new actions.

1.3 Managing Project Documents During the Test Phase

Though the test phase is often thought of as separate and distinct from the development effort—first you develop, **then** you test—testing is a concurrent process that provides valuable information for the development team.

There are at least three options for integrating Project Builder into the test phase:

- Your testers do not install Project Builder. You use Project Builder functionality to compile and source-control the modules to be tested and hand them off to the testers, whose process remains unchanged.
- The testers import the same project or projects that the developers use.
- You create a project based on the development project but customized for the testers (for example, it does not include support documents, specs, or source), who import it.

Recommendation: A combination of the second and third options works best. Associating your application with a project can be useful during the testing phase, as well. You can create actions to automatically run test scripts or add script types and make them dependent on the modules they are to test.

During unit test, testers can use the same project or projects as the developers, if the project is organized by functional units, or separate projects have been created for functional units.The project or projects can also be exported, so unit test can take place in a variety of environments and on a variety of platforms.

System test might require a new, stripped-down version of the development projects that includes only the modules being tested, especially if you need to concatenate several smaller projects.

1.3.1 On the development side

The goal of the development group in this phase of the process is to provide the test group with the modules to be tested in as smooth a manner as possible.

1.3.1.1 The test phase: Project Administrator

The tasks involved in creating and exporting a project for testing purposes are the same as the tasks required when creating and exporting a project to a development team:

- Create a test project based on deliverable modules (optional)
- Create the test version
- Export the project to different test environments

1.3.2 On the test side

Although members of the test team generally are not responsible for any modifications to the modules of an application, they do have input (modules to test) and deliverables (fully-tested modules and lists of bugs uncovered during the testing phase).

Project Builder can help the test team keep track of its input and deliverables in the same way it helps development team members.Testers can add scripts and logs to a project, modify actions to include debugging options, and add subprojects containing testing information.

1.3.2.1 The test phase: Testers

If you have decided to use Project Builder to help test your application, you'll need to do some preparatory work that is very similar to that of the developers when they are first setting up their projects.You may need to:

- Import the test project and set up the testing environment
- Add test scripts and test data to the project
- Modify actions and macros to facilitate testing

1.3.2.1.1 Importing the test project and setting up the testing environment

The process of importing a test project and setting up a testing environment is the same as the process for importing a project and setting up the environment for development.See Section 1.2.2, "Creating a project", for more information.

1.3.2.1.2 Adding test scripts and test data to the project

You may need to add some items, such as test scripts, to the project.In addition, you may need to add connection strings to database accounts containing test data.

Remember that you can automate the running of test scripts just as you can automate actions associated with the modules in your application.

1.3.2.1.3 Modifying actions and macros to facilitate testing

If actions specifying "run with debugging" have not already been provided, you can either modify existing actions to include a debug flag, or create new actions.

1.4 Managing Project Documents During the Release Phase

When your application has been thoroughly tested and is ready to release, Project Builder can help you simplify the process of delivering the application to customers.

1.4.1 On the development side

During the release phase, the development group passes the tested and verified versions of all modules necessary for installation to the releaser.Because Project Builder marks all modules to be included in the final application and associates special commands with them, this hand-off can be automated in the same fashion as other processes, such as compiling your project and source controlling it.

1.4.1.1 The release phase: Project Administrator

Once your project has been thoroughly tested and is ready for release, you have one remaining task: package the project.

1.4.1.1.1 Packaging the project

Project Builder provides the Delivery Wizard to help you package your applications as installable components, as well as to:

- Copy or FTP your completed project to a staging area.From the staging area, you can copy or archive your files to a distribution medium, or make them available internally.
- Generate the necessary scripts so your project is installable on Windows 95 and NT through the Oracle Installer.You can even package the Forms Developer or Reports Developer runtime environments with your project, so your users can install the entire package—your application, plus the required runtime environment—from a single invocation of the Oracle Installer.
- Run a customized Deliver action to TAR or ZIP your files.

The modules actually packaged by the Delivery Wizard are determined by the value of the **Deliver file** property associated with each item (Yes to include the module in the package, No to leave it out).

1.5 Deploying Completed Applications

After you have packaged your application, you're ready to make it available to your customers.In addition to installing your application, your customers will also need

to use the Oracle Installer to install the Runtime environment on which your application depends.To simplify the installation process for your customers, both Forms Developer and Reports Developer provide the Oracle File Packager, with which you can make your own application installable with the Oracle Installer on Windows NT and Windows 95.When you've completed the steps in this section, your customers can install everything they need—your application, plus the required Runtime environment(s)—using a single mechanism.

1.5.1 Before You Begin

Before discussing how to package your application, it's a good idea to familiarize yourself with the terminology and background information relevant to the installation / packaging process:

- Section 1.5.1.1, "Terminology"
- Section 1.5.1.2, "The Oracle Installer files"
- Section 1.5.1.3, "The contents of the TEMPLATES directory"

1.5.1.1 Terminology

This table defines some important terms for the installation / packaging process:

Term	Definition
Stage (or staging) area	The area on your PC or network where files and installation scripts are prepared, then copied to the distribution media.
Distribution media	The set of media (for example, tapes, CDs, or diskettes) from which users install your application.
Installable component	Any product (for example, Forms Runtime, GUI Common Files, and so on) that has its own set of Oracle Installer files (MAP, VRF, INS, and DEI).
Product file (PRD file)	A file that lists all installable components in a given staging area.
Oracle File Packager	A wizard that creates the product file and all the Oracle Installer files (MAP, VRF, INS, DEI) needed to make your Windows application installable through the Oracle Installer.

1.5.1.2 The Oracle Installer files

The Oracle Installer files control how and where an application is installed (and de-installed) on a user's machine.While the Oracle File Packager creates the Oracle Installer files for you, you may have to make some slight modifications manually. If you just want to look at some sample installer files, take a look at:

\TEMPLATES\RELEASE\YOURAPP

\FORMSAPP

FORMSAPP.MAP
FORMSAPP.VRF
FORMSAPP.INS
FORMSAPP.DEI

\DEV2KAPP

DEV2KAPP.MAP
DEV2KAPP.VRF
DEV2KAPP.INS
DEV2KAPP.DEI

All of these files are text files and should be viewable and editable in a text editor.

1.5.1.2.1 The PRD file

The PRD file lists all the installable components in a given staging area. It also identifies the base filename and location for the Oracle Installer files of each component.In other words, the PRD lists all the files that appear in the Available Products pane of the Oracle Installer. Its name reflects the platform it describes; e.g., WIN95.PRD and NT.PRD.There is one PRD file per staging area, per platform.

Column Name	Description
####	Product number.You shouldn't have to modify this.
Product	A unique name used to identify your application.
Parent	Leave as "root".
Filename	Base filename of your MAP, VRF, INS, and DEI installation scripts.
Version	Version number of your application.
Interface Label	Name of your application as it appears in the Available Products window of the Oracle Installer.

Column Name	Description
Location	Relative path to the directory that contains the installation script files (MAP, INS, VRF, and DEI) and all the files that make up your application.
Size	Total size of the installable component.Set automatically by the CHECKMAP utility.
Visible?	Makes the component visible (or not) in the Available Products window of the Oracle Installer.
Selected?	Makes the component selected (or not) in the Available Products window of the Oracle Installer.
Open?	Used for parent/child components.You should not need to modify this field.
Description	Describes your application.
Volume	Should match what appears in the Filename field.Not used for CD or LAN installations.

1.5.1.2.2 The MAP file

The MAP file is a table that lists all the files that make up your application.

Column Name	Description
Source	File to be copied to the user's machine.
Destination	Directory to which the file is copied.
Group	Program group that will hold the program item(s).
Item	Name of the item or icon as it appears in the menu.
Command	Command that is executed when the item or icon is invoked.Appears in the format: command line working_directory alternate_icon Working_directory and alternate_icon are optional, however, if "command line" appears alone, it must end with a semicolon.

Note: Group, Item, and Command are required only for applications that appear in the Start menu.To see an example of how these fields are filled in, use your OS search capabilities to locate `DEVDEM60.MAP`, the map file for the Forms and Reports Developer demos. (If you can't find it, you may have to install "Forms Developer [or Reports Developer] Demos and Add-ons" from your Oracle distribution media.)

1.5.1.2.3 The VRF file

The VRF file VeRiFies that all the correct dependencies are identified and installed. For example, by specifying that your application depends on Forms Runtime, your application's installation process will automatically detect whether Forms Runtime is already present on a user's machine.If it is not, Forms Runtime will be installed.

The VRF file also prompts the user for information, such as where the product should be installed.In addition, the VRF file sets up the user's environment, defining such things as environment variables in the Windows registry.

1.5.1.2.4 The INS file

The INS file INStalls the files that make up an installable component, sets any needed environment variables, and registers the product with the Oracle Installer.It works in coordination with the MAP file and the VRF file.

1.5.1.2.5 The DEI file

The DEI file DEInstalls the files that make up an installable component.It also removes environment variables and unregisters the component after successful deinstallation.It works in coordination with the MAP file.

1.5.1.3 The contents of the TEMPLATES directory

The `TEMPLATES` directory provides everything you need to set up and customize your own staging area. Available on your Oracle distribution media, the `TEMPLATES` directory contains:

- The `RELEASE` subdirectory, which serves as a starting point for creating your own staging area.
- `RELEASE\INSTALLR\INSTALL\WIN95.PRD`, a PRD file that lists the installable components for Forms, Reports, and Graphics Runtime environments on Windows 95:
  - Required Support Files
  - System Support Files
  - GUI Common Files
  - Tools Utilities
  - Forms Runtime
  - Reports Runtime

  - Graphics Runtime
- RELEASE\INSTALLR\INSTALL\NT.PRD, a PRD file that lists the installable components for Forms, Reports, and Graphics Runtime environments on Windows NT (see the previous bullet for a components list).

1.5.2 Making your application an installable product

This section contains instructions for creating a one-step or a multi-step installation process for your customers:

- One-step process: Your customers install your application and the Runtime environment they need from a single PRD file.Another way to think of this is that your customers install everything they need—your application, plus the required Runtime environment(s)—from a single invocation of the Oracle Installer.
- Multi-step process: Your customers install applications from many different staging areas, each of which has its own PRD file. This approach works well if you need to distribute many Forms Developer or Reports Developer applications, or if the required Runtime environment is already available to your customers from a common area.

Whichever process you choose, to make your application installable with the Oracle Installer, you will:

- Copy the TEMPLATES\RELEASE directory from the Oracle distribution media to your machine to serve as a starting point for your own staging area.
- Use the Oracle File Packager to create the PRD, MAP, VRF, INS, and DEI files you need to make your application installable through the Oracle Installer.
- Copy your files from your development area to the staging area.From there you can copy the files to your distribution media.

The rest of this chapter contains specific instructions for completing these tasks.

1.5.2.1 Deploying your application on Windows

If your application is installable on Windows 95 and NT, you can use the Oracle File Packager to create the Oracle Installer files and to copy your files from your development area to the staging area. The following steps address both one-step and multi-step installations.

Step 1: Install the Oracle File Packager

1. From TEMPLATES\OISFP10 (on your Oracle distribution media), click SETUP.EXE to invoke the Oracle Installer.SETUP.EXE detects which operating system is running and launches the appropriate Oracle Installer.
2. Select Oracle File Packager from the list of installable products.
3. Complete the installation process as prompted.

Step 2: Prepare your staging area

1. Copy TEMPLATES\RELEASE to a drive on your PC or a networked drive.
2. Create a subdirectory for your application under TEMPLATES\RELEASE\FORWIN95, even if your application is targeted for the NT environment.

   If you are staging more than one application, create a subdirectory for each.

Step 3: Move your files to the staging area and create the Oracle Installer files.

Repeat this step for each staging area you established in Step 2.

1. From the Start menu, select Oracle for NT or Windows 95, then select Oracle File Packager.
2. Follow the steps presented in the Oracle File Packager, using the online help to assist you.

Notes:

- The internal string you specify in Step 3 is prepended to your Oracle Installer files (MAP, INS, VRF, and DEI).
- When prompted for the Staging Area Location, specify the subdirectory under TEMPLATES\RELEASE\FORWIN95.

Step 4: Merge your PRD file with NT.PRD and WIN95.PRD

This step creates a one-step installation process.If you're creating a multi-step installation, go to Step 5.

1. Copy the line from your own application's PRD file and paste it into RELEASE\INSTALLR\INSTALL\WIN95.PRD and/or RELEASE\INSTALLR\INSTALL\NT.PRD.

Step 5: Modify the Oracle Installer files

1. If you want your application to appear as an icon in the Start menu, add the Group, Item, and Command fields to the MAP file(s) for your

application(s). To see an example of how to fill in these fields, use your operating system's search capabilities to find the Oracle Demos MAP file, DEVDEM60.MAP.

2. If you wish to establish some dependencies for your application, add them to the VRF file.

   For example, if you establish Forms Runtime as a dependency for your application, the installation process will automatically detect whether Forms Runtime is already present on a user's machine. If it is not, Forms Runtime will be installed.

3. In each staging area, click SETUP.EXE to bring up the Oracle Installer. Examine the files listed in the Available Products pane. If you do not want a file to appear in this pane—for example, a file has already been established as a dependency in the VRF file and does not need to be installed explicitly—edit the staging area's PRD file and change the file's "Visible?" value to false.

Step 6: Test your installation

Test your installation on a "clean" machine (a machine with no previously-installed products) that is representative of the projected end-user environment. **Do not** rely on tests conducted on a developer's machine—that machine may already have files such as icons or libraries that you inadvertently omitted from your map file, or registry settings that were not included in your INS file. This is one of the most common causes of installation problems.

1. Install your application and make sure it installs the components it should.
2. Launch the application to make sure it runs correctly.
3. Test removing your application using the Oracle Installer.

Step 7: Copy the staging area to your distribution media

When you are ready to copy your application to CD, tape, diskette, or another medium—or simply to a LAN or other networked machine—be sure you include the entire staging area—that is, TEMPLATES\RELEASE in its entirety. If you include only your subdirectory, the required runtime environment(s) will not be accessible.

2

Designing Visually Effective Applications

This chapter offers guidelines to help you develop a graphical user interface (GUI):

Section	Description
Section 2.1, "Understanding the Process"	Briefly describes the process for developing a GUI-based application. (If you've developed GUIs in the past, you may want to go directly to Section 2.1.3, "Planning the user interface".
Section 2.2, "Creating an Effective Form"	Addresses visual considerations for creating forms. The section begins with a review of basic Form Builder terminology, discusses the importance of object libraries as a means of enforcing standards, and presents guidelines for employing forms objects in your GUI.
Section 2.3, "Creating an Effective Report"	Helps you understand how to control the placement of report objects and where page breaks occur.
Section 2.4, "Creating an Effective Display"	Presents some visual considerations for graphical representations of your data.

2.1 Understanding the Process

Even more important than understanding the process for developing an effective GUI is understanding the people who will use it. In fact, your success is directly related to how well you understand your users—the tasks they perform, the order in which they perform them, their surroundings, and their expectations.

If you're like many application developers, this idea may require a profound shift of focus. Applications typically evolve from the inside out: from the datasource itself, to the code, and finally to the GUI. If you are committed to developing an effective GUI, you must reverse this process: first, interview your users; next, design a user

interface that supports their specific tasks; and finally, create the underlying code base that makes it all work.

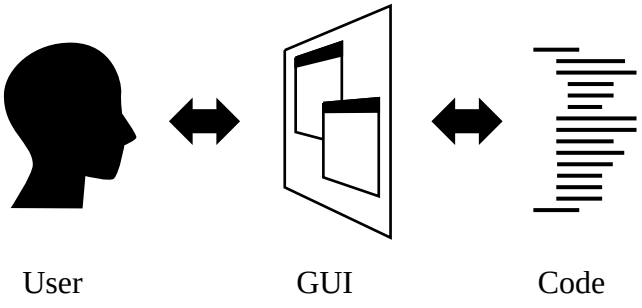


Figure 2–1 Thinking about the user first

No set of prepackaged standards or guidelines can serve as a substitute for developing an accurate understanding of your users' needs. This chapter can help you develop that understanding, as well as assist you in creating an interface uniquely tailored to your particular group of users.

2.1.1 What are the stages?

As shown in the Figure 2–2, the process for developing a GUI consists of four major stages:

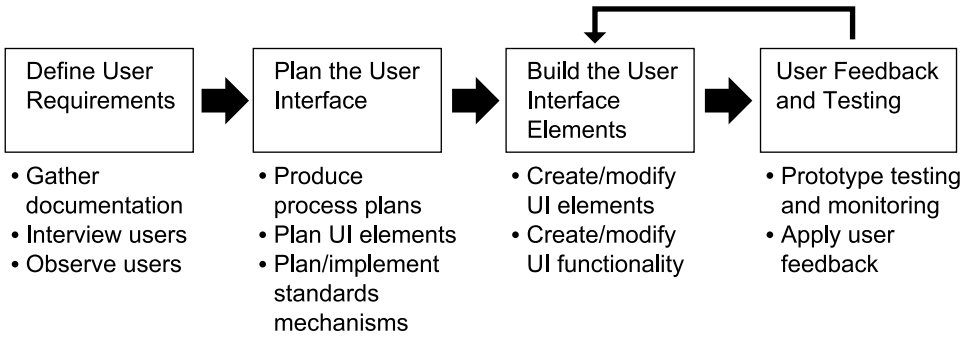


Figure 2–2 Stages in developing a user interface

The rest of this section offers guidelines for completing each of these stages:

- Section 2.1.2, "Defining user requirements"
- Section 2.1.3, "Planning the user interface"
- Section 2.1.4, "Building the user interface elements"
- Section 2.1.5, "Gathering user feedback"

Note: This chapter is not intended to treat the subject of GUI development exhaustively. If you require more detail on how to proceed in a given stage, you may want to visit your local library or computer bookstore. In particular, Jeffrey Rubin's "Handbook of Usability Testing" is an excellent source of information on defining user requirements and gathering user feedback.

2.1.2 Defining user requirements

In the first stage of GUI development, you determine what the user needs and expects from your application. While it may be tempting to skip this stage and move right to the design phase, it's risky to do so. Without a clear understanding of the users themselves and the tasks they must perform, it is virtually impossible to create an effective GUI.

To define user requirements:

- **Gather documentation.** Relevant policies and procedures manuals and existing documentation about the system (whether previously computerized or not) will help you formulate the necessary background for conducting user interviews.
- **Observe users doing their jobs.** Make a list of the tasks users perform and the order in which they perform them.
- **Interview users**. Find out what people want from a GUI-based system. When conducting your interviews:
  - Ask not only what users do, but how they work. For example, does a clerk need to be able to work on several orders at the same time, or just on one?
  - Find out what users like and don't like about the current system (even if it's not computerized).
  - Ask users how they envision the GUI. Encourage them to provide as much detail as possible.
  - Get to know the users. Do users typically stay on the job for a long time or is there high turnover? Will they use the application constantly or only occasionally? For infrequently used applications, you'll want to provide a lot of buttons, text, and guidelines to help reduce the amount of familiarization time. For applications that are used daily, try to provide a lot of shortcuts and accelerator keys to help experienced users complete their tasks quickly.
  - Find out if users have any disabilities or special circumstances you should consider. For example, are users typically standing when they use the application? If so, they won't have the time or patience for excessive navigation.
- **Sample a wide variety of users.** Feedback from users at a single customer site are biased toward their specific experiences.

2.1.3 Planning the user interface

In the second stage, you plan and document how you will implement a user interface that meets the users' needs. This involves:

- Developing a set of standards that you will adhere to and, if necessary, obtaining buy-in from your team. Refer to Section 2.1.3.1, "Creating your standards".

- Considering platform-specific requirements and other restrictions in the deployment environment. Refer to Section 2.1.3.2, "Considering portability".
- Mapping out each screen and deciding which types of interface elements to use in order to meet user needs effectively. Refer to Section 2.1.3.3, "Creating a prototype".

2.1.3.1 Creating your standards

A set of consistent development standards is crucial to the success of any development effort. By developing and enforcing standards pertaining to layout, use, and behavior of various GUI elements, you can ensure that even disparate parts of the application have a common look and feel. Both Forms Developer and Reports Developer offer several mechanisms to assist you in developing a consistent set of standards.

Table 2–1 Standards mechanisms

Mechanism	Description
Object Library (Form Builder)	An object library is a set of objects and standards that you create and make available to your entire development team. Through the use of subclassing, each developer can ensure that changes made to the objects in the object library are propagated throughout all applications that use them. **Object libraries are the preferred mechanism for standardizing your Form Builder applications.** Form Builder provides two object libraries which you can customize to meet your own site requirements: ■ Standard Object Library, which contains suggested standards optimized for the Windows 95 environment. ■ Oracle Applications Object Library, which contains standards for cross-platform applications: Windows 95, Solaris, Macintosh, and character mode. For more information, see the Form Builder online help topics "About object libraries" and "About subclassing".

Table 2–1 Standards mechanisms

Mechanism	Description
Object Group (Form Builder)	An object group is a container for a group of objects. You define an object group when you want to package related objects so you can copy or subclass them in another module. For example, suppose you build an appointment scheduler using several types of objects, including a window and canvas, blocks, items that display dates and appointments, and triggers that contain the logic for scheduling and other functionality. By packaging these objects into an object group, you can copy all of them to other forms in one simple operation. For more information, see the Form Builder online help topic "Guidelines for using object groups".
Visual attributes (Form Builder)	Visual attributes are the font, color, and pattern properties you set for form and menu objects that appear in your application's GUI. Visual attributes can include the following properties: ■ Font properties: Font Name, Font Size, Font Style, Font Width, Font Weight ■ Color and pattern properties: Foreground Color, Background Color, Fill Pattern, Charmode Logical Attribute, White on Black For more information, see the Form Builder online help topic "Guidelines for using visual attributes".
Template (Form Builder, Report Builder)	In Form Builder, you can create templates to provide other team members with a default starting point for new forms. Templates typically include generic objects, such as graphics (like corporate logos), toolbars, program units, standard window layouts, toolbars, and menus, and other common objects. Report Builder not only allows you to create your own templates to help control the appearance of your reports, but provides a wide variety of pre-defined templates as well. Using the Report Wizard, you select the objects you want to include in your report, then select a template to arrange those objects and apply standard formatting attributes. For more information, search the Form Builder or the Report Builder online help index for "templates".

2.1.3.2 Considering portability

If you intend to deploy your application in more than one environment, it's important to understand how various GUI elements are rendered on each platform

and which elements are restricted altogether. For example, due to formatting constraints between platforms, interactive buttons that you create for Windows may shrink and become less readable when displayed on Solaris. Chapter 5, "Designing Portable Applications", helps you understand platform-specific constraints and provides tips and guidelines for working around them. It also provides considerations for character mode, which restricts the UI in numerous ways.

2.1.3.3 Creating a prototype

Prototypes are an extremely effective means for ensuring usability in your application. The most effective prototypes follow an iterative development model, beginning with a storyboard and ending with a fully functional application. The process breaks down as follows:

1. Draft a *storyboard* to give you a clear picture of how the application will actually look and behave. A storyboard is a frame-by-frame drawing of screens showing transition and appearance. Include a narrative to describe how the screens relate to the tasks you identified when you defined the users' requirements.

Here is an example of three panels from a storyboard for an ordering application:

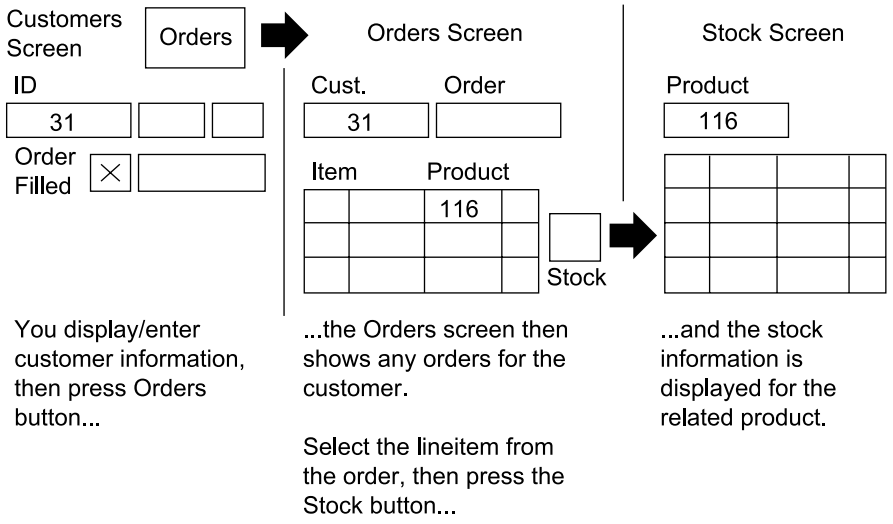


Figure 2–3 Example of a storyboard

2. Show the storyboard to users. Verify that your planned application addresses their needs and supports their tasks **the way they perform them**.
3. Expand the storyboard into a *paper prototype*. Whereas a storyboard sketches task and window flow at a high level, a paper prototype is a fairly detailed illustration of the entire application. A paper prototype typically contains one piece of paper for each window you've planned, complete with widgets, arrows to represent task flow and navigation, and so on.
4. Show the paper prototype to users. Most of the organizational issues should have been identified during the storyboard phase, so you can now focus on details: the placement of buttons, the layout of a supporting dialog, and so on. Section 2.1.5, "Gathering user feedback" offers some tips for conducting the session with users.

5. Based on user feedback, create a functional prototype using Forms Developer or Reports Developer. The following sections can help you select the appropriate objects for your prototype:
   - Section 2.2, "Creating an Effective Form"
   - Section 2.3, "Creating an Effective Report"
   - Section 2.4, "Creating an Effective Display"
6. Let users experiment with the functional prototype. Be sure to include users who were not involved in the earlier sessions so you can determine whether the application is easily grasped by new users.
7. Repeat steps 5 and 6 until you are satisfied that you have met all the objectives stated in your user requirements.

2.1.4 Building the user interface elements

Only when you have devoted sufficient time to developing your conceptual model—that is, when you fully understand your users and the tasks they perform and have designed smoothly flowing dialogs in support of those tasks—only then are you ready to begin building your user interface. This chapter contains three sections to help you choose your user interface elements carefully:

- Section 2.2, "Creating an Effective Form"
- Section 2.3, "Creating an Effective Report"
- Section 2.4, "Creating an Effective Display"

2.1.5 Gathering user feedback

When you have developed a working prototype, either on paper or with Forms Developer or Reports Developer, return to the users you interviewed in the first phase and let them experiment with it. To gather user feedback effectively:

- Produce instructions for user tests using a task-based approach.
- To ensure a broad perspective, use at least six typical users.
- Record user activity through notes, sound, and video monitoring.
- Question users about the prototype's performance.
- Get more than one designer to interpret the results.

Remember: only the actual user of your application is qualified to comment if the UI is appropriate.

After testing the prototype on users and gathering their feedback, return to the build stage, modify the user interface accordingly, then test your changes again. Continue this cycle until the interface meets the objectives you outlined in the requirement definition phase.

2.2 Creating an Effective Form

This section explains how to build an effective GUI using Form Builder.

Note: The information in this section assumes a Eurocentric viewpoint. (If you are developing for a non-Western audience, be sensitive to the cultural background of the users. If practical, have your design reviewed by several members of your target audience.)

2.2.1 Understanding forms

Before addressing specific considerations for forms, it may be helpful to briefly introduce some basic forms concepts. (Experienced Form Builder users should go to Section 2.2.2, "Guidelines for building forms".) For more details on these and other related forms topics, see the Form Builder online help and/or the Forms Developer Quick Tour.

2.2.1.1 What is a module?

When you build an application with Form Builder, you work with individual application components called *modules*. There are four types of modules in Form Builder:

Module Type	Description
Form module	A collection of objects and code routines. Some of the objects you can define in a form module include windows, text items (fields), check boxes, buttons, alerts, lists of values, and blocks of PL/SQL code called triggers.
Menu module	A collection of menus (a main menu object and any number of submenu objects) and menu item commands.

Module Type	Description
PL/SQL Library module	A collection of user-named procedures, functions, and packages that can be called from other modules in the application.
Object Library module	A collection of objects that can be used to develop applications. See Table 2–1, "Standards mechanisms" for more information.

This chapter does not address the use of PL/SQL library modules. For information on this topic, refer to the Form Builder online help.

2.2.1.2 What are forms, blocks, items, regions, and frames?

Simply put, a *form* (or *form module*) is an application that provides access to information stored in a datasource. When you look at a form, you see interface *items* such as check boxes, radio groups, and so on, which enables the user to interact with the datasource. These interface items belong to a container called a *block*. In Figure 2–4, the fields Customer ID, First name, Title, and so on all belong to the same block.

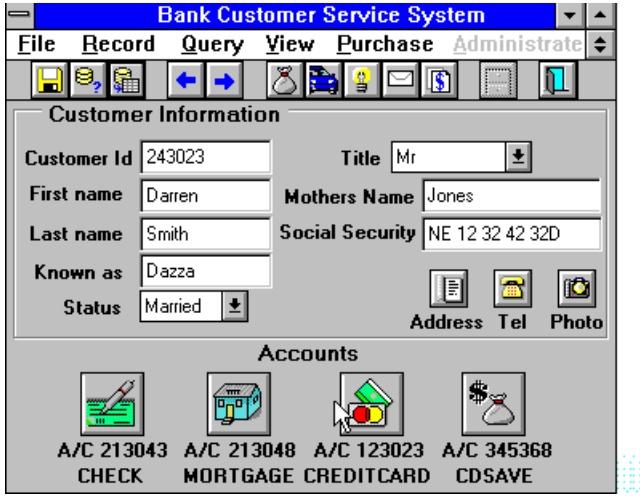


Figure 2–4 Sample form

There are two types of blocks: a *data block*, which serves as a link between the datasource and the user, and a *control block*, which is not associated with a datasource. Each data block can enable the user to view and access data from one table in the datasource. Blocks can be single-record blocks, which means that they show one row of data at a time, or multi-record blocks, which enable users to see many rows of data at once. All of the fields in Figure 2–4 are in single-record blocks.

A *region* is a rectangle or line that separates a logical grouping of fields from others in the block. In Figure 2–4, the rectangle that separates the Customer Information fields from the Accounts icons is a region.

A *frame* is a pre-defined way of arranging certain items in a block. For example, the block shown in Figure 2–4 was arranged by a frame that established its margins and offsets, the distance between the items and prompts, and so on.

2.2.1.3 What are windows and canvases?

A *window* is the container for all visual objects that make up a Form Builder application. A single form can include any number of windows; all but the simplest of forms have several windows associated with them. Several types of windows are available:

Window type	Description
Container (MDI)	Holds all other windows. It usually, but not always, contains the toolbar and main menu. (Windows only)
Modeless	Enables the user to interact with any other window, as well as the toolbar and the menu. Modeless windows are used most often in GUIs when the user is free to choose among many tasks.
Modal	Forces the user to work within a single window, then either accept or cancel the changes they have made. The toolbar and menu are not accessible. Use a modal window when the user must complete a particular task before continuing.

Here is an example of a typical Form Builder window:

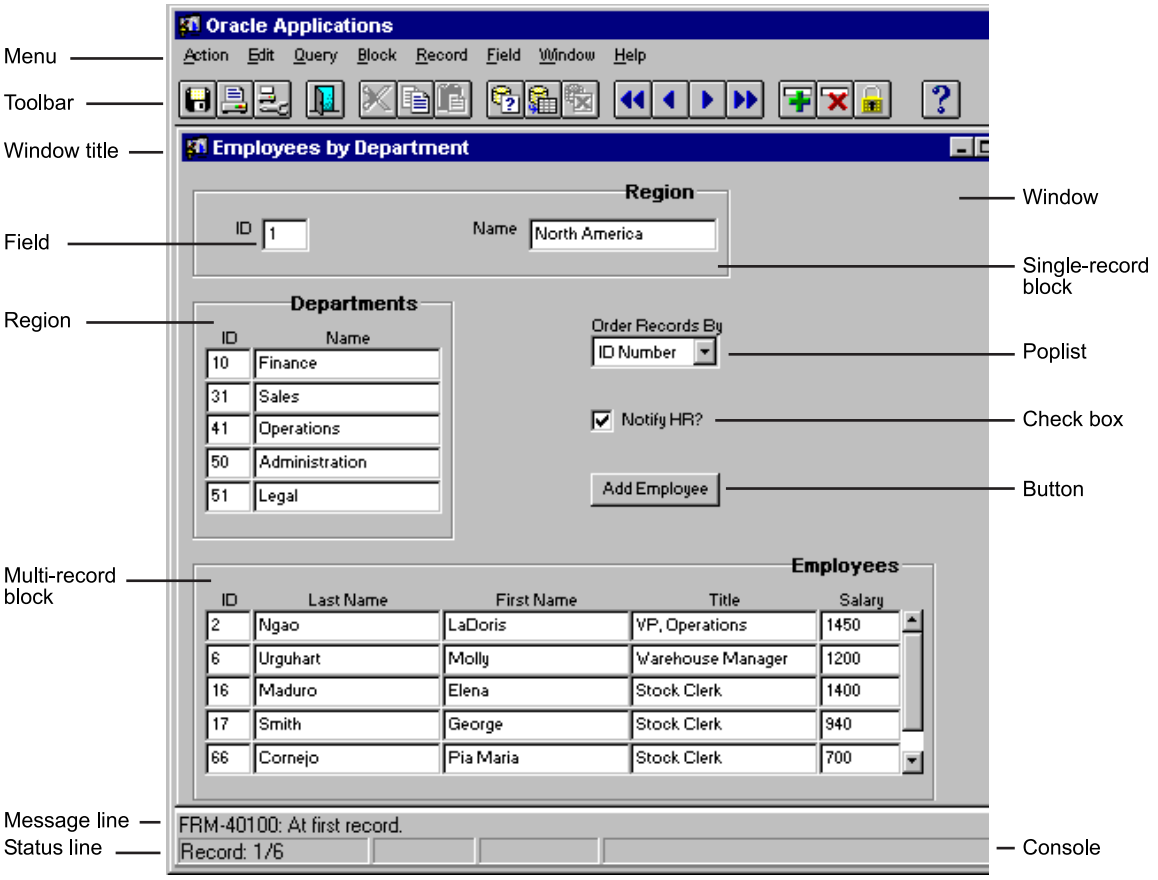


Figure 2–5 Typical Form Builder window

Like most Windows 95 Form Builder windows, this one contains:

- Window title
- Menu bar and pull-down menus
- Buttons and other control items that do not correspond to the data
- Data items in the blocks
- Console, which includes the message line and status line

A *canvas* is the background object upon which interface items appear. There are four types of canvases:

Canvas Type	Description
Content canvas	Occupies the entire pane of the window in which it is displayed (and possibly more, if the window enables scrolling). Every window has at least one content canvas.
Stacked canvas	Displayed atop—or stacked on—the content canvas assigned to the current window. Stacked canvases are useful for conditionally obscuring areas of the content canvas—unpopulated fields, for example. Through the use of *viewports* you can control how much of a stacked canvas is visible.
Tab canvas	A set of tabs that enable you to group and display a large amount of related information on a single dynamic canvas.
Toolbar canvas	Used to create toolbars for individual windows.

Each window may display one or more canvases. You can also conditionally display a canvas in a window, depending on whether certain conditions are met.

2.2.2 Guidelines for building forms

The following sections offer specific recommendations for building an effective GUI with Form Builder:

- Section 2.2.2.1, "Using object libraries"
- Section 2.2.2.2, "Understanding basic design principles"
- Section 2.2.2.3, "Adding color"
- Section 2.2.2.4, "Creating canvases"
- Section 2.2.2.5, "Creating windows"
- Section 2.2.2.6, "Creating regions"
- Section 2.2.2.7, "Adding items to blocks"
- Section 2.2.2.8, "Designing messages"
- Section 2.2.2.9, "Implementing online help"

- Section 2.2.2.10, "Building effective menus"

2.2.2.1 Using object libraries

Perhaps the most important means of standardization available to you as a form developer is the object library. An object library is a set of objects and standards that you create; each object or standard can determine the appearance and layout of an entire frame, window, or region. When housed in an object library, these objects become available to all the developers on your project or site, thus ensuring that even developers working at different locations can produce an application—or different modules within the same application—with a common look and feel. Through the use of subclassing, each developer can ensure that changes made to the objects in the object library are propagated throughout all applications that use them.

A good strategy for using object libraries is to create a separate one for each logical grouping of standards. For example, you may want to have one object library for corporate standards that you make available company-wide, and another tailored for the specific needs of your project.

To help you get started building your own object libraries, Form Builder provides two samples:

- Standard Object Library, which contains objects for Windows 95-only deployments where multi-language support is not a requirement
- Oracle Applications Object Library, recommended for multi-platform deployments

Before you create your object library, it's a good idea to test the contents of the Standard or Oracle Application Object Libraries to see what works well and what you need to modify.

To test:	Do this:
The items in the Standard Object Library in a **data block**	**1.** Use the Data Block Wizard to create a data block. **2.** Click an item in your control or data block, then click the right mouse button. A list of Smart Classes applicable to that item is displayed; click the SmartClass you want.
The items in the Standard Object Library in a **control block**	**1.** Open `STNDRD20.OLB`. **2.** Drag and drop the item(s) into the block.

To test:	Do this:
Only the visual attributes in the Standard Object Library	Open the STNDRD20.OLB template form.
The objects in the Oracle Application Object Library	Open the APPSTDS.OLB template form.

If you use the Standard Object Library, be sure to subclass all the attributes under the VAGs tab to the Visual Attributes node in your form. Many of the standards are based upon these visual attributes and will not display correctly if the visual attributes are not applied. By subclassing (rather than copying) the visual attributes, you ensure that you always have access to the latest definitions.

If you know that you will be using a particular set of visual attribute groups in all or most of your forms, create a template form that already contains the visual attribute groups subclassed from the standard object library. Then you can name this template when prompted by the Layout Wizard.

For more information on the object libraries, see the help topics "Standard Object Library" and "Oracle Application Object Library" under "Reusable Components" in the Forms Developer Demos online help.

2.2.2.2 Understanding basic design principles

Here are some general guidelines for building forms:

- Use a Real Coordinate system with a measurement unit of inches, centimeters, or points. Choose a single unit and use it across all modules.

Points	Often the easiest to use, since you can designate sizes in whole numbers. Since text is always specified in points, it's easier to size objects relative to text if the objects are in points as well.
Inches and centimeters	Enables you to specify a higher precision than points (but it's not as easy to compare the size of objects to text). Useful if your target environment is exclusively SVGA or better.
Pixels	Strongly discouraged. Use only if you are certain that all users have identical screens and resolutions and will continue to have them in the future.
Characters	Use only if you intend to deploy to a character mode platform.

- Place users in control by enabling them to enable or disable dialogs wherever it makes sense to do so. Making this determination requires you to carefully balance your knowledge of the users with the freedoms or restraints imposed by their working conditions.

  Example: All users should be enabled simple freedoms, like the ability to interrupt an application and resume it later on. But enabling users the ability to completely rearrange a company-issued invoice may not be wise, since it affords the user power to disregard company standards.

  The extent to which users should have control is also determined by the user's experience level. If you are developing an application for both experienced and inexperienced users, consider providing a wizard to provide step-by-step assistance for those who want it, along with manual alternatives.
- Make it obvious to users when a task is finished, either by closing a window, opening another window, or displaying an informational message.
- Make windows only as large as necessary.
- Use blank space as a way to group information.
- Use the frame objects in the Standard Object Library to help you obtain a consistent layout style across form modules.
- Orient screen layouts based on a top-to-bottom task sequence. Arrange blocks, regions, and items in the order they will be used, from left-to-right, then top-to-bottom.
- In single-record blocks, left-align items where possible. (Right-align fields containing currency and numbers.) In multi-record blocks, stack items horizontally and align them along the top.

2.2.2.3 Adding color

- Use color sparingly, and only to get the user's attention.
- Use color meaningfully and consistently. Example: Use color coding to differentiate between required, optional, and display-only fields, making sure that all such fields are color coded the same way.
- Consider enabling users to change the color scheme, if possible.
- Do not rely on color alone for communicating status or other information; always provide alternative cues, such as sound or other highlights. For example, if you display a negative total in red, include parentheses so that the message is clear without the use of color. Also, avoid references to specific colors in messages, as many users are color blind.
- Use object libraries or Visual Attributes to standardize color usage.
- When choosing colors, remember:
  - Red and blue combinations are hard on the eyes.
  - Blue text has a receding effect.
  - Deep blue backgrounds are hard on the eyes over long periods, as are other bright colors.
  - A significant number of people have color-identification problems, especially with red-green.
  - Colors have different implications in different countries. Follow the cultural color coding in your target market, observing the needs of different professions, situations, and so on. For example, while green generally has positive connotations for most of the western world, to those in chemical-related professions the color green might mean danger.

Color	Implies...
Blue	Cool
Black	Profit (financial)
Green	Go, OK, Danger (for chemists)
Red	Hot, Stop, Danger, Loss (financial)
Yellow	Warning, Attention

2.2.2.4 Creating canvases

The following table presents recommendations for creating canvases.

Table 2–2 Recommendations for creating canvases

Canvas Type	Recommendation
General	■ Provide plenty of white space between items and regions. ■ Consider placing optional information on separate canvases. ■ Avoid scrolling windows, if possible. Studies have shown that productivity decreases sharply when the user has to scroll a window to complete a task. ■ Plan separate windows for canvases that need to be viewed concurrently. ■ Although your planned layouts may fit comfortably on a monitor using Super VGA mode, they may scroll off-screen in different resolutions, like VGA. Test your layout on all your users' monitors.
Content canvas	■ Set content canvases to Display immediately. ■ Remember that the view size for a content canvas is determined by the current size of its assigned window. ■ Consider using non-white canvases so that the bevel effects of objects on the canvas are maximized. In addition, white backgrounds are often so bright that they can be tiring. ■ Use one content canvas per window. Using more than one can be confusing if the user does not understand why the entire window is being replaced. If you do use more than one content canvas, make sure they are logically related, and require the user to move between them explicitly. One successful implementation of multiple content canvases is a word processing application in which the user chooses between several views of the same document: print preview, normal, and outline.

Table 2–2 Recommendations for creating canvases

Canvas Type	Recommendation
Stacked canvas	■ Use stacked canvases to hide and display groups of objects, including boilerplate. ■ Size the stacked canvas only large enough to contain the necessary items. ■ Be sure you know how stacked canvases behave before you implement them. For example, if the user uses Next Field or Next Record to navigate to a field that is obscured by a stacked canvas, the stacked canvas seems to disappear--that is, it is automatically placed beneath the content canvas. To re-display the stacked canvas, users must either navigate to an item on the stacked canvas or navigate away from the stacked canvas and select the Show Canvas action.
Tabbed canvas	■ Limit the number of tabs to 4-6. ■ Use tabs to organize related information about a single object. For example, employee information such as salary, benefits, and job description might work well as a tabbed dialog.

2.2.2.5 Creating windows

The following table presents recommendations for creating windows.

Table 2–3 Recommendations for windows

Attribute	Recommendations
General	■ Do not use bevels around the edge of the window. ■ Inherit color settings from the environment. ■ Leave the top and bottom lines of the window blank, except for buttons and coordination check boxes. ■ Leave the left and right edge character cell columns blank, except for region lines and block boundary lines. ■ Use modeless (non-modal) windows to allow scrolling, and for "leave and come back" navigation (use the STD_DIALOG_WINDOW_MODELESS object in the Standard Object Library). ■ Use modal windows to prevent mouse navigation elsewhere and for dependent tasks that are part of a procedure (use the STD_DIALOG_WINDOW_MODAL object in the Standard Object Library).
Title	■ Title each window in a form uniquely so that iconified names and entries in the Windows menu are significant.
Position	■ Make sure each window is fully visible when it is first opened. ■ Make all windows moveable. ■ Retain window positions when a form is exited.
Scrollbar	■ Design your windows so that scrolling is not required by default. Scrolling is acceptable only when provided as a result of the user re-sizing the window.
Toolbar	■ Place the toolbar only on the container window (on Windows) or the root window (on all other platforms). ■ Provide hints for the toolbar buttons in tooltip help displayed just beneath each button as the mouse passes over it. (See Section 2.2.2.9.1, "Implementing Tooltips".)

2.2.2.5.1 Choosing a title for modeless windows

While the STD_DIALOG_WINDOW_MODELESS object in the Standard Object Library addresses all issues pertaining to positioning, closing, resizing, and placement, you still have to choose your own title. When doing so:

- If the window performs a product-specific task, use the format <*Verb*><*Noun*>, as in Transfer Items, Post Journals, and AutoCreate Quotes.
- Pluralize window names—that is, use "Items" instead of "Item"—except when the window pertains to a single instance of data.
- Provide context for child windows in the form <window title> - <context>, where context is the topmost master record or, for a new record, [New].

  Examples: Assignments (OR1) - [John Doe]

  Purchase Order Lines (ABC) - [New]

2.2.2.6 Creating regions

The following table presents recommendations for creating regions.

Table 2–4 Recommendations for regions

Attribute	Recommendation
General	■ Avoid creating regions or adding boilerplate lines to group items unless doing so is meaningful to the user and improves the usability of the screen. ■ Make the line or rectangle creating the region black, with an inset bevel. ■ Use a frame for regions containing an entire block. A frame has properties that control the layout of the items within it, such as the space between the frame and items (margin), spacing, and visual attributes. Using standard frames ensures the consistency of your window layout. (Although the Layout Wizard creates a frame, you can always override it by applying a frame stored in your object library.)

Table 2–4 Recommendations for regions

Attribute	Recommendation
Title	■ Add a title to the region unless the information contained within is obvious. Use boldface. ■ Position the title on top of the rectangle or line, leaving one leading and one trailing space in the title text. ■ To display the title, use one of these widgets: • Boilerplate (for static region titles) • Frame title (for frames) • Display item, designed to look like boilerplate (for dynamic region titles) • Poplists (for alternative regions) • Check boxes (if an entire region is applicable or non-applicable)

2.2.2.7 Adding items to blocks

The following table should help you decide when to choose one form item over another. It also presents some guidelines that you can use if you decide to modify an object or standard in the Standard Object Library. The items are presented in alphabetical order.

Table 2–5 Recommendations for items

Item	When to use	Recommendations
Boilerplate text	■ Use for text that is neither a prompt nor a title.	■ Use mixed case. ■ Avoid overuse of italics and underlining. ■ Use font styles consistently. For example, if you use bold for emphasis, do not use bold for any other purpose. ■ Avoid excessive variations of fonts, sizes and colors.

Table 2–5 Recommendations for items

Item	When to use	Recommendations
Buttons (non-iconic)	■ Use as dialog responses (in modal windows) and for item-related actions.	■ Use one of the STD_BUTTON_*type* objects in the Standard Object Library. ■ Limit six to a window. Arrange in a single row, if possible, or a single column. ■ Align buttons, leaving 0.1" space between them. Separate logical groupings of buttons by 0.5". ■ Leave 0.1" between the right edge of the rightmost button and the right edge of the window. ■ Capitalize label words; for example, 'Print Invoice'. ■ Use an ellipsis (...) at the end of a button label if the button opens a modal window or if the user must provide more information about the action in another window (modal or not) before the action can be completed. ■ Place OK and Cancel buttons together. ■ Put affirmative and cancellation buttons first, unique buttons last. ■ Use chevrons (>>) to indicate that the dialog will be expanded. ■ For labelled buttons (except OK and Cancel), always provide an accelerator key (underlined letter). • Use the first letter of the first or second word in the label ("F" for "File" or "P" for "Start Posting"). If a stronger link exists (like "X" for "Exit"), use that letter. • Use consonants instead of vowels when possible. • Make access keys unique within a window. Ensure they do not conflict with the keys used by the top level of the menu.
Check boxes	■ Use only when the label on the check box can clearly be thought of as having "true" (checked) and "false" (unchecked) states. Otherwise, use a radio button group with two items.	■ Use the STD_CHECKBOX object in the Standard Object Library. ■ Use positive statement labels: Not good: Don't show this alert in the future. Better: Show this alert in the future.

Table 2–5 Recommendations for items

Item	When to use	Recommendations
Display items	■ Use for display-only fields in which the user can never type; for example, the Total field in a financial application.	■ Use the `STD_DISPLAY_ITEM` object in the Standard Object Library.
Icons	■ Use only for frequent or critical actions. ■ Use where a picture conveniently conveys a task or mimics a real-world object.	■ Place frequently used buttons on a toolbar. ■ Group related tools together and separate groups with white space. ■ Disable buttons that are unavailable. ■ Always provide tooltips, as users are often confused by the meaning of icons. ■ Icons are often cultural. Be aware that you may need to translate them. ■ Avoid "visual puns", such as a running figure for "Run". Their meanings are not obvious, and will certainly not be understood in other languages.
Lists (see also Poplists and T-Lists)	■ Use when it is easier for the user to select a value than to type in a value. ■ Use for data entry and display of text values in a selectable list format. ■ Use when displayed value entries are relatively short (up to 30 characters each). ■ Use the combo-box style if the user may enter new values.	■ For 15 entries or less, use a poplist. For more than 15 entries, use an LOV (List of Values). For more than 30 entries with a lot of real estate, use a T-List. ■ Make all related fields the same length. ■ Use the color of the canvas background for text items that have become non-enterable.
LOVs	■ Use when the user must select from a list of more than 15 rows or to show several columns of data.	■ Automatically select a row for the user when there is only one valid value. ■ Move the cursor automatically to the next field after a selection is made. ■ If there are more than 100 rows in the LOV, prompt the user to reduce the list of valid values before making a selection.
Poplists	■ Use when only one value is applicable and the list of choices is 15 or less.	■ Before implementing a poplist, consider whether frequent users can type faster than they can select.

Table 2–5 Recommendations for items

Item	When to use	Recommendations
Pop-up menus	■ Use to associate menu options with an item, rather than the whole application. ■ Use to provide access to frequently used commands.	■ Use the STD_POPUP_MENU_ITEM object in the Standard Object Library.
Prompts	■ Use as labels for fields, check boxes, lists, etc.	■ Place toward the top or left of the element they are describing. ■ Always place single-record blocks prompts to the left of the field and multi-record block prompts above the field. ■ Use the terms "From" and "To" to identify fields involved in a range rather than "Start" and "End" or "Low" and "High". ■ For percentages, place the percent sign (%) after the field. Do not include it in the prompt.
Radio groups	■ Use to present mutually exclusive choices. ■ Use to set a 'mode', such as what type of information will be displayed.	■ Use the STD_RADIO_GROUP object in the Standard Object Library. ■ Use vertical orientation instead of horizontal. ■ Group related buttons into radio groups with a title. ■ If the choices are binary (ON / OFF, YES / NO), use a check box instead. ■ Always provide a default value.
T-Lists	■ Use only when one value is applicable and the list of choices is never expected to grow beyond 30.	■ Always show at least five rows of data. ■ Use only in forms with a lot of available real estate.
Text items	■ Use for data entry and display of character values. ■ Use for lengthy or unprepared values (that is, those that do not appear in a short, pre-defined list).	■ Use the color of the canvas background for text items that have become non-enterable. ■ Use a bevel if the user can enter values in the field. ■ Use the STD_TEXT_ITEM or one of the STD_DATE_*type* objects in the Standard Object Library.

2.2.2.8 Designing messages

Messages are shown either in the window console area or in popup windows called *alerts*. How you display messages depends upon their type and whether a reply is required by the user. Here are some suggestions:

Table 2–6 Displaying messages

Message Type	Recommendations	Example
Errors	■ To present an error message, use the `STD_ALERT_STOP` object under the Alerts tab in the Standard Object Library. ■ Use when an error is serious enough to halt processing. Include a stop sign icon in the dialog. ■ Use sparingly.	"You do not have sufficient authority to approve this Order."
Warning	■ Use the `STD_ALERT_CAUTION_1, STD_ALERT_CAUTION_2`, or `STD_ALERT_CAUTION_3` objects under the Alerts tab in the Standard Object Library. While in the Library, click the object once to see a description of the message text. ■ Use to present a question that the user must respond to before processing continues. Include a yield sign icon (!) in the dialog. ■ Keep the warning short and concise. For example, use "Delete this order?" rather than "Do you really want to delete this order?" ■ Phrase questions positively ("Save changes?" rather than "Are you sure you don't want to save changes?")	"Copy all lines on this invoice?"
Information	■ Use the `STD_ALERT_INFORMATION` object in the Standard Object Library. ■ Use to present messages that the user must acknowledge when no choice is involved. Include the information icon (the letter "i" in a circle) and the OK button.	"Line and Shipment Quantities currently do not match." "There are items awaiting your attention."
Hints	■ Appears on the Form Builder message line in the Console. ■ Use to present messages of very little consequence, or process indicators that do not require a response.	"Working..." "At first record." "Processed Order line 12 of 37."

2.2.2.8.1 Creating Message Text

If possible, error messages should include:

- What was done
- Cause (why it was wrong)
- Action (how to fix it)

Here are some examples of bad and good message text:

Bad	Good
Invalid Date	"Please re-enter the date as DD-MON-YY."
Do not enter a start date later than the ending date	"The start date must be earlier than the end date."
Error: 1623, constraint violation	"Please re-enter a unique value for this field."
You should not receive this message	Don't display the message at all.
Tool lost at sea	Replace it with an appropriate message or take it out altogether

When writing message text, try to adhere to these guidelines:

Recommendation	Example
Use active voice.	"Do this now", not "This will need to be done"
Use imperative voice.	"Enter a commission plan", not "You can enter a commission plan"
Use "can" instead of "may" or "could".	"You cannot delete a printed release", not "You may not delete a printed release"
Refer to actual field names when possible.	If a field is labelled "Sales Associate", don't use the message "Please enter a different salesperson".
Use uppercase for commands and keywords.	ALTER CLUSTER statement is no longer supported."
Avoid the use of humor.	You made a boo-boo!

Recommendation	Example
Avoid accusatory messages. Do not insinuate that the user is at fault. Do not mention the user's mistake unless it pertains to the problem's solution.	Instead of "You didn't choose a value", try "Please choose a value".
When a message contains instructions, use "please".	"Please choose a value" is preferred over "Choose a value".
Address the user as "you", not "the user". Avoid using "I", "He", or "She".	Instead of "The user should back up his modules", try "Please back up your modules".
Consider providing context-sensitive help when errors occur.	See Section 2.2.2.9.2, "Implementing Online Help".

2.2.2.9 Implementing online help

This section discusses using two types of online help:

Tooltips	Also known as popup hints and microhelp. Displayed when the user moves the mouse over an item on the screen.
Online Help	Contains context-sensitive help and hypertext links that enable users to jump to related topics.

2.2.2.9.1 Implementing Tooltips

Each item has a property called Tooltip and another called Tooltip Visual Attribute Group. In the Property Palette's Tooltip property field, enter the text you want to display in the pop-up. To ensure consistency across your application, apply the `STD_TOOLTIP` visual attribute from the Standard Object Library. If you don't apply a visual attribute, the tooltip uses a platform-specific default.

2.2.2.9.2 Implementing Online Help

- Consider using help authoring tools.
- Create standalone topics that are hyperlinked to other topics.
- Keep text short and concise.
- Provide a Help button on each dialog, a Help option on the main menu, and a Help button on the toolbar.

2.2.2.10 Building effective menus

Form Builder provides a default menu for every form. The default menu includes commands for all basic database operations, including querying, inserting, and deleting. If your application has specific requirements not met by the default menu you can quickly create a custom menu. (See "Creating a menu" in the Form Builder online help for instructions.) While building a menu, keep the following ideas in mind:

- Organize commands according to the tasks they belong to.
- Although Forms Developer supports scrolling menubars, try to keep valid items on the screen where the user can see them.
- Disable items that are unavailable.
- Limit submenus to two levels, if possible.
- Capitalize labels.
- Use the standard menus in the demos and add-ons as models for your own menus.

2.3 Creating an Effective Report

The first steps in using Report Builder to design an effective report are the same as those for designing an effective form or display. Before reading the rest of this section, it's a good idea to read Section 2.1.2, "Defining user requirements", if you haven't already.

Here are a few questions to help you determine the user requirements for your report:

- What data will people viewing the report want, and in what priority?
- Will users want to "drill down" on data, so they can see more details? If so, you'll want to include buttons in your reports. Buttons can have blocks of PL/SQL code associated with them, so they can invoke secondary reports, play videos or sounds, and so on.
- Will users want charts in the report to present data visually? If so, what data? You can create a chart in Graphics Builder, then pass data to the chart from Report Builder (instead of performing a second query).
- If users modify data using a form application, will they want to print the data afterward? If so, you'll want to call the report from a form, and have the form pass data to the report.

- Will users want a report to be embedded in a form? If so, you'll want to design a template that has font and color standards similar to your forms.
- Will users want to view the report in HTML, PDF, or hardcopy? Will they want to make a few formatting changes in the Live Previewer before (optionally) printing? If so, you will need to specify the report destination parameter (DESTYPE), or enable your users to do so.
- Will users want to specify parameters for a report, as in "Show only the top 10 sales for the userid SCOTT"? If so, you will need to create user parameters, and have users specify their values in a form or the Runtime Parameter dialog.
- Based on network traffic and machine performance, should the report run in a client/server or 3-tiered architecture?
- Do you have a corporate standard that you want to propagate in the reports? If so, you should define standard templates.
- For Web reports, will the number of reports be static, or do you want to dynamically generate the Web sites?

2.3.1 Understanding Reports

Before addressing specific considerations for reports, it may be helpful to briefly introduce some basic reports concepts. (Experienced users of Report Builder should skip this section.) For more details on these and other related reports topics, see the Report Builder online help and/or the Report Builder section of the Reports Developer Quick Tour.

When you build a report, you work with two application components:

Module Type	Description
Report module	A collection of objects and code routines. Some of the objects you can define in a report module include repeating frames, frames, fields, boilerplate, anchors, and blocks of PL/SQL code called triggers.
PL/SQL Library module	A collection of user-named procedures, functions, and packages that can be called from other modules in the application.

This section does not address the use of PL/SQL libraries. For information on this topic, refer to the Report Builder online help.

For some reports, you will use the Report Wizard (to choose a report type, define a data model, and a layout for the data) and the Report Editor's Live Previewer (to fine-tune the report). For other reports, you will use other views of the Report Editor:

View	Used to:
Data Model view	Create a report with more than one query.
Layout Model view	■ Create reports with multiple sections (e.g., a single report with a tabular and matrix style) ■ Add new layout objects (e.g., buttons) ■ Control how objects are sized or positioned
Parameter Form view	Present users with a dialog in which they can specify parameter values before running the report.

Because this chapter discusses how to create visually effective applications, the remainder of this section focuses on using the templates, objects, and settings found in the Layout Model view.

2.3.2 Using Templates in Report Builder

Perhaps the most important means of standardization available to you as a report developer is the *template*. A template is a collection of boilerplate objects, and layout and report settings that determine the appearance of an entire report. Several templates are shipped with Report Builder, and you can create your own. By creating corporate or group templates and making them available to your entire development team, you can ensure a common look and feel. For instructions on how to create a template, see the Report Builder online help.

2.3.3 Understanding Layout Objects

The Layout view of the Report Editor may contain the following objects:

Object	Description
Frames	Containers that control repeating frames, fields, boilerplate, buttons, and child frames. Unlike Form Builder frames, Report Builder frames do not have formatting properties that control the location of child objects.
Repeating frames	Containers that control: ■ Fields containing report data ■ Other objects owned by the repeating frame
Fields	Containers that display the report data, dates, page numbers, and so on.
Boilerplate	Text or graphics that appear as often as required by the object that surrounds it (the report, frame, or repeating frame), or to which it is attached.
Anchors	Objects that determine how two objects in a report layout relate to one another (i.e., parent/child relationships and relative positioning).
Buttons	Objects that perform an action when users click on them.

With the exception of anchors, layout objects may have format triggers, such as PL/SQL blocks that are invoked each time the object is activated.

2.3.4 Controlling Layout Objects in Report Builder

When designing a report, remember that the size of the entire report and the size of many of its individual objects may vary, which can affect pagination in a printed report. Consider a report based on this query:

```
select ename, sal from emp
where sal > 2000
```

The size of this report and its objects is based on several factors:

- The amount of data that satisfies the query, which can range from a few records, to hundreds or thousands of records.

- When you run the report. For example, the number of records might change dramatically the week after pay raises are distributed (for example, the number of people with salaries above 2000 might increase).
- Whether group filters or format triggers are used, which exclude data or objects from the report.

Instances of the same object may also vary in size. For example, suppose you have a VARCHAR2 column in the database called COMMENTS. For one record, COMMENTS might contain two sentences. For another, it might contain 10 sentences. The size of the field in your layout that corresponds to the COMMENTS column must then be able to accommodate values of different length. In addition, objects around that field may have to be "pushed" or "pulled" to avoid being overwritten or leaving large gaps in the report.

Fortunately, Report Builder provides a variety of mechanisms in the Layout View of the Report Editor that enable you to control how objects are sized and positioned. These mechanisms are described in the following sections:

- Section 2.3.4.1, "Using anchors"
- Section 2.3.4.2, "Using the Print Object On and Base Printing On properties"
- Section 2.3.4.3, "Understanding Horizontal and Vertical Elasticity"
- Section 2.3.4.4, "Using the Page Break Before and After property"
- Section 2.3.4.5, "Using the Page Protect property"
- Section 2.3.4.6, "Using the Keep with Anchoring Object property"

2.3.4.1 Using anchors

Anchors determine how objects in a report layout relate to one another. When two objects are anchored together, one object is considered the parent and the other the child. By defining parent-child relationships between objects, anchors establish a hierarchy for the objects in a report. Based on this hierarchy of objects, Report Builder decides how objects should be printed in relation to each other, whether it should attempt to keep the two objects on the same page, and how objects should be pushed or pulled depending on the size of surrounding objects.

Anchors can be created in one of two ways:

- Automatically, by Report Builder. This is known as an implicit anchor. In most cases, implicit anchors are the only ones you need.
- By you, using the Anchor Tool in the Layout view of the Report Editor. This is known as an explicit anchor. Explicit anchors are necessary only when you need

to override the implicit anchors for some reason. See the topic "About anchors" in the Report Builder online help.

2.3.4.2 Using the Print Object On and Base Printing On properties

The Print Object On property determines the frequency with which an object appears in a report. The Base Printing On property specifies the object on which to base the Print Object On property.

For example, if you specify a Print Object On of All Pages and a Base Printing On of Anchoring Object, the object is triggered to print on every logical page on which its anchoring object (parent object) appears. Objects created by the Report Wizard have these properties set for them. In most cases, the values that Report Builder chooses are the best ones for the object. The only time you should need to set these properties yourself is when you want to override the default value set by Report Builder.

In applying the Print Object On property, Report Builder considers the first page of an object to be the first logical page on which some part of the object is printed. Likewise, the last page is considered to be the last logical page on which some part of the object is printed. For example, if you specify a Print Object On of First Page and a Base Printing On of Enclosing Object, the object will be triggered to print on the first logical page on which its enclosing object appears.

Notes:

- If an object is inside a repeating frame, Base Printing On refers to each instance of the repeating frame. If the object is outside the repeating frame and explicitly anchored to it, Base Printing On refers to the repeating frame as a whole.
- Just because an object is triggered to print on a logical page does not mean it will print on that logical page. Other settings (e.g., Page Break Before) or the amount of space available on the page may cause Report Builder to print an object on a page other than the one on which it was initially triggered to print.

For more information, refer to the Report Builder online help, index entries: Print Object On and Base Printing On.

2.3.4.3 Understanding Horizontal and Vertical Elasticity

The Horizontal and Vertical Elasticity properties determine how the horizontal and vertical sizes of the object may change at runtime to accommodate the objects or data within it:

- For frames and repeating frames, elasticity defines whether the size of the frame or repeating frame should vary with the objects inside of it.
- For objects containing text, elasticity defines whether the field or boilerplate should vary with the size of the text. Fixed size text will wrap within the defined size of the object and may be truncated if there is not enough room. Number or date data will appear as asterisks if the data cannot fit within the defined size.
- For images, drawings, and chart objects, Report Builder uses proportional scaling. The elasticity options for images, drawings, and chart objects determine the scaling factor.

Objects created by the Report Wizard have these properties set for them. In most cases, the values that Report Builder chooses are the best ones for the object. The only time you should need to set these properties yourself is when you want to override the default value set by Report Builder.

Different elasticity settings can produce unexpected results in the output. For example, if an object with variable horizontal elasticity contracts, all objects to the right are moved to the left, since these objects are implicitly anchored to the variable object.

For more information, refer to the Report Builder online help, index entries: Horizontal Elasticity and Vertical Elasticity.

2.3.4.4 Using the Page Break Before and After property

Unlike word processing documents, reports and their objects can vary in size and position at runtime. As a result, page breaks in a report can be difficult to predict.

- Use the Page Break Before property to indicate that you want an object to be formatted on the page **after** the page on which it is initially triggered to print. Note that this does not necessarily mean that all the objects below the object with Page Break Before will move to the next page. If one of the objects below does not have Page Break Before set and can fit on the page, it may print above the object which has Page Break Before set.
- Use the Page Break After property to indicate that you want all children of the object to move to the next page. In other words, any object that is a child object of an anchor (implicit or explicit) to this object will be treated as if it has Page Break Before set. Note that this does not necessarily mean that all the objects below the object with Page Break After will move to the next page. If one of the objects below does not have Page Break After set and is not a child of the other object, it might print above the object which has Page Break After set.

For more information, refer to the Report Builder online help, index entries: Page Break Before and Page Break After.

2.3.4.5 Using the Page Protect property

Use the Page Protect property to try to keep the entire object and its contents on the same logical page. If the contents of the object cannot fit, they are moved to the next logical page. Note that this does not necessarily mean that all the objects below the object with Page Protect set will move to the next page. If one of the objects below can fit on the page, it might print above the object which has Page Protect set.

For more information, refer to the Report Builder online help, index entry: Page Protect.

2.3.4.6 Using the Keep with Anchoring Object property

Use the Keep with Anchoring Object property to keep an object and the object to which it is anchored on the same logical page. If the object, its anchoring object, or both cannot fit on the logical page, they are moved to the next logical page.

If you set Keep with Anchoring Object for a repeating frame, the first instance of the repeating frame must be able to fit on the same page as its anchoring object. Otherwise, the Keep With Anchoring Object condition is not satisfied. If you set Keep With Anchoring Object to Yes for any layout object other than a repeating frame, the object must be able to format entirely on the same page as its anchoring object.

The anchor between the two objects may be explicit or implicit. Consequently, Keep With Anchoring Object may have an effect even if you have not explicitly created an anchor between two objects.

2.4 Creating an Effective Display

Graphics Builder enables you to produce displays for inclusion in both forms and reports. A display can be an application by itself, or included in a form or report.

Use displays when you want to:

- Show relationships between different categories (number of tennis shoes sold as compared to dress shoes)
- Show trends rather than specific values
- Provide user interaction with graphical areas and shapes (maps, sectors of images, and so on). Graphics Builder allows you to respond to mouse

interactions with the shapes that you create in the layout editor. Irregular, transparent buttons can be placed over areas of a diagram or bitmapped image so that users can effectively make selections from pictures.

When creating graphics, keep the following guidelines in mind:

- Keep things simple. Displays containing too many lines, bars, slices, and so on can quickly overwhelm users and render your graph or chart unusable. If you have a lot of data, summarize it at the highest level and use drill-downs to present more detailed information. Or, consider breaking up a complicated graph into smaller, individual graphs, then creating a form from which users can select which graph they want to view.
- Use 3-D effects only if they help communicate information, as they can be resource-intensive.
- Test your graphics on all the display devices in your deployment environment and make sure they perform well even on the lowest resolution monitor.
- Use colors to show transition; use primary colors to show differences. See Section 2.2.2.3, "Adding color" for more information on using color.
- Use legends for complicated graphs.
- Remember, you can pass mouse events from forms to graphics modules. For example, you can create a When-Mouse-Click trigger on a form's chart item and call the OG.MouseDown procedure from this trigger to pass mouse information to a display. The display can then return information to the form, including details of which sector in the display was clicked by the user. See the Graphics Builder demo called "Map Example" in the product's standard demo set for more information.

2.4.0.7 Choosing the Right Graph

Here are some guidelines for implementing the most commonly used displays:

Table 2–7 Recommendations for displays

Display type	When to use:	Recommendations
Bar graph	Showing relationships between discrete objects and their related values.	Limit bars to 20-25.
Pie chart	Showing part-to-whole relationships. Typically used to show percentage values.	Limit slices to 10.

Table 2–7 Recommendations for displays

Display type	When to use:	Recommendations
Line chart	Show the cumulative effect of continuous data.	Limit lines to 6-8.
Double-Y	Comparing data within a large range of values.	Limit plots to 4 or less.
Gantt	Scheduling and date duration data.	Limit bars to 40-50.
High-low	Displaying daily temperature values, stock market values, and similar data tracking high, low, and current values.	Limit rows to 30 or less.
Mixed	Comparing actual values (bar) to projected values (line).	Limit rows to 30 or less.
Scatter	Showing relationships between numeric data on the X and Y axis.	Limit rows to less than 50 per inch.

3

Performance Suggestions

This chapter details suggestions for improving performance of your applications. It includes the following sections:

- Summary
- Introduction: What Is Performance?
- Measuring Performance
- General Guidelines for Performance Improvement
- In a Client/Server Structure
- In a Three-Tier Structure

3.1 Summary

The following table summarizes the available performance suggestions, and indicates where a detailed explanation can be found in this chapter.

Before getting into the details, you should read the introductory information in Section 3.2. The material on performance measurement in Section 3.3 may also be helpful.

The suggestions are grouped according to their scope, in this sequence:

1. those that apply to any Forms Developer or Reports Developer application, in any environment
2. those for specific Builder applications (Forms, Reports, or Graphics), in any environment
3. those for any Forms Developer or Reports Developer application in a client/server (2-tier) environment

4. those for any Forms Developer or Reports Developer application in a 3-tier environment

Performance Suggestion	Explanation
For any application:	
Upgrade your software	on page 3-10
Upgrade your hardware	on page 3-11
Use array processing	on page 3-11
Eliminate redundant queries	on page 3-11
Improve your data model	on page 3-12
Choose one of the database server's optimizers	on page 3-12
Perform your calculations within your query SQL	on page 3-13
Avoid using explicit cursors	on page 3-13
Use group filters	on page 3-14
Share work between components	on page 3-14
Move wait time forward	on page 3-14
For any Forms application: (all the general suggestions also apply to Forms)	
Tune array processing	on page 3-15
Base data blocks on stored procedures	on page 3-15
Optimize SQL processing in transactions	on page 3-17
Optimize SQL processing in triggers	on page 3-18
Control inter-form navigation	on page 3-18
Raise the record group fetch size	on page 3-18
Use LOBs instead of LONGs	on page 3-18
Erase global variables after use	on page 3-19
Reduce widget creation	on page 3-19
Examine the necessity for locking	on page 3-19

Performance Suggestion	Explanation
For any Reports application: (all the general suggestions also apply to Reports)	
Reduce layout overhead	on page 3-20
Use format triggers carefully	on page 3-20
Link your tables	on page 3-21
Control runtime parameter settings	on page 3-22
Turn off debug mode	on page 3-22
Use transparent objects	on page 3-22
Use fixed sizes for non-graphical objects	on page 3-22
Use variable sizes for graphical objects	on page 3-23
Use image resolution reduction	on page 3-23
Avoid word wrapping	on page 3-23
Simplify formatting attributes	on page 3-23
Limit your use of break groups	on page 3-24
Avoid duplicating work with Graphics Builder	on page 3-24
Choose between PL/SQL and user exits	on page 3-24
Use PL/SQL instead of SRW.DO_SQL for DML	on page 3-25
Evaluate the use of local PL/SQL	on page 3-26
Use multiple attributes when calling SRW.SET_ATTR	on page 3-26
Adjust the ARRAYSIZE parameter	on page 3-26
Adjust the LONGCHUNK parameter	on page 3-26
Adjust the COPIES parameter	on page 3-27
Avoid fetch-ahead in previewing	on page 3-27
Choose appropriate document storage	on page 3-28
Specify path variables for file searching	on page 3-28
Use the multi-tiered server	on page 3-28

Performance Suggestion	Explanation
For any Graphics application: (all the general suggestions also apply to Graphics)	
Pre-load graphics files	on page 3-29
Update displays only if necessary	on page 3-29
Move display updates out of loops	on page 3-29
Use common elements wherever possible	on page 3-29
Limit the DO_SQL procedure to DDL statements	on page 3-29
Use handles to reference objects	on page 3-30
Consider not using shortcut built-ins	on page 3-30
For any application in a client/server environment: (all the general suggestions earlier also apply to client/server)	
Choose an appropriate installation configuration	on page 3-30
Choose the best application residence	on page 3-31
For any application in a 3-tier environment: (all the general suggestions earlier also apply to 3-tier)	
Increase network bandwidth between tiers 1 and 2	on page 3-32
Minimize changes to the runtime user interface	on page 3-32
Adjust stacked canvases (non-visible, raise on entry)	on page 3-32
Perform validation at a higher level	on page 3-32
Avoid enabling and disabling menu items	on page 3-32
Keep display size small	on page 3-33
Identify paths for graphics	on page 3-33
Limit the user of multimedia	on page 3-33
Avoid use of animations driven from the application server	on page 3-33
Take advantage of hyperlinks	on page 3-33
Put code into libraries	on page 3-33
Reduce start-up overhead with JAR files	on page 3-33

Performance Suggestion	Explanation
Reduce start-up overhead with pre-loading	on page 3-34
Use just-in-time compiling	on page 3-34
Increase Tier 2 hardware power	on page 3-34
Use multiple Tier 2 components (Web Application Server)	on page 3-35

3.2 Introduction: What Is Performance?

Before setting out to improve performance, it's helpful to have a clear view of specific goals, and what is involved in achieving them.

You need to be precise about what areas you want to improve, and how performance in those areas is perceived or measured.

Additionally, improving performance can involve understanding the many interrelationships and dependencies in today's computing environment, the costs involved, and the trade-offs that may occur in improving performance in one area.

3.2.1 Performance When?

The use of Forms Developer and Reports Developer is divided into design time and runtime. Design time, when programmers are building the applications, is not usually a concern in terms of performance. It is runtime — when the applications are being exercised by multiple end users in the daily business environment — that is almost always the main concern. As a result, the rest of this chapter focuses on performance in the runtime environment.

3.2.2 Performance of What?

There are many ways to view an application's performance. Its storage requirements, its coding efficiency, its network loading, and its server usage, are just a few areas. Every situation is different, and every site and department will have its own set of priorities and its own view of which performance area is most important. In addition, "good" and "bad" performance in these areas are relative things. There are rarely any absolute standards.

Often the most visible area is performance in terms of response time for end users (that is, how long people using the application must wait after making a choice or entry). Here, too, there are no absolute standards. No matter what the actual response time, users will have an opinion — which will depend in part on what

they are accustomed to and what their expectations are. Real numbers are irrelevant. If end users are not happy with the response time, then that area is certainly a candidate for improvement.

3.2.3 Interrelationships

Applications do not run in a vacuum. In a client/server environment, the application is dependent on two underlying hardware and operating system configurations, plus a hardware and software network connection. In a three-tier environment, the situation is even more complex. In addition, the application is interacting with one or more database servers, and may also be calling other software components as it runs.

Further, an application is often sharing these hardware and software resources with other applications. Because of this sharing, an application that is efficient in itself can be adversely affected by other inefficient applications.

Performance of an application, then, is not just a result of its own design and usages, but a very complex result of the combined interactions of a great number of different components and factors.

3.2.4 Trade-offs

Some improvements in performance are straightforward and purely beneficial. Eliminating useless code in an application would be an example.

Other improvements might not be so clear cut, however. For example, giving one application a higher network priority by necessity lowers the relative priority of the others. As another example, we might restructure a database to improve access time for one type of application, but find that we have actually degraded access time for other important applications.

At the single application level, a classic trade-off is space versus speed. We might be able to decrease our main storage requirements by off-loading some components, but that would most likely degrade the application's response time (since those components would need to be loaded when needed). On the other hand, we might move the loading operations into the start-up phase, which would improve later response time, but at the cost of higher initial start-up overhead.

Before deciding on any particular improvement effort, it's helpful to understand the broader implications, and make choices according to your priorities.

3.3 Measuring Performance

How do we tell if our applications are performing adequately?

In the case of response time, the opinion of our end users is paramount. But in other areas we would like more tangible data; some hard numbers.

3.3.1 Forms Developer- and Reports Developer-Specific Measurements

The Ora_Prof built-in package is distributed with both Forms Developer and Reports Developer. It allows you to examine the PL/SQL in an application, and find out how much time a specific piece of code takes to run.

The following product-specific measurement tools are also available.

3.3.1.1 Forms Measurements

You can obtain general information about a Forms application by setting the runtime option `STATISTICS=YES`.

3.3.1.1.1 PECS

You can use Form Builder's Performance Event Collection Services (PECS) to gather more detailed information about an application's runtime behavior.

You activate the PECS data collection by specifying the runtime option `PECS=ON`.

The simplest use of PECS is to collect application-wide statistics. This does not require any changes to the existing application (only the activation of PECS via the runtime option).

PECS also allows you to focus on specific areas in your application. PECS provides you with a number of built-ins that you can insert into your code to identify sections or events or classes that you want to examine in detail.

Once the data has been collected, you can use the PECS Assistant to view and analyze it. The Assistant produces various types of reports that let you see such things as elapsed time, CPU time, events or occurrences reached, usage of your PL/SQL code, and so forth. Your analysis of the application's runtime behavior can help you spot potential areas for improvement. For example, some section of code might be taking considerably longer to execute than the others, and would therefore be a candidate for closer investigation.

3.3.1.2 Reports Measurements

Report Builder offers two measurement tools: the Reports profile option and the Reports trace option.

3.3.1.2.1 Reports Profile

The Reports profile option, when set, produces a log file that shows where the report spent its processing time. This may help you identify performance bottlenecks.

To set the profile option, specify `PROFILE=<filename>`, where `<filename>` is the name of the required log file. Profile can be either a report parameter or a command line argument.

Typical profile output from a sample report is shown below:

Total Elapsed Time:	29.00 seconds
Reports Time:	24.00 seconds (82.75% of TOTAL)
Oracle Time:	5.00 seconds (17.24% of TOTAL)
UPI:	1.00 seconds
SQL:	4.00 seconds

From this profile, it is possible to see the execution time (total elapsed time) for the report, the amount of time that was spent formatting the retrieved data (Reports Time), and the amount of time spent waiting for the data to be retrieved (Oracle Time). UPI time is the time spent establishing the database connection, and parsing and executing the SQL. The SQL time is the time spent while the database server fetches the data, and time spent executing `SRW.DO_SQL()` statements (the DML and DDL statements that your application may contain).

In this example, the profile shows that the majority of the time was spent laying out the data rather than querying and fetching.

3.3.1.2.2 Reports Trace

The Reports trace option produces a file that describes the series of steps that a report carries out during the execution of the report. The trace option can be set so that all events are logged in the file, or only a subset of steps are logged (for example, only SQL execution steps). The trace file provides an abundance of information that is not only useful for performance tuning, but also in finding out what executed when.

The trace option can be set either from the main menu (choosing Trace under Tools) or from the command line arguments TRACEFILE (filename for trace information), TRACEMODE (either append trace information from future runs to the existing trace file, or replace the trace file), or TRACEOPTS (a list of the event types where tracing is required).

3.3.2 Server- and Network-Specific Measurements

Database servers and network systems often provide measurement and analysis tools that you can use to obtain performance information in those areas.

For example, an invaluable aid to tuning your SQL is the SQL trace functionality provided by the Oracle database server. SQL trace enables you to see the SQL sent to the database, as well as the time taken to parse, execute, and fetch data from the statement. Once a trace file has been generated, use the TKPROF utility to generate an Explain Plan, which is a map of the execution plan used by the Oracle Optimizer. The Explain Plan shows, for example, where full-table scans have been used, which may suggest that the application could benefit from an index (depending on the performance hit). More information about the Explain Plan is available in the *Oracle SQL Language Reference Manual*.

As well as measurement and analysis tools offered by the servers and network systems your application uses, you should also consult with the administrators of those areas. They may be able to offer direct assistance, or suggestions for ways to improve application performance in the existing environment.

3.4 General Guidelines for Performance Improvement

The following performance-improvement guidelines apply to Forms Developer and Reports Developer in general (all their component Builders), and to both deployment architectures (client / server and three-tier).

The general guidelines cover these areas:

- Upgrades of hardware and software
- Data design (data modeling)
- Work sharing between components
- Wait time transfer
- Debug mode

3.4.1 Upgrades of Hardware and Software

Perhaps the simplest way to obtain improved performance is to upgrade your hardware and/or software. While there is effort involved in upgrading, the performance improvements offered by the newer components often make it worthwhile.

3.4.1.1 Software Upgrades

3.4.1.1.1 Upgrading Oracle software

Each successive release of Oracle software offers improvements and extensions over its predecessors. Improvements are in many categories, and vary in nature from release to release. But often a new release will offer not only new functionality, but also something in the way of performance enhancements — perhaps additional tuning aids or even automatic performance improvement.

For example, Release 1.6 improves on Release 1.5 by providing a load balancer that can make efficient use of multiple application servers. Release 2 offers the returned-table-of-records feature, which allows passing changes once to a stored procedure that in turn distributes them to multiple tables, saving network trips. As yet another example, Release 6 contains re-written internal code that uses the more efficient OCI language to interface with the database server, providing improvements without any required customer action.

Consider upgrading to a later, more efficient release.

3.4.1.1.2 Upgrading Other Software Components

Both Forms Developer and Reports Developer, of course, run with other software, most notably the Oracle database server and the PL/SQL language components. Better performance in associated components will often be reflected in better performance in your applications. More tuning features in those areas may offer more opportunity for making improvements in the applications.

The later releases of the associated software almost always offer better performance. For example, the Oracle8 database server offers a number of performance improvements over Oracle7: for example, table and index partitioning, enhanced parallel processing, and deferred constraint checking.

Therefore, to the extent you are able to control or influence the choice of database server and other associated software, consider upgrading to a higher, more efficient level.

3.4.1.2 Hardware Upgrades

Increasing the capacities and/or speeds of the underlying hardware systems is an obvious approach to improving performance. This includes not only the desktop and server machines, but also the network connections between them.

3.4.2 Suggestions for Data Usage

Accessing a database is a major activity of typical Forms Developer and Reports Developer applications. Being efficient in reading and writing that data can have a significant effect on overall performance.

3.4.2.1 Use Array Processing

Both Forms Developer and Reports Developer are able to take advantage of the Oracle database server's array processing capabilities. This allows records to be fetched from the database in batches instead of one at a time, and results in significantly fewer calls to the database. The downside of array processing is that more space is required on the execution platform for storing the arrays of records returned.

If load on the network becomes a major bottleneck in the production environment, then set the Developer product's runtime ARRAYSIZE parameter to as large a value as possible for the execution environment.

3.4.2.2 Eliminate Redundant Queries

Ideally, an application should have no redundant queries (queries which return data which is not required), since they will clearly diminish performance. However, situations can arise where an application not only needs to produce a different format for different users, but also needs to utilize different query statements. Clearly this could be achieved by developing two different applications, but it may be desirable to have a single application for easier maintenance.

For example, in a report, you could disable redundant queries by use of the SRW.SET_MAXROW() procedure. The following code in the Before Report trigger will disable either Query_Emp or Query_Dept, depending on a user parameter:

```
IF :Parameter_1 = 'A' then
    SRW.SET_MAXROW('Query_Emp',0);
ELSE
    SRW.SET_MAXROW('Query_Dept',0);
END IF;
```

There are several points to remember when using SRW.SET_MAXROW():

- The only meaningful place to use SRW.SET_MAXROW( ) is in the Before Report trigger (after the query has been parsed). If SRW.SET_MAXROW( ) is called after this point, then the SRW.MAXROW_UNSET packaged exception is raised.
- The query will still be parsed and bound, but no data will be returned to the report.

3.4.2.3 Improve Your Data Model

If an application is known to be spending an inordinate amount of time in the database, then it is often beneficial to review the structure of the data and how it is being used. Both Forms Developer and Reports Developer are non-procedural tools that are optimized for set-based logic, and a bad schema design can have a dramatic negative effect. For example, an overly normalized data model can result in many avoidable joins or queries, while a lack of appropriate indexes can result in many costly full-table scans.

The specific nature of your application will determine the most efficient data model. A query-driven application can benefit from de-normalized tables; normalized tables are usually best for applications that do many updates and inserts.

Efficient design and operation of the database and server will clearly benefit its client applications. However, because the creation and management of the database is a large topic, and one usually outside the domain of the application developers, the topic of database performance is only introduced here. The subject is covered in its own manual: *Oracle Server Tuning*. Using that manual and such server tools as SQL trace and the TKPROF utility, you can determine where your data model could be improved.

Even if that area is outside your direct control, you might still want to consult with the database server personnel to see if specific performance concerns could be addressed to mutual advantage.

3.4.2.4 Use SQL and PL/SQL Efficiently

Both Forms Developer and Reports Developer use SQL to talk to the database and retrieve data, and it is helpful for anyone tuning applications to have a good working knowledge of SQL and to understand how the database is going to execute these statements.

3.4.2.4.1 Choose an Appropriate Optimizer

Inefficient SQL in your application can severely impact its performance. This is particularly true in applications that have large queries.

The Oracle database server provides you with two SQL optimizers: cost-based and rule-based. Using the cost-based optimizer gives you a significant amount of automatic optimization without having to involve yourself in the complexities of tuning your SQL ; in addition, hints are provided that allow for additional tuning. Using the rule-based (heuristic) optimizer allows you to fine-tune your SQL to potentially achieve an even higher level of optimization, although it does require more work on your part and some understanding of SQL processing.

For most applications, the cost-based optimizer will give a satisfactory level of optimization. Indeed, an untuned cost-based optimization is often superior to a hand-tuned rule-based optimization. However, a developer who understands the spread of the data and the rules governing the optimizer, and who wants to attempt to achieve the highest level of efficiency, can try using the rule-based method.

In any event, it is important to choose one or the other optimizer. Either:

- activate the cost-based optimizer (by either running ANALYZE on the tables or setting the `init.ora` parameter), or
- optimize all your SQL following the suggestions and access path choices provided for you by the rule-based optimizer.

3.4.2.4.2 Perform Calculations within the Query SQL

When performing calculations within an application, the general rule of thumb is that the more calculations that can be performed within the query SQL the better. When calculations are included in the SQL, they are performed by the database before the data is returned, rather than the data being returned and cached before the calculation is performed by the application. From Oracle 7.1 onwards, you can include server-stored user-defined PL/SQL function calls in the query select list. This is more efficient then using a local PL/SQL function (e.g., in a formula column), since the calculated data is returned as part of the result set from the database, so no further calculations are required.

In Oracle8, calls to methods can use the SELF parameter, which simplifies and speeds the passing of arguments.

3.4.2.4.3 Avoid Explicit Cursors

Declaring and using explicit cursors in your application gives you complete control over your database queries. However, such cursors are rarely necessary. (Both Forms Developer and Reports Developer create any needed cursors implicitly, and manage them for you.) Explicit cursors also add to network traffic, and therefore should be avoided in most applications.

3.4.2.5 Use Group Filters

Group filters are available in the Reports and Graphics components.

The main use for group filters is to restrict the number of records being retrieved to be the first or last n records, although there is also an option to create a PL/SQL filter condition. When using a group filter of either type, the query is still passed to the database and all data will still be returned to the application, where the filtering will take place. Therefore, even if the application displays only the top five records, the result set returned will contain all the records returned by the query.

For this reason, it is usually more efficient to try to incorporate the group filter into the where clause of the query wherever possible. This will restrict the data returned by the database.

3.4.2.6 Share Work Between Components

Both Forms Developer and Reports Developer offer the ability to construct applications that use multiple components. For example, data might be fetched and manipulated by Forms, which then calls Reports to produce some output. Reports, too, could call Graphics to display some output visually.

While each called component could re-query the data, it is more efficient to have Forms create a record group to hold the data, and then pass that along as a parameter to Reports, and Reports similarly to Graphics. (This technique is sometimes referred to as query partitioning.) Using this technique, the data is queried only once.

3.4.2.7 Move Wait Time Forward

When one component calls another, and the called component is not already in memory, there is a certain amount time taken to load it. While this load-upon-demand makes more efficient use of memory, it can cause a perceptible wait for the end user.

It is possible to reduce the wait time by having the called component loaded initially (along with the calling component). This does lengthen start-up time (as well as use memory less-efficiently), but a wait at start-up is usually less noticeable than a wait in the middle of processing.

(This technique is more useful for Forms calling Reports than it is for Reports calling Graphics.)

3.4.3 Forms-Specific Suggestions

All the general suggestions offered earlier in this chapter also apply to Forms applications. In addition, consider the following.

3.4.3.1 Tune Your Array Processing

The general value and trade-offs of array processing have already been noted. In Forms, this setting for querying is controlled in the block property Query Array Size. For updating/inserting/deleting, the array processing setting is controlled in the block property DML Array Size.

3.4.3.2 Base Data Blocks on Stored Procedures

If you can, base your data block on a stored procedure.

Stored procedures are the most direct way of moving processing to the server. When correctly designed, stored procedures can also eliminate many network round trips. For example, by basing a query on a stored procedure, the foreign key lookups and calculations can be performed on the server rather than in Post-Query triggers. Such triggers typically add at least one round trip per row, thereby losing the benefit of array fetches.

Similarly, by performing updates through a stored procedure, audit trails or denormalized data can be written without an additional network round trip; so can validations that might be necessary before attempting to perform the DML. This eliminates network round trips that previously might have occurred in Pre-Update, Pre-Insert, and Pre-Delete triggers.

If you are using a release prior to 2.0, you can manually build a data block on a stored procedure by writing transactional triggers, such as On-Select and On-Fetch Using Release 2.0 or later, you can perform array fetches through stored procedures.

You also have two options for queries through stored procedures. The first option is to base a data block's query on a stored procedure that returns a Ref Cursor; the other is a stored procedure that returns a Table of Records.

3.4.3.2.1 Query Based on Ref Cursor

A Ref Cursor is a PL/SQL construct that allows the stored procedure to open a cursor, and return to the client a "pointer" or reference to the cursor. The client can then fetch records from the cursor just as if the client had opened the cursor itself. In the case of Forms, records are fetched through the Ref Cursor using array fetches exactly as if Forms had opened the cursor itself for a data block based directly on a table or view.

A data block based on a Ref Cursor has many similarities to a data block based on a view, but there are two major advantages to a Ref Cursor. First, a stored procedure provides better encapsulation of the data. By denying direct query access to the tables, you can ensure that applications query the data only in ways that are meaningful (for example, a set of tables might be designed to be joined in a specific way to produce a particular set of information) or only in ways that are efficient (for example, queried in such a way that the indexes can be used).

The second advantage is that the stored procedure can be more flexible. The procedure can determine at runtime which one of several Select statements to execute in opening the cursor. This decision might depend on the role or authority of the user. For example, a manager might see all of the columns in the Emp table, but a clerk would be shown blanks for the salary. Or it might depend on a parameter so that a different set of data - historical versus current, for instance - can be displayed in a single data block. This decision can be as complex as you wish, providing you can write the PL/SQL. The only limitations are that all of the different Select statements must return a compatible set of columns and the Select statement cannot be composed dynamically at run time. (The database doesn't yet support Ref Cursors with dynamic SQL).

Note: Use of the REF cursor prevents use of Query-by-Example.

3.4.3.2.2 Query Based on Table of Records

Introduced with PL/SQL release 2.3, a Table of Records is an in-memory structure similar to a database table. The stored procedure can build an in-memory table consisting of, quite literally, any data at all you can construct, row by row, much like an array. Whereas a Ref Cursor allows you to return anything that you know how to construct in SQL, a Table of Records allows you to return anything that you know how to construct in PL/SQL. Not only can you perform lookups and calculations on the server side, you can also make complex decisions about which records to include or exclude from the returned record set.

One example of something relatively easy to do in PL/SQL and very hard to do in SQL would be to return the employees in each department whose salary is in the top 5 salaries for their department. (What makes this hard in SQL is that several people could have the equal fifth high salary. In PL/SQL, it's a relatively simple loop.)

When called in response to a Forms query, the procedure builds the Table of Records on the server side. It then returns the whole result set to the client at once, using as few physical network round trips as the network packet size allows. Each record in the Table becomes a row in the Forms block. This frees up server resources

and uses the network bandwidth very efficiently, at the cost of client resources and potentially wasting network traffic for unneeded records.

Note that when used for a master / detail query, the Table of Records technique will return all the detail records on the query. Thus it is suited only for smaller queries.

In summary then, although a Table of Records allows the procedure the greatest flexibility in determining the result set, it should be used with care.

Note: Use of the REF cursor prevents use of Query-by-Example.

3.4.3.2.3 Insert/Update/Delete Based on Table of Records

In Release 2.0, you can also use a Table of Records returned to a stored procedure to perform inserts, updates and deletes from a block. The stored procedure can then "fan out" your changes to as many tables as necessary, potentially saving many network round trips if your data model is highly normalized. Writing audit trails is another possible use. This technique requires that you provide a procedure for each of Insert, Update and Delete.

As with a block based on a regular table, Forms automatically maintains the state of each record to determine if it is an inserted, updated or deleted record. At commit time, Forms constructs a Table of Records for all of the inserted records, another for the updated records and another for deleted records. It then calls each of the procedures, passing it the relevant Table of Records. As with query, the Table of Records is passed in a "single shot." In this case, though, there is no disadvantage to sending the whole Table at once, since all the records have to be sent to the server to be committed anyway.

3.4.3.2.4 Combining Techniques

Finally, it is worth noting that you might combine these other techniques in any way. For example, you might choose to query through a Ref Cursor while performing DML through a Table of Records, giving you the best of both worlds.

3.4.3.3 Optimize SQL Processing in Transactions

By default, Forms assigns a separate database cursor for each SQL statement that a form executes implicitly or as part of posting or querying data. This behavior enhances processing, because the statements in each cursor need to be parsed only the first time they are executed in a Runform session — not every time.

Forms does allow you to save some memory by having a single cursor for all implicit SQL statements (other than query SELECTs). You would do this by setting the runtime option `OptimizeTP` to No.) However, the memory savings are usually

insignificant, and when you do this, processing is slowed because all Insert, Update, Delete, and Select for Update statements must be parsed every time they are executed.

Therefore, it is recommended that you *avoid* using the `OptimizeTP=NO` setting.

3.4.3.4 Optimize SQL Processing in Triggers

By default, Forms assigns a separate database cursor for each SQL statement that a form executes explicitly in a trigger. This behavior enhances processing, because the statements in each cursor need to be parsed only the first time they are executed in a Runform session — not every time.

Forms also allows you to save some memory by having a single cursor for all SQL statements in triggers. (You would do this by setting the runtime option `OptimizeSQL` to No.) However, the memory savings are usually insignificant, and when you do this, processing is slowed because the SQL statements must be parsed every time they are executed.

Therefore, it is recommended that you *avoid* using the `OptimizeSQL=NO` setting.

3.4.3.5 Control Inter-Form Navigation

It is often more efficient to divide a large application into multiple small forms, and then navigate between the various forms as needed.

You can reduce navigation time if you keep a frequently-used form open after its initial use, rather than opening and closing it each time it is used. Closing and re-opening a form involves considerable overhead, which slows performance.

To keep forms open, navigate by using the OPEN_FORM built-in instead of the NEW_FORM built-in. (NEW_FORM closes the previously-used form when opening the new one.)

3.4.3.6 Raise the Record Group Fetch Size

A larger fetch size reduces the number of fetches required to obtain a record group. If your application is using record groups (or constructing record groups at runtime), set this size using the Record Group Fetch Size property.

3.4.3.7 Use LOBs instead of LONGs

If you are using the Oracle8 server, it is more efficient to use the LOB (large object) datatypes instead of LONG or LONG RAW.

3.4.3.8 Erase Global Variables

Each global variable takes 255 bytes. If an application is creating a large number of these variables, consider erasing them when they are no longer required. (Use the Erase built-in for that purpose.)

3.4.3.9 Reduce Widget Creation on Microsoft Windows

The following suggestions may improve resource usage for very large forms running on Microsoft Windows. (These suggestions differ from standard design practice, and should only be used in those cases where resource usage is a problem.)

- Have some commands available as entries on the Special menu rather than as buttons on the form.
- Change horizontal scrolling areas into alternative regions showing only a subset of the columns at a time.
- Reduce the number of windows by using multiple alternative regions in one window or by improving the window flow.
- Display multiple rows of certain information (primary key and descriptor fields) and one row of less important information by using overflow regions or combination blocks.
- Use the Rendered property whenever possible.
- In general, limit the number of native widgets (non-rendered items) used in the form.

3.4.3.10 Examine the Necessity of Locking

Whenever a user updates a record, Forms locks the record, and a round-trip to the database takes place.

If only a single user is updating the data, the locking is not necessary. To turn off locking, set the Form property Isolation Mode to Serializable; set the block property Locking Mode to Delayed.

However, the suppression of Forms' locking should be done only if you are quite certain that there can never be simultaneous use of the data.

3.4.4 Reports-Specific Suggestions

All the general suggestions offered earlier in this chapter also apply to Reports. In addition, consider the following:

3.4.4.1 Areas to Focus On

Once the data has been retrieved from the database, Reports needs to format the output following the layout model that the user has created. The time taken to generate the layout is dependent on a number of factors, but it is mostly devoted to preventing an object from being overwritten by another object, and performing any calculations or functions in the format triggers. Greater efficiency in these two areas will have the greatest payoff.

3.4.4.2 Reduce Layout Overhead

When generating a default layout, Reports puts a frame around virtually every object to protect it from being overwritten when the report is run. At runtime, every layout object (frames, fields, boilerplate, etc.) is examined to determine the likelihood of that object being overwritten. In some situations (for example, boilerplate text column headings), there is clearly no risk of the objects being overwritten, and hence you can remove the immediately surrounding frame. This reduces the number of objects that Reports has to format, and hence improves performance.

Similarly, when an object is defined as having an undefined size (variable, expanding or contracting in either or both the horizontal and vertical directions) then extra processing is required, because Reports must determine that instance of the object's size before formatting that object and those around it. Where feasible, set this sizing to fixed, which will eliminate this additional processing, since the size and positional relationships between the objects is already known.

3.4.4.3 Use Format Triggers Carefully

It is generally preferable to use declarative format commands rather than format triggers. However, format triggers are useful for making runtime changes. Specifically:

- Disabling and enabling objects dynamically at runtime.
- Dynamically changing the appearance of an object at runtime.

Care should always be exercised when using format triggers, being aware that the trigger does not only fire for every instance of its associated object on the output media, but every time the object is formatted at runtime.

These two purposes noted above may seem like the same thing, but consider the following example. A tabular report includes a single repeating frame that can expand vertically and has page protect set on. As this report is formatted, there is room for one more line at the bottom of the first page. Reports starts to format the

next instance of the repeating frame and fires its associated format trigger. One of the objects inside the repeating frame is found to have expanded, and this instance of the repeating frame is therefore moved to the following page, and the format trigger for the repeating frame is fired again. Hence, although the repeating frame only appears once (at the top of the second page), the format trigger has fired twice. Had this format trigger contained an INSERT statement, then two rows of the same data would have been inserted.

Format triggers should also be placed at the highest level possible in the object/frame hierarchy, so that the trigger fires at the lowest possible frequency. For example, if there are four fields in a frame, a format trigger at the field level will fire four times, whereas a format trigger at the frame level will need to fire only once.

If PL/SQL must be used in a format trigger, place it in the trigger of the object with the lowest frequency possible. For example, PL/SQL in the format trigger of a frame instead of a field typically makes the report run faster. The PL/SQL in a format trigger is executed for each instance of its object. The lower the frequency of the object, the fewer times the PL/SQL will be executed and the faster the report will run.

Because you cannot be sure how many times a format trigger will fire for a particular object, you should not perform calculations or use DML in a format trigger.

If the display attributes of a field are to change dynamically (for example, to draw attention to values outside the norm), then all attribute changes can be set in a single call to `SRW.ATTR()`, or in multiple calls to `SRW.SET` built-ins with each call setting a separate attribute. Although the latter technique makes for more readable code, runtime efficiency is usually better with a single `SRT.ATTR()` call — especially if many attributes need to be set.

3.4.4.4 Consider Linking Tables

As with most operations, there are a number of ways to create data models that include more than one table. Consider, for example, the standard case of the department-employee join; i.e., the requirement is to list all the employees in each department in the company. In Reports Developer, the programmer can either create a single query, or two queries and use a master-detail relationship between the two.

On the application side, when designing the data model it is preferable to minimize the actual number of queries by using fewer, and larger (multi-table) queries rather than more, and simpler (single-table) queries. Each time a query is issued, Reports Developer needs to parse, bind and execute a cursor. A single query is therefore

able to return all the required data in a single cursor rather than many. Also be aware with master-detail queries that the detail query will be re-parsed, re-bound and re-executed for each master record retrieved. In this instance it is often more efficient in a report to merge the two queries and use break groups to create the master-detail effect.

It should be noted, however, that the larger and more complex a query becomes, the more difficult it can be to maintain. Each site needs to decide how to balance its performance and maintenance requirements.

3.4.4.5 Control Your Runtime Parameter Settings

Having designed a report to run efficiently, you can further improve the overall performance of a report by setting specific runtime arguments.

The ARAYSIZE and RUNDEBUG settings have been discussed previously.

If a parameter form or on-line previewing of the report is not required, then you can bypass these functions by setting the PARAMFORM and BATCH system parameters appropriately.

3.4.4.6 Turn Off Debug Mode

When your application is in regular, production use, make sure it is not running in debug mode.

In debug mode, the runtime products will gather information and perform other internal operations. Once your application is debugged, these operations are no longer needed and detract from performance.

In Reports, debug mode is controlled through the runtime parameter RUNDEBUG; this should be set to NO.

3.4.4.7 Use Transparent Objects

Give layout objects (e.g., frames and repeating frames) a transparent border and fill pattern.

Transparent objects do not need to be rendered in a PostScript file. As a result, processing is faster when objects are transparent.

3.4.4.8 Use Fixed Sizes for Non-Graphical Objects

Make your non-graphical layout objects (e.g., boilerplate text or fields with text) fixed in size — that is, Vertical and Horizontal Elasticity of Fixed. In particular, making repeating frames and their contents fixed in size can improve performance.

Non-graphical objects that are variable in size require more processing because Report Builder must determine their size before formatting them. Non-graphical objects that are fixed in size do not require this additional processing because their size is already known.

3.4.4.9 Use Variable Sizes for Graphical Objects

Make your graphical layout objects (e.g., images and Oracle Graphics objects) variable in size — that is, Vertical and Horizontal Elasticity of Variable.

Graphical objects that are fixed in size usually need to have their contents scaled to fit inside of the object. Scaling an object's contents requires more processing. If the object is variable in size, it can grow or shrink with the contents, and scaling is not necessary.

3.4.4.10 Use Image Resolution Reduction

Specify Reduce Image Resolution for image objects whose size you reduce. (This option is available as a drawing option under the Format menu.)

When you reduce the size of an image, it requires less information to display it than when it was larger. Reduce Image Resolution eliminates the unnecessary information and reduces the amount of space needed to store the image. This can be particularly useful for large, multi-colored images.

3.4.4.11 Avoid Word Wrapping

Make fields that contain text one line long and ensure that their contents fit within their specified width (e.g., by using the SUBSTR function).

If a field with text spans more than one line, then Report Builder must use its word-wrapping algorithm to format the field. Ensuring that a field only takes one line to format avoids the additional processing of the word-wrapping algorithm.

3.4.4.12 Simplify Formatting Attributes

Minimize the use of different formatting attributes (e.g., fonts) within the same field or boilerplate text.

If text in a field or boilerplate object contains numerous different formatting attributes, it requires longer to format.

3.4.4.13 Limit Your Use of Break Groups

Ensure that the break order property is set for as few columns in the break group as possible (break order is indicated by a small triangle to the left of the column name in the group). Each break group requires at least one column within in to have break order set.

If sorting is necessary for a break group, use an ORDER BY clause in its SQL. This will cause the rows to be returned already sorted by break order, and improve performance by reducing the amount of sorting that must be done on the client.

For each column that has break order set, Reports places an extra column into the appropriate query's ORDER BY clause. The fewer columns in the ORDER BY, the less work the database server has to do before returning the data. The creation of a break group may make the ORDER BY clause defined in the query redundant. If this is the case, then the redundant ORDER BY should be removed, since this will require extra processing on the database.

Break order columns should be as small as possible, and should also be database columns (as opposed to summary or formula columns) wherever this is feasible. Both of these conditions can help the local caching that Reports does before the data is formatted to be as efficient as possible. Clearly, these conditions can not always be met easily, but are worth considering all the same.

3.4.4.14 Avoid Duplicate Work with Graphics Builder

If a Graphics Builder display referenced by a report uses some or all of the same data as the report, pass the data from the report to the display. You can specify that data be passed in the Property Palette for the display in Report Builder.

If the report and the display use the same data, passing the data reduces the amount of fetching that needs to be done. If you do not pass the data from the report to the display, the data is actually fetched twice: once for the report and once for the display.

3.4.4.15 Choose Between PL/SQL and User Exits

Depending upon the circumstances, PL/SQL or a user exit may perform better. Following are the items you should consider when deciding between PL/SQL and user exits:

- If you need to make many references to Report Builder objects, PL/SQL is typically faster.
- If the report needs to be portable, or if the action is executed on the group or report level, then use PL/SQL.

- If a report does not need to be portable use user exits, instead of PL/SQL, to perform DML.
- User exits are especially beneficial when the action is executed for each record in a large report, or requires a large number of complex computations.

PL/SQL is highly recommended, because it allows the code to be procedural and still portable. PL/SQL also offers performance advantages when referencing report-level objects. User exits will perform better, but they require linking and are not portable. It makes sense to include a user exit if the action required will be executed for every record in a very large report or performs numerous calculations. A PL/SQL procedure makes more sense if the action is only for each group or at the report level. Furthermore, not everything can be done in PL/SQL; some actions like controlling external devices have to be done in a C program.

Note: If there is no performance improvement or other good reason to use a user exit, you should use PL/SQL because it is easier and portable.

3.4.4.16 Use PL/SQL instead of SRW.DO_SQL for DML

Use PL/SQL for DML, unless you want to pass parameters to your DML statements.

`SRW.DO_SQL()` should be used as sparingly as possible, because each call to `SRW.DO_SQL()` necessitates parsing and binding the command and opening a new cursor (just as with a normal query). Unlike the query, however, this operation will occur once each time the object owning the `SRW.DO_SQL()` fires. For example, if a PL/SQL function calls `SRW.DO_SQL()`, and the group where the function resides returns 100 records, then the parse/bind/create cursor operation will occur 100 times. It is therefore advisable to only use `SRW.DO_SQL()` for operations that cannot be performed within normal SQL (for example, to create a temporary table, or any other form of DDL), and to use it in places where it will be executed as few times as possible (for example, in triggers that are only fired once per report).

Writing DML statements in PL/SQL is faster than an `SRW.DO_SQL` call containing the same statement. The reason to use `SRW.DO_SQL` for DML statements is that it can concatenate bind parameters to construct the DML statement. For example, you can have `SRW.DO_SQL` create a table whose name is determined by a parameter entered on the runtime parameter form:

```
SRW.DO_SQL ('CREATE TABLE' || :tname || '(ACOUNT NUMBER
    NOT NULL PRIMARY KEY, COMP NUMBER (10,2))');
```

Usage Notes: You can also use the dbms_sql package that comes with Oracle 7.1 or later for DML. Refer to your Oracle database server documentation for more information.

3.4.4.17 Evaluate the Use of Local PL/SQL

Your PL/SQL code can be local (in the Program Units node of your report in the Object Navigator) or stored externally in a PL/SQL library on the server.

Depending on conditions, local PL/SQL might execute more quickly than a reference to a procedure or function in an external PL/SQL library. However, even if you determine that local PL/SQL would run faster under your conditions, you should still weigh that benefit against the loss of the benefits of the library method (e.g., sharing the code across many applications).

3.4.4.18 Use Multiple Attributes When Calling SRW.SET_ATTR

Minimize the number of calls to `SRW.SET_ATTR` setattr>referenc by specifying multiple attributes in one call. You can specify multiple attributes per call to SRW.SET_ATTR instead of making a separate call for each attribute.

Rationale: The fewer calls you make to `SRW.SET_ATTR`, the faster the PL/SQL will run.

3.4.4.19 Adjust the ARRAYSIZE Parameter

The value of array processing has been noted earlier.

For Report Builder's `ARRAYSIZE` executable argument (e.g., `ARRAYSIZE=10`), enter as large a value as you can. Note that the array size is measured in kilobytes, not rows. `ARRAYSIZE` means that Report Builder can use that number of kilobytes of memory per query in executing your report. Report Builder uses Oracle's array processing, which fetches multiple records in batches, instead of one record at a time. As a result, you can control the amount of data to be fetched by the batch processes.

3.4.4.20 Adjust the LONGCHUNK Parameter

For Report Builder's `LONGCHUNK` executable argument (e.g., `LONGCHUNK=10`), enter as large a value as you can. Refer to the Oracle installation information for your operating system for the recommended amount for your machine. `LONGCHUNK` determines the size of the increments in which Report Builder will retrieve a LONG value. The `LONGCHUNK` size is measured in kilobytes.

By increasing LONGCHUNK as much as possible, you can reduce the number of increments it takes Report Builder to retrieve LONG values.

3.4.4.21 Adjust the COPIES Parameter

When printing to PostScript, specify COPIES=1.

If COPIES is set to something greater than 1 for a PostScript report, Report Builder must save the pages in temporary storage in order to collate them. This can significantly increase the amount of temporary disk space used by Report Builder, and the additional writing to files can slow performance.

3.4.4.22 Avoid Fetch-Aheads in Previewing

Report Builder provides you with the ability to display data such as total number of pages, or grand totals in the report margins or on the header pages. This is an extremely useful function, but has the requirement that the entire report must be processed before the first page can be displayed.

Avoiding "fetch-ahead" operations when designing a report for the Previewer or Live Previewer will help speed the display of the first page of the report.

The following items can result in fetching ahead when referenced before the data on which they rely:

- total number of pages / panels
- grand totals
- break columns that are formulas
- break columns that have Value if Null specified

When you use a total number of pages field source, Report Builder must save all of the pages in temporary storage in order to determine the total number of pages. This can significantly increase the amount of temporary disk space used by Report Builder, and the additional writing to files can slow performance.

Cross-product groups also cause fetching ahead. In order to cross-tabulate the data in a cross-product group, Report Builder must first fetch all of the data. It should be noted that these items are not really performance "problems." They slow down the Previewer or Live Previewer, but they do not affect performance when writing to a file or some other destination.

Note: A column can cause fetching ahead even if it is not displayed. For example, a grand total may not appear in the report output, but, since it is in the report, fetching ahead may still occur when Report Builder calculates it.

3.4.4.23 Choose Appropriate Document Storage

Store documents in files to enhance performance. Store documents in reports and in the database for security. If you open a report from or save a report to a database, some Report Builder tables will be put into memory. As a result, you need to ensure that you have enough resources to cache the tables.

For documents, writing to and reading from files tends to be much faster than the database.

Exception: If you must use a busy network or slow machine to access the files, you may not realize performance gains from storing in files.

3.4.4.24 Specify Path Variables for File Searching

Specifying path variables may be of some help in speeding up file searching and creation / access of temporary files. (Report Builder provides two environment variables, REPORTSnn_PATH and REPORTSnn_TMP, that govern where to search for files and where to store temporary files. The nn is the Report Builder release level.) For REPORTSnn_PATH, specify the path in which files referenced by the report are located. For REPORTSnn_TMP, specify a path that has sufficient free space for temporary files and is on a device with fast response time (e.g., a RAM disk).

REPORTSnn_PATH is the default path in which Report Builder will search for files (e.g., link file boilerplate). By specifying the path in which the files referenced by the report are located, you can reduce the amount of searching Report Builder needs to perform to retrieve the files. (By using REPORTSnn_PATH instead of hard-coding the paths in the report definition, you also maintain the portability of your report.) REPORTSnn_TMP is the path in which Report Builder will create its temporary files.

If using the server, set the SOURCEDIR= parameter on the server-name.ora file. That directory will be searched before using the REPORTSnn path.

3.4.4.25 Use the Multi-Tiered Server

The Multi-Tiered Reports Server is a feature designed to efficiently handle large-scale production reports — reports that are not practical to run on a desktop machine.

With this feature, you can run multiple large reports simultaneously on a robust server machine more appropriate for the task. The server can invoke multiple Reports engines if desired, thus further maximizing efficiency. In addition, report output can be cached on the server, where it can be available to multiple Reports users in the network (so the report need be generated only once).

3.4.5 Graphics-Specific Suggestions

The general suggestions offered earlier in this chapter also apply to Graphics applications. In addition, consider the following:

3.4.5.1 Pre-Load Your Graphics Files

Start-up time for an application that uses graphics will be faster if the OGD graphics files have been pre-loaded. If it is uncertain which specific files will be needed at runtime, a dummy OGD can be created and pre-loaded.

3.4.5.2 Update Displays Only If Necessary

Understand and control the damage update flag — which is one of the arguments to most Graphics PL/SQL built-ins. If you allow the damage flag to default, it will be set to TRUE, which means that redrawing will occur every time the Graphics display list is modified. Such redrawing may not always be necessary.

3.4.5.3 Move Display Updates Out of Loops

Performance is improved if PL/SQL program units (including button procedures, triggers, and so forth) update the display only once. Don't include updates in loops if not necessary.

3.4.5.4 Use Common Elements Wherever Possible

If the Graphics application is called by Forms or Reports, try to design the applications to share as many elements as possible. For example, when charting data already fetched by Forms, pass the same data to the display in record groups (instead of having the display re-query the database).

If all data is being shared and the Graphics application has no need to call the database server, set the LOGON parameter to NO when the Graphics application is invoked. (If LOGON is not set to NO, Graphics will reconnect to the server, slowing down its initiation.)

Also, use the same color palette and same fonts in your form or report and in your display. In addition, keep the same coordinate system, if possible.

3.4.5.5 Limit the DO_SQL Procedure to DDL Statements

The DO_SQL procedure is useful for executing DDL statements. However, do not use this procedure to execute DML statements. In general, DML statements are executed more efficiently within program units than with the DO_SQL procedure.

3.4.5.6 Use Handles to Reference Objects

When you use a built-in subprogram to perform an operation on a Graphics object, you need to identify the object. If you are going to reference an object multiple times in PL/SQL, it is more efficient to assign a handle (that is, a pointer) to the object and identify the object by its handle, rather than to identify the object by its name. Providing the handle reduces internal search time.

3.4.5.7 Consider Not Using Shortcut Built-ins

Graphics provides a series of built-in subprograms that simplify the process of creating objects and getting or setting their attributes. Using these built-ins in place of the attribute record approach reduces development time, and makes program units easier to read and understand.

However, using these built-ins has an adverse effect on runtime performance. Each call to a built-in requires Graphics to define and populate a new internal attribute record. It also takes longer to execute multiple set routines than to execute just one. In addition, using these built-ins requires your application to rely on default settings.

As a rough guideline, if you need to set three or more attributes, it is more efficient to use attribute masks or create a library of your own shortcuts with pre-defined defaults.

3.5 In a Client/Server Structure

In the traditional client/server structure, the application runs on the client, and the database and its software reside on the server. All of the general suggestions offered earlier in this chapter are applicable in a client/server set-up. In addition, consider the following client/server-specific suggestions:

3.5.0.8 Choose the Best Installation Configuration

Both Forms Developer and Reports Developer give you an install-time choice in where their software will reside. Each configuration has assets and drawbacks. Choose the one best suited for your situation.

- Product kept on the server. This saves disk space on the client, but the software will run more slowly because the client must run the software over the network.
- Product kept on the client. All product files are fully replicated on the client machine. This takes up the most client disk space, and adds to installation and maintenance overhead, but provides the fastest execution.

3.5.0.9 Choose a Suitable Application Residence

After you have created an application, you have the choice of storing it on the client or on the server. Storing applications on the server allows shared access to them, and also saves disk space on the clients. On the other hand, applications stored locally on the clients allow faster access.

In addition to the space and sharing considerations, storing on the server may offer the additional advantage of superior security.

Given these considerations, choose the residence best suited to your situation.

3.6 In a Three-Tier Structure

In a three-tier structure, Tier 1 is the runtime user's desktop machine. It runs a Java applet, which loads part of the Forms runtime product, known as the Forms client portion. Tier 2 is an application server, which runs the remaining portion of the Forms runtime product, known as the Forms server portion. Tier 3 is the database server. Communication takes place between the Forms client and Forms server, and also between the Forms server and the database server.

The general performance suggestions offered earlier in this chapter also apply here in the three-tier world. For example, the interaction between the application server component and the database server is essentially the same as that between the application server and the database in the two-tiered client/server environment. Therefore, areas such as improved use of PL/SQL and more efficient use of the database are equally relevant here.

With a three-tier environment, obviously there is communication not just between the application server and the database (Tiers 2 and 3), but also between the client on the desktop machine and the application server (Tiers 1 and 2). Therefore, the reduction of network usage becomes an even more important area on which to focus.

The suggestions below are those that are specific to the three-tier environment.

3.6.1 Maximizing Tier 1 - Tier 2 Scalability

The interactions between the client on the Tier 1 desktop machine and the server on the application server machine become more significant as the number of end users increases. The following suggestions will help you maximize your application's scalability. (These suggestions apply to any Forms Developer or Reports Developer application.)

3.6.1.1 Increase Network Bandwidth

The network connection between Tiers 1 and 2 is often heavily used in the three-tier environment, and therefore network efficiency is an important area for performance. Increasing the bandwidth here can lead to significant improvements.

3.6.1.2 Minimize Changes to the Runtime User Interface

Changes in the user interface during execution require interactions between the Tier 1 and Tier 2 machines. Such changes slow down performance (as experienced by the end user).

You can speed up execution by *avoiding* the following types of runtime activities:

- making dynamic (runtime) visual attribute changes
- using current record visual attributes
- changing item size and position
- changing labels and prompts
- enabling, disabling, hiding objects.

As a general principle, you should limit activities that involve frequent screen refreshing. For example, avoid the use of short-interval visual timers or clocks. (Timers with intervals longer than one minute are usually not a problem.) Design your user interfaces so that events are initiated by user interaction rather than elapsed clock time.

3.6.1.3 Adjust Stacked Canvases

If your application uses stacked canvases, set their Visible property to No, and set their Raise on Entry property to No. This will minimize runtime interface changes.

3.6.1.4 Perform Validation at a Higher Level

Try to perform validation at a higher level. Application design and scalability decisions often involve a trade-off; for example, field-level validation will generate significantly more network traffic than block-level validation, but will be more interactive for users.

3.6.1.5 Avoid Enabling and Disabling Menu items

Enabling and disabling menu items programmatically can reduce performance in Webforms.

3.6.1.6 Keep Display Size Small

If your application uses graphics, limiting the size of the display files will help performance. To help keep the display size small, you can, for example:

- Limit the number of layers in the display.
- Create objects programmatically.
- Take advantage of stored procedures for data-intensive displays.

3.6.1.7 Identify Paths for Graphic URLs

If your application uses graphics (JPG files), use the environment variables `FORMSnn_MAPPING` and `FORMSnn_PATH` to identify their URL location.

3.6.1.8 Limit the Use of Multimedia

Use multimedia only if it is important for the user interface. Where you do use it, define (or redefine) button triggers to make a call to a URL that contains media information.

3.6.1.9 Avoid Use of Animations Driven from the Application Server

Running animations over a network is extremely costly. If such elements are required, look at using animated graphic files that are client-side based.

3.6.1.10 Take Advantage of Hyperlinks

Take advantage of custom hyperlinks to create hyperlink drill-downs. With this technique, code is not loaded to the user machine unless it is actually needed.

3.6.1.11 Put Code into Libraries

Put as much code as possible into libraries to maximize code sharing between objects and applications and to minimize file size during loading.

Beginning with Release 2.0, libraries are shared across multiple forms. This means that program units don't have to be re-loaded and unpacked with each form. It also means that less memory is used, because there is only a single copy of the program unit in memory.

3.6.1.12 Reduce Start-up Overhead with JAR Files

When an application begins running on the desktop machine, it requires the availability of a number of Java class files. In the typical application, there may be a

considerable number of these files, and downloading them from the server adds to start-up overhead.

Beginning with Release 6.0, some of these Java class files are packaged as JAR files. The JAR files can then be stored on the desktop machine instead of on the application server, so application start-up is faster.

You can also place the remaining class files required for your application into JAR files on the desktop machine. This can be done using the Oracle Java Developer Kit.

3.6.1.13 Reduce Start-up Overhead with Pre-Loading

In some situations where fast user interaction is desired, it may be advantageous to pre-load the application; that is, to start it before the actual intensive usage will be needed. In this way, the initial loading phase will already have been completed, and the subsequent invocations will be faster.

3.6.1.14 Use Just-in-Time Compiling

When the application is invoked from the desktop, the user can choose to have it downloaded in an uncompiled state, and compiled on the desktop as it begins running. This option may produce faster overall invocation time.

3.6.2 Maximizing Tier 2 - Tier 3 Scalability

The suggestions for Tier 2 - Tier 3 interaction (interaction between the application server and database server) are the same as for the client/server environment discussed earlier in this chapter. For example, you can use the DML Array Size property, use data blocks based on stored procedures, and so forth. All those earlier suggestions for database interaction apply here as well.

3.6.3 Increase Tier 2 Power — Hardware

Increasing the power of the underlying hardware anywhere in the three-tier system will almost certainly have a positive effect on performance.

Some recent test results suggest that the most significant improvements can be obtained by upgrading the power of the Tier 2 processor. However, each site and situation is unique, and these results may not be universally applicable.

3.6.4 Increase Tier 2 Power — Software

In a three-tier structure, it is possible to have multiple versions of the Tier 2 component. You can employ several intermediate server machines, each running a copy of the Forms or Reports Server component.

You use the Oracle Application Server to coordinate processing. Requests from the client (Tier 1) machines come to the Oracle Application Server, which passes them to one of the Tier 2 servers.

With multiple Tier 2 servers operating and sharing the work load, performance on that tier can be improved.

4

Designing Multilingual Applications

This chapter explains how to design multilingual applications.

Section	Description
Section 4.1, "National Language Support (NLS)"	An overview of National Language Support and its components.
Section 4.2, "Using National Language Support During Development"	Instructions on developing applications using National Language Support.
Section 4.3, "Translating Your Applications"	Instructions on translating application elements not handled by NLS.

4.1 National Language Support (NLS)

Oracle's National Language Support makes it possible to design *multilingual applications*. A multilingual application is an application which can be deployed in several different languages. Oracle supports most European, Middle Eastern, and Asian languages.

National Language Support makes it possible to:

- use international *character sets* (including multi-byte character sets).
- display data according to the appropriate language and territory conventions.
- extract strings that appear in your application's user interface and translate them.

4.1.1 The language environment variables

You can use the following parameters as *language environment variables* to specify language settings:

Parameter	Specifies
NLS_CALENDAR	the calendar system used.
NLS_CREDIT	the string used to indicate a positive monetary value.
NLS_CURRENCY	the local currency symbol.
NLS_DATE_FORMAT	the default format mask used for dates.
NLS_DATE_LANGUAGE	the default language used for dates.
NLS_DEBIT	the string used to indicate a negative monetary value.
NLS_ISO_CURRENCY	the ISO currency symbol.
NLS_LANG	the language settings used by Forms Developer and Reports Developer.
DEVELOPER_NLS_LANG	the language for the Builder.
USER_NLS_LANG	the language for the Runtime component.
NLS_LIST_SEPARATOR	the character used to separate items in a list.
NLS_MONETARY_CHARACTERS	the decimal character and thousands separator for monetary values.
NLS_NUMERIC_CHARACTERS	the decimal character and grouping separator for numeric values.
NLS_SORT	the type of sort used for character data.

4.1.1.1 NLS_LANG

The NLS_LANG language environment variable specifies the language settings used by Forms Developer and Reports Developer.

NLS_LANG specifies:

- the language for messages displayed to the user, such as the "Working..." message
- the default format masks used for DATE and NUMBER datatypes
- the sorting sequence
- the character set

The syntax for NLS_LANG is as follows:

NLS_LANG=language_territory.charset

language specifies the language and its conventions for displaying messages and day and month names. If language is not specified, the value defaults to American.

territory specifies the territory and its conventions for default date format, decimal character used for numbers, currency symbol, and calculation of week and day numbers. If territory is not specified, the value defaults to America.

charset specifies the character set in which data is displayed. This should be a character set that matches your language and platform. This argument also specifies the character set used for displaying messages.

For example, let's say you want your application to run in French. The application will be used in France. Data will be displayed using the WE8ISO8859P1 character set. You would set NLS_LANG as follows:

```
NLS_LANG=French_France.WE8ISO8859P1
```

Now let's say you want your application to run in French, but this time the application will be used in Switzerland. You would set NLS_LANG as follows:

```
NLS_LANG=French_Switzerland.WE8ISO8859P1
```

Note: You are strongly advised to set the language and territory parameters of NLS_LANG to the same values on the server side and the client side. (The value of the charset parameter on the server side is specified when the database is created and cannot be changed.) Use the SQL command ALTER SESSION to change the values of the language and territory parameters on the server side. For example, this statement changes the language parameter to French and the territory parameter to France:

```
ALTER SESSION
    SET NLS_LANGUAGE = French NLS_TERRITORY = France
```

4.1.1.2 DEVELOPER_NLS_LANG and USER_NLS_LANG

If you must use two sets of resource and message files at the same time, two other language environment variables are available:

- DEVELOPER_NLS_LANG specifies the language for the Builder.
- USER_NLS_LANG specifies the language for the Runtime component.

The syntax of DEVELOPER_NLS_LANG and USER_NLS_LANG is the same as NLS_LANG.

Use these variables instead of NLS_LANG in the following situations:

- You prefer to use the Builder in English, but you are developing an application for another language, the two variables allow you to use different language settings for the Builder and Runtime.
- You are creating an application to run in a language that uses a bidirectional character set.
- You are creating an application to run in a language for which a local-language version of the Builder is not currently available.

If these environment variables are not specifically set, they take their default values from NLS_LANG.

4.1.2 Character sets

The character set component of the language environment variable specifies the character set in which data is represented in the user's environment. Net8 ensures that data created using one character set can be correctly processed and displayed on a system that uses a different character set, even though some characters may be represented by different binary values in the different character sets.

Refer to Section 4.1.1, "The language environment variables" for more information on the language environment variables.

4.1.2.1 Character set design considerations

If you are designing a multilingual application, or even a single-language application that will run with multiple character sets, you should determine the character set most widely used at runtime and generate with the language environment variable set to that character set.

If you design and generate an application in one character set and run it in another character set, performance may suffer. Furthermore, if the runtime character set does not contain all the characters in the generate character set, question marks will appear in place of the unrecognized characters.

PDF does not support multi-byte character sets.

Note: For Form Builder, the character set used when generating is used at runtime, regardless of the character set specified in the runtime environment.

4.1.2.2 Font aliasing on Windows platforms

There may be situations where you create an application with a specific font but find that a different font is being used when you run that application. You are most likely to encounter this when using an English font (such as MS Sans Serif or Arial) in a non-Western European environment. This occurs because both Forms Developer and Reports Developer check to see if the character set associated with the font matches the character set specified by the language environment variable. If the two do not match, Forms Developer or Reports Developer automatically substitutes the font with another font whose associated character set matches the character set specified by the language environment variable. This automatic substitution assures that the data being returned from the database gets displayed correctly in the application.

Note: If you enter local characters using an English font, Windows does an implicit association with another font.

There may be cases, however, where you do not want this substitution to take place. You can avoid this substitution by mapping all desired fonts to the WE8ISO8859P1 character set in the font alias file. For example, if you are unable to use the Arial font in your application, add the following line to your font alias file:

```
Arial.....=Arial...WE8ISO8859P1
```

Refer to Section 4.1.1, "The language environment variables" for more information on the language environment variables.

4.1.3 Language and territory

While the character set ensures that the individual characters needed for each language are available, support for national conventions provides correct localized display of data items.

The specified language determines the default conventions for the following characteristics:

- language for server messages
- language for day and month names and their abbreviations (specified in the SQL functions TO_CHAR and TO_DATE)
- symbols for equivalents of AM, PM, AD, and BC
- default sorting sequence for character data when ORDER BY is specified (GROUP BY uses a binary sort, unless ORDER_BY is specified)

- writing direction
- affirmative / negative response strings

For example, if the language is set to French, the messages:

```
ORA-00942: table or view does not exist
FRM-10043: Cannot open file.
```

will appear as:

```
ORA-00942: table ou vue inexistante
FRM-10043: Ouverture de fichier impossible
```

The specified territory determines the conventions for the following default date and numeric formatting characteristics:

- date format
- decimal character and group separator
- local currency symbol
- ISO currency symbol
- week start day
- credit and debit symbol
- ISO week flag
- list separator

For example, if the territory is set to France, numbers will be formatted using a comma as the decimal character.

4.1.4 Bidirectional support

Bidirectional support enables you to design applications in Middle Eastern and North African languages whose natural writing direction is right-to-left. Bidirectional support enables you to control:

- layout direction, which includes displaying items with labels at the right of the item and correct placement of check boxes and radio buttons.
- reading order, which includes right-to-left or left-to-right text direction.
- alignment, which includes switching point-of-origin from upper left to upper right.

- initial keyboard state, which controls whether Local or Roman characters will be produced automatically when the user begins data entry in forms (the end user can override this setting).

When you are designing bidirectional applications, you may wish to use the language environment variables DEVELOPER_NLS_LANG and USER_NLS_LANG rather than NLS_LANG. For example, if you want to use an American interface while developing an Arabic application in a Windows environment, set these environment variables as follows:

```
DEVELOPER_NLS_LANG=AMERICAN_AMERICA.AR8MSWIN1256
USER_NLS_LANG=ARABIC_territory.charset
```

Refer to Section 4.1.1, "The language environment variables" for more information on the language environment variables.

4.1.4.1 Bidirectional support in Form Builder

Four properties are used to specify the appearance of objects in bidirectional applications: Direction, Justification, Reading Order, and Initial Keyboard State.

Direction is an umbrella property that provides as much functionality for each object as possible. For all objects except text items and display items, the Direction property is the only bidirectional property, and its setting controls the other aspects of bidirectional function. (List items, however, include an Initial Keyboard State property.)

Text items and display items do not have a Direction property; instead, you can specifically set Justification, Reading Order, and Initial Keyboard State properties for these items.

You may restrict the keyboard state to one language. For example, setting Keyboard State to Local prevents the end user from switching the keyboard to another language.

When the bidirectional properties are set to Default, those properties inherit their values from the natural writing direction specified by the language environment variable. In most cases, this will provide the desired functionality. You only need to specify the bidirectional properties when you want to override the inherited default values.

Inheritance for bidirectional properties is as follows:

Form	Default setting derives value from language environment variable.
All objects, such as Alert, Block, LOV, Window, and Canvas-view	Default setting derives value from form.
All items, such as Text Item, Display Item, Checkbox, Button, Radio Group, and List Item	Default setting derives value from canvas-view.

Most properties related to bidirectional function can be retrieved and set programmatically. For more information, see the appropriate built-in subprogram description. For example, for information about getting the value of the Direction property for buttons, refer to the description for GET_ITEM_PROPERTY in the Form Builder online help.

4.1.4.2 Bidirectional support in Report Builder

Three properties are used to specify the appearance of objects in bidirectional applications: Justify, Direction (for an object), and Direction (for the report). The bidirectional properties are added to objects in the following hierarchy:

Module

Boilerplate

Field

External Boilerplate

Button

Parameter Form Boilerplate

Objects not in this list either do not require bidirectional support (for example, images) or they are defaulted from one of the above object's properties.

4.1.5 Unicode

Unicode is a global character set that allows multilingual text to be displayed in a single application. This enables multinational corporations to develop a single multilingual application and deploy it worldwide.

Global markets require a character set that:

- allows a single implementation of a product for all languages, yet is simple enough to be implemented everywhere
- contains all major living scripts
- supports multilingual users and organizations
- enables worldwide interchange of data via the Internet

4.1.5.1 Unicode support

Both Forms Developer and Reports Developer provide Unicode support. If you use Unicode, you will be able to display multiple languages, both single-byte languages such as Western European, Eastern European, Bidirectional Middle Eastern, and multi-byte Asian languages such as Chinese, Japanese, and Korean (CJK) in the same application.

Use of a single character set that encompasses all languages eliminates the need to have various character sets for various languages.

For example, to display a multi-byte language such as Japanese, the NLS_LANG environment variable must be set to (for Windows platform):

```
Japan_Japanese.JA16SJIS
```

To display a single-byte language such as German, NLS_LANG must be set to (for Windows platform):

```
German_Germany.WE8ISO8859P1
```

The obvious disadvantage of this scheme is that applications can only display characters from one character set at a time. Mixed character set data is not possible.

With the Unicode character set, you can set the character set portion of NLS_LANG to UTF8 instead of a specific language character set. This allows characters from different languages and character sets to be displayed simultaneously.

For example, to display Japanese and German together on the screen the NLS_LANG variable setting must be:

```
Japan_Japanese.UTF8
```

or

```
German_Germany.UTF8
```

Unicode capability gives the application developer and end user the ability to display multilingual text in a form. This includes text from a database containing Unicode, multilingual text, text in GUI objects (for example button labels), text input from the keyboard, and text from the clipboard. Both Forms Developer and Reports Developer currently support Unicode on Windows NT 4.0 and Windows 95 (limited support).

Note: If you develop applications for the Web, you can use Unicode because of the Unicode support provided by Java.

4.1.5.2 Font support

Both Forms Developer and Reports Developer rely on the Windows operating system for the font and input method for different languages. To enter and display text in a particular language, you must be running a version of Windows that supports that language. Font support is limited but not restricted to the Windows NT operating system font support.

Windows NT 4.0 provides True Type Big Fonts. These fonts contain all the characters necessary to display or print multilingual text. If you try to type, display, or print multilingual text and see unexpected characters, you are probably not using a Big Font. Big Fonts provided by Microsoft under NT 4.0 are as follows: Arial, Courier New, Lucida Console, Lucida Sans Unicode, and Times New Roman. Third-party Unicode fonts are also available.

4.1.5.3 Enabling Unicode support

To enable Unicode support, set NLS_LANG as follows:

```
NLS_LANG=language_territory.UTF8
```

Refer to Section 4.1.1, "The language environment variables" for more information on the language environment variables.

4.2 Using National Language Support During Development

If you wish to use Form Builder, Report Builder, or Graphics Builder in a language other than English, simply specify the correct language and territory in the language environment variable. Messages, menus and menu items, dialog boxes, prompts and hints, and alerts are displayed in the appropriate language and numbers and dates in default values, ranges, and parameters are displayed in the appropriate format. If the appropriate message file is not available, the default is the US message file.

Refer to Section 4.1.1, "The language environment variables" for more information on the language environment variables.

4.2.1 Format masks

4.2.1.1 Format mask design considerations

When working with date and currency fields in multilingual applications, you should make all screen items (boilerplate, text items, interface objects such as buttons and lists of values) longer to allow for translation of text and different ways of displaying data. For example, if you develop an application in American English with a 9-character DD-MON-YY date and then run the application in Norwegian, you must increase the size of the field to allow for the 10-character Norwegian date DD.MM.YYYY.

You should also consider whether you need to use the format mask characters to create special format masks or if the default format masks specified by the territory component of NLS_LANG are acceptable.

For implicit datatype conversions, PL/SQL always expects items in the American_America default format DD-MON-YY, so if you use an item whose type is territory-specific in PL/SQL, you must specify the correct format masks. Use TO_DATE to translate territory-specific items in PL/SQL.

Avoid hard-coding a string containing a month name. If a hard-coded month name is essential, avoid using the COPY built-in. If you use COPY, the month name may be incorrect, depending on which language is specified.

Language-dependent example (not recommended):

```
:emp.hiredate := '30-DEC-97';
copy ('30-DEC-97','emp.hiredate');
```

Language-independent example (recommended):

```
:emp.hiredate := TO_DATE('30-12-1997','DD-MM-YYYY');
```

Refer to Section 4.1.1, "The language environment variables" for more information on the language environment variables.

4.2.1.2 Default format masks

The language environment variable specifies the set of default *format masks* used to display data such as day and month names, numbers, dates, and currency. Specifically, both Forms Developer and Reports Developer use the default format

masks associated with the territory specified in the current language environment variable:

- in the Builder: When the Builder displays default values for items, ranges, or parameters
- at runtime: If a user enters data in a text item whose type is territory-specific, such as DATE or NUMBER

For example, suppose the current territory is America. You create an item of type DATE, and enter a default value of 20-OCT-98. If you then change the territory to Norway, the default value for the item will automatically change to 20.10.1998.

Refer to Section 4.1.1, "The language environment variables" for more information on the language environment variables.

4.2.1.3 Format mask characters

The following format mask characters allow you to override the default format masks.

Character	Returns
D	Digit for the day (1-7)
DY	Name of the day (abbreviated)
DAY	Name of the day (padded with blanks to the length of 9 characters)
WW	Digit for the week, calculated by the algorithm `int((day-jan1)/7)`
IW	Digit of the ISO week
MON	Name of the month (abbreviated)
MONTH	Name of the month (padded with blanks to the length of 9 characters)
RM	Character for the Roman numeral month
I, IY, IYY, IYYY	Last one, two, or three digits of the ISO year or the ISO year, respectively
BC, AD, B.C., A.D.	BC or AD indicator (with or without periods)
AM, PM, A.A., P.M.	AM or PM indicator (with or without periods)

Character	Returns
C	International currency symbol
L	Local currency symbol
D	Decimal separator
G	Group (thousands) separator

4.2.2 Sorting character data

When you are designing multilingual applications, you want to sort character data according to the alphabetic conventions of a particular language rather than according to the characters' binary values. The SQL function NLSSORT makes it possible to do this.

4.2.2.1 Comparing strings in a WHERE clause

Strings in a WHERE clause are compared according to the characters' binary values: one character is considered greater than another if it has a higher binary value in the database character set. However, because the sequence of characters based on their binary values does not match the alphabetic sequence for a particular language, these comparisons yield incorrect results.

For example, suppose you have a column called COL1 that contains the values ABC, ABZ, BCD, and ÄBC. The database character set is ISO 8859/1. You write the following query:

```
SELECT COL1 FROM TAB1 WHERE COL1 > 'B'
```

The query returns BCD and ÄBC since Ä has a higher numeric value than B.

Now suppose you write this query:

```
SELECT COL1 FROM TAB1 WHERE NLSSORT(COL1) > NLSSORT('B')
```

If the language component of the language environment variable is set to German, the query returns BCD, because Ä comes before B in the German alphabet. If the language component of the language environment variable is set to Swedish, the query returns BCD and ÄBC, because Ä comes after Z in the Swedish alphabet.

4.2.2.2 Controlling an ORDER BY clause

If the language component of the language environment variable is set correctly, it is not necessary to use NLSSORT in an ORDER BY clause.

The following query yields a correct result:

```
SELECT ENAME FROM EMP
ORDER BY ENAME
```

4.2.3 NLS parameters

4.2.3.1 Using ALTER SESSION

You can use the SQL command ALTER SESSION to override the NLS defaults. For example, suppose you create some parameters (such as language, territory, etc.), and a user specifies values for them: you could then alter the session as they specified.

In Form Builder, you can specify any of the following NLS parameters for the ALTER SESSION command. However, for Report Builder and Graphics Builder, you can only specify the NLS_SORT parameter.

Parameter	Description
NLS_LANGUAGE	Language used by the server to return messages and errors
NLS_TERRITORY	Territory used for default date and currency masks
NLS_DATE_FORMAT	Default format mask used for dates
NLS_DATE_LANGUAGE	Default language used for dates
NLS_NUMERIC_CHARACTERS	Decimal character and group separator
NLS_ISO_CURRENCY	ISO international currency symbol
NLS_CURRENCY	Local currency symbol
NLS_SORT	Character sort sequence
NLS_CALENDAR	Current calendar system

For example, this statement changes the decimal character to a comma and the group separator to a period:

```
ALTER SESSION
    SET NLS_NUMERIC_CHARACTERS = ',.'
```

These new characters are returned when you use their number format elements D and G:

```
SELECT TO_CHAR(SUM(sal), 'L999G999D99') Total FROM emp

TOTAL
---------------
    FF29.025,00
```

This statement changes the ISO currency symbol to the ISO currency symbol for the territory America:

```
ALTER SESSION
    SET NLS_ISO_CURRENCY = America
```

The ISO currency symbol defined for America is used:

```
SELECT TO_CHAR(SUM(sal), 'C999G999D99') Total FROM emp

TOTAL
---------------
    USD29.025,00
```

This statement changes the local currency symbol to DM:

```
ALTER SESSION
    SET NLS_CURRENCY = 'DM'
```

The new local currency symbol is returned when you use the L number format element:

```
SELECT TO_CHAR(SUM(sal), 'L999G999D99') Total FROM emp

TOTAL
---------------
    DM29.025,00
```

Refer to Section 4.1.1, "The language environment variables" for more information on the language environment variables.

4.2.3.2 Using NLS parameters in SQL functions

Wherever you use SQL, you can use the following NLS parameters to override default NLS behavior.

SQL Function	NLS Parameter
TO_DATE	NLS_DATE_LANGUAGE
	NLS_CALENDAR
TO_NUMBER	NLS_NUMERIC_CHARACTERS
	NLS_CURRENCY
	NLS_ISO_CURRENCY
TO_CHAR	NLS_DATE_LANGUAGE
	NLS_NUMERIC_CHARACTERS
	NLS_CURRENCY
	NLS_ISO_CURRENCY
	NLS_CALENDAR
NLS_UPPER	NLS_SORT
NLS_LOWER	NLS_SORT
NLS_INITCAP	NLS_SORT
NLSSORT	NLS_SORT

4.2.3.3 Form Builder NLS parameters

You can use Form Builder built-in functions to obtain the current value of the language environment variables for use in PL/SQL code:

Environment Variables	DEVELOPER_NLS_LANG (defaults to NLS_LANG)	USER_NLS_LANG (defaults to NLS_LANG)
Built-in	GET_FORM_PROPERTY	GET_APPLICATION_PROPERTY
Parameter	MODULE_NLS_LANG	USER_NLS_LANG

Because both USER_NLS_LANG and DEVELOPER_NLS_LANG default to the value of NLS_LANG, the Form Builder NLS parameters will hold the value of NLS_LANG if either variable is not specifically set.

Both Form Builder NLS parameters have four variations which allow you to retrieve either the complete environment variable or a specific portion of it. This table shows the four parameters of the GET_APPLICATION_PROPERTY built-in that return the USER_NLS_LANG environment variable:

Parameter	Returns
USER_NLS_LANG	Entire USER_NLS_LANG variable
USER_NLS_LANGUAGE	Language portion only
USER_NLS_TERRITORY	Territory portion only
USER_NLS_CHARACTER_SET	Character set portion only

To retrieve the DEVELOPER_NLS_LANG environment variable, call GET_FORM_PROPERTY using the MODULE_NLS_LANG parameter.

Refer to Section 4.1.1, "The language environment variables" for more information on the language environment variables.

4.2.3.4 Report Builder report definition files

When using reports in character mode, you should define the physical page width of a report as one character less than the page width defined in the printer definition file (.PRT file). Otherwise, multi-byte characters might start on the last character space of a line and have to overflow to the next line in order to complete. For example, if the physical page width is 80 characters and the width in the printer definition is 80 characters, a multi-byte character might start on the 80th character. Since multi-byte characters may not be separated, the line would have to overflow to an 81st character in order to complete the multi-byte character. To avoid this, the physical page width should be set to 79 for the report.

The following Arabic and Hebrew specific NLS parameters can be set in the printer definition file:

Use	To	Values
nls locale	set the locale of printing engine.	hebrew or arabic
nls datastorageorder	set logical or visual data storage	logical or visual
nls contextuallayout	perform pre contextual layout for Arabic printers	no or yes

Use	To	Values
nls contextualshaping	perform pre contextual shaping for Arabic printers	no or yes
nls pcharste	specify the character set of the printer	any character set

4.3 Translating Your Applications

In any Forms or Reports application, the user sees the following:

- error messages from the database
- runtime error messages produced by Forms Developer or Reports Developer
- messages and boilerplate text defined as part of the application

If the language environment variable is set correctly and the appropriate message files are available, translation of messages in the first two categories is done for you. To translate messages in the third category, use one of the methods described in the following sections.

Refer to Section 4.1.1, "The language environment variables" for more information on the language environment variables.

4.3.1 Translating your applications using Translation Builder

Translation Builder can help you translate menus, boilerplate text, item labels, messages, and hints defined on item property sheets in your applications. Using Translation Builder, you can generate separate binary files for each language.

If you plan to use Translation Builder to translate your application, develop the application in the following stages:

- Create one basic definition (for example, .FMB) in the source language.
- Use Translation Builder to extract strings for translation, translate the strings into one or more languages, and store the translated strings back into the definition.
- Manually translate messages in PL/SQL libraries. (Refer to Section 4.3.3, "Using PL/SQL libraries for strings in code" for more information on translating messages that are displayed programmatically.)
- Use the Generate component to generate a binary version (for example, .FMX) for each target language.

4.3.1.1 Advantages

This is the simplest way to implement multiple language applications quickly. With this approach, you can use Translation Builder for maximum efficiency. If you have a stable application, this approach will work well for you.

For example, if you create an application in four languages and then change a field label, you would do the following:

- Make the change in the Builder and save the change in the definition file.
- Use Translation Builder to translate the new field label and insert the new messages into the definition file.
- Regenerate to create a binary file containing the new label.

4.3.1.2 Disadvantages

If your applications must support multiple languages simultaneously, you must use the runtime language switching approach instead.

4.3.2 Translating your applications using runtime language switching

A small number of applications must support multiple languages simultaneously. For example, the application may begin by displaying a window in English which must stay up throughout the application, while an end user may press a button on that window to toggle the prompts into French, and then back into English.

If your application requires runtime language switching, you can include more than one language in a single application as long as they share the same character set, but you cannot use Translation Builder to locate translatable text if you are dynamically populating the text at runtime. Instead, you would build case structures (IF...THEN...ELSIF) to change the application to another language by checking the value of the NLS environment variable using the GET_FORM_PROPERTY built-in.

Using the runtime language switching approach, you could develop your application in the following stages:

- Develop the entire application for one language, including libraries.
- Manually translate each library.
- Design boilerplate labels as appropriately sized display items that are dynamically populated at runtime.

Form Builder supports attaching multiple libraries, so you can use one library specifically for messages that will be translated, and other libraries for other purposes.

4.3.2.1 Advantages

The main advantage of this approach is it allows you to support sophisticated applications which may be highly dynamic. In these cases, this approach avoids some maintenance problems, because you do not have to generate separate files for each language each time the application changes.

4.3.2.2 Disadvantages

This approach is more complicated, because it involves considerable effort to create the language-specific message storage, population, and maintenance involved and to perform the translation manually. For example, you would set up a WHEN-NEW-FORM-INSTANCE trigger to set the labels for each button, pulling the correct labels from an attached library, based on the value of the NLS environment variable.

4.3.3 Using PL/SQL libraries for strings in code

While Translation Builder helps you translate strings in your application's user interface, it cannot pull out string constants in PL/SQL triggers and procedures. Manual translation is required for constant text within a PL/SQL block because that text is not clearly delimited, but is often built up from variables and pieces of strings. To translate these strings, you can use PL/SQL libraries to implement a flexible message structure.

Refer to Section 4.3.1, "Translating your applications using Translation Builder" for more information on translating strings in your application's user interface.

You can use the attachable PL/SQL libraries to implement a flexible message function for messages that are displayed programmatically by the built-in routines MESSAGE or CHANGE_ALERT_MESSAGE, or by assigning a message to a display item from a trigger or procedure. The library can be stored on the host and dynamically attached at runtime. At runtime, based on a search path, you can pull in the library attached to the form. For example, a library might hold only the Italian messages:

```
FUNCTION nls_appl_mesg(index_no NUMBER)
RETURN CHAR
IS
   msg CHAR(80);
```

```
BEGIN
   IF      index_no = 1001 THEN
       msg := 'L''impiegato che Voi cercate non esiste...';
   ELSIF   index_no = 1002 THEN
       msg := 'Lo stipendio non puo essere minore di zero.';
   ELSIF   ...
   :
   ELSE
       msg := 'ERRORE: Indice messaggio inesistente.';
   END IF;
   RETURN msg;
END;
```

A routine like this could be used anywhere a character expression would normally be valid. For example, to display an alert with the appropriately translated application message, you might include the following code in your form:

```
Change_Alert_Message('My_Error_Alert', nls_appl_mesg(1001));
n := Show_Alert('My_Error_Alert');
```

To change the application to another language, simply replace the PL/SQL library containing the nls_appl_msg function with a library of the same name containing the nls_appl_mesg function with translated text.

4.3.4 Screen design considerations

When you are designing multilingual applications, remember to leave extra space in the base screen design for widgets and boilerplate labels. To accommodate multiple character sets and allow for expansion caused by translation, a rule-of-thumb is to leave 30% white space around fields, borders, and boilerplate text.

Specifically:

- Prompt on left of field: Allow for 30% expansion to the left of the prompt.
- Prompt above field: Allow for 30% expansion to the right of the prompt.
- Buttons, checkboxes, radio groups, and poplists: Allow for 30% expansion.
- Form titles: Size any bounding box so the title can expand to the right by 30%.
- Display-only fields: Size 30% wider than needed for base language.
- All widgets: Make widgets large enough to accommodate translation. For example, buttons should be large enough to hold translated labels. Check

button height as well as length to be sure the height of the button will accommodate the tallest character you need to use. Calculate pixels needed to render Kanji characters.

5

Designing Portable Applications

With both Forms Developer and Reports Developer, deploying a single application on multiple platforms—Windows, Motif, the Web, even character-mode—is simply a matter of re-compiling. Both Forms Developer and Reports Developer automatically translate the standard control objects (buttons, check boxes, radio buttons, and so on) to the appropriate format for your target platforms. With careful pre-development planning, you can create a single application that satisfies users across environments, providing each with the native look and feel they expect.

This chapter helps you anticipate the issues you will face when developing portable applications and provides suggestions for ensuring that you can move your application across platforms with ease.

Section	Description
Section 5.1, "Before You Begin"	Presents some high-level questions you should answer before developing a portable application.
Section 5.2, "Designing Portable Forms"	Addresses issues of portability with respect to the GUI and the operating system. Also discusses an approach for cross-platform development, as well as considerations unique to the character-mode environment.
Section 5.3, "Designing Portable Reports"	Discusses developing a report for maximum portability.
Section 5.4, "Designing Portable Displays"	Discusses developing a display for maximum portability.

If you're using Project Builder to manage your application, see Section 1.2.4, "Managing projects and project documents across multiple platforms" in Chapter 1.

5.1 Before You Begin

Before you begin designing any application—not just those you intend to port—it's important that you take time to think about the needs you are trying to address with your application. At a minimum, you should have answers to the following questions:

- Which platforms are you supporting? If you plan to deploy on more than one platform, you must consider issues such as fonts, colors, layout, screen size, and screen resolution, to name a few. Section 5.2.1, "Considering the GUI" helps you tackle these issues.
- Is character-mode support required? If so, your options are considerably limited. Consult Section 5.2.4, "Designing forms for character-mode" or Section 5.3.1, "Designing a report for character-mode environments" for some recommendations.
- What displays must you accommodate? Displays can vary a great deal, even on the same deployment platform. See Section 5.2.1.2, "Considering monitors" for a discussion on the limitations monitors can impose.
- Will your application rely on user exits or foreign functions? You'll probably have to rewrite them for each of your target platforms. Section 5.2.2.1, "Including user exits" offers some suggestions and workarounds.

5.2 Designing Portable Forms

Whether you're designing a new form for multiple platforms or preparing an existing form for a new environment, the issues you face fall into the same two key areas, described in the following sections:

- Section 5.2.1, "Considering the GUI"
- Section 5.2.2, "Considering the operating system"

If you've never developed for multiple platforms before, you may also wish to read Section 5.2.3, "Strategies for developing cross-platform forms" for some recommendations on how to approach cross-platform development. If you're developing for character-mode, see Section 5.2.4, "Designing forms for character-mode" for considerations unique to that environment.

5.2.1 Considering the GUI

When developing a portable application, the first thing you must decide is whether the GUI should look the same across all platforms, or if your users expect the

application to inherit the native look-and-feel of their own environment. In most cases, you'll probably opt for the latter approach. However, if users are likely to use the application on multiple platforms, they'll probably want it to look the same on all of them, ignoring local conventions. The only way to determine this is to interview your users, paying close attention to how they work and which tasks they're trying to perform. (See Section 2.1.2, "Defining user requirements" for suggestions on how to determine users' needs.)

Once you've made your decision, the next step is to create an object library for each platform you're supporting. An object library is a set of objects and standards that you create; each object or standard can determine the appearance and layout of an entire frame, window, or region. When housed in an object library, these objects become available to all the developers on your project or site, thus ensuring that even developers working at different locations can produce an application—or different modules within the same application—with a common look and feel.

To fully exploit the power of the object library, it's a good idea to create one library for each of your target platforms. To help you populate your libraries, Form Builder provides the Oracle Applications Object Library, a set of layouts and items that function well in all of Forms Developer's GUI deployment environments (Windows 95, Motif), as well as character-mode. Test these items and objects one by one on each of your platforms. You should be able to add most of the objects to your libraries without modification, although some may need slight adjustment to meet platform-specific requirements.

Section 5.2.3, "Strategies for developing cross-platform forms" provides more details on how to incorporate your object libraries into an overall development strategy.

5.2.1.1 Choosing a coordinate system

For GUI terminals, use the Real Inch, Real Centimeter, or Real Point coordinate systems. These systems allows you to size your objects to the exact shape you want instead of being snapped to the nearest character cell size.

If you're designing for character-mode, use the Character coordinate system and turn on the grid snap. This will ensure that your objects' sizes are in multiples of the character cell size. See Section 5.2.4, "Designing forms for character-mode" for more information on designing character-mode applications.

5.2.1.2 Considering monitors

Even on the same platform, monitors of different sizes and resolutions can greatly impact the usability of your application. For example, while a 6 pt. font on a laptop running Windows 95 is unreadable, the same font on a 17-inch monitor is perfectly

acceptable. The only way to be certain your application is truly portable is to thoroughly test your application on each of the monitors in the deployment environment.

If there are several different sized monitors in your deployment environment, design for the smallest size. Taking the time to find out which monitors your users have—and how many use each size—can help you plan your application more effectively. For example, if your mobile sales force uses laptops for lead tracking and sales management applications, but everyone else uses 17-inch SVGA terminals, you can simplify your task by restricting the window size of only the two critical laptop applications.

Table 5–1 Platform restrictions: Monitors

Platform	Monitor Restrictions
Windows	Size is determined by screen resolution, not by absolute measurement. For example, widgets developed on a 96 dots per inch (dpi) 17-inch monitor appear smaller than the same widgets displayed on a 20-inch 96 dpi monitor, even though the measurement systems appear to be the same. In other words, an inch is not always an inch on Windows.
Motif	Many Motif users are limited to gray-scale monitors, so you can't rely on color for those users.

5.2.1.3 Using color

Restrain your use of color to three or four basic colors that work well together. Colors that are typically available on many platforms include blue, red, magenta, cyan, green and yellow.

Using too many colors can exceed the system's maximum color limit and cause background objects to snap to strange colors, leaving only the foreground color intact. Be sure to test your color combination on all target systems, including monochrome, gray-scale monitors, to make sure they work as expected.

Table 5–2 Platform restrictions: Color

Platform	Color Restrictions
Windows	Widgets can be one of 16 colors defined in the system color palette. If you assign another color, the widget snaps to the closest of the sixteen.
Motif	Many Motif users are limited to gray-scale monitors; do not use color to make important distinctions.

5.2.1.4 Resolving font issues

Fonts play a fundamental role in the user's sense of familiarity and comfort with a GUI system. Table 5–3 lists the recommended font for each GUI platform:

Table 5–3 Platform recommendations: Fonts

Platform	Font
Windows	MS Sans Serif
Motif	Helvetica

When developing a portable application, decide early how you'll use font styles such as boldface, italics, and underlining. (In general, you shouldn't need either underlining or italics; use boldface sparingly, and only for emphasis.) You should also standardize the type size of different display objects. For example, making all labels 10 points will help if you need to translate a font on a different platform.

To meet users' expectations, a ported application must be rendered in the expected font on each platform. To achieve this, you must translate the fonts between platforms using either of these methods:

- Defining aliases for fonts on each deployment platform
- Defining port-specific classes

The next two sections briefly outline these processes.

Note: On Motif, each different size of a given font is considered a separate entity that must be explicitly installed from the font file. For example, suppose you want to port a Windows-based form containing 10, 12, and 28 point Arial fonts to Motif. Rather than simply verifying that Arial has been installed on Motif, you must ensure that each of the desired point sizes—10, 12, and 28—have been installed as well. If Forms Developer can't find the font it needs on the target platform, it substitutes another font using a platform-specific "closest match" algorithm.

5.2.1.4.1 Defining font aliases

Forms Developer provides a font alias file for each platform (`UIFONT.ALI`, in the `ORACLEHOME\TOOLS\COMMON60` directory). In most cases, the file ensures that fonts appear consistently across platforms. However, if you employ custom or non-standard fonts in your applications, some of them may not be recognized on all target platforms. You can tailor the font alias file to define substitutions for the fonts that are not recognized.

Enter each line in the file in this format:

```
source_font = destination_font
```

For each font, you can specify these attributes:

```
<face>.<size>.<style>.<weight>.<width>.<character_set>
```

Example:

When porting from MS Windows to Motif, change all MS Sans Serif fonts to Helvetica:

```
"MS Sans Serif"=Helvetica
```

See the Form Builder online help for more information and examples of font mapping.

5.2.1.4.2 Using classes

When you require greater control over your font aliasing, use classes. For example, suppose you want your poplists and text items to have different fonts on Motif, rather than just imposing a strict conversion of MS Sans Serif to Helvetica. To achieve this:

1. Create two classes, one for poplists and the other for text items.
2. On MS Windows, specify that both classes use MS Sans Serif as the font in `Window.olb`. (See Section 5.2.3.1, "Creating a single source" for information on `Window.olb`.)
3. In Motif.olb, specify that the poplist class uses the Helvetica 9-point font; specify that the text item class uses Helvetica 11-point.

This approach allows you to customize the font used for each class of objects in your application, thus providing a higher level of flexibility.

5.2.1.5 Using icons

Icons are platform-specific. If you use iconic buttons in your application, create a separate icon directory for each platform. Use the same names for the icons on each platform and set the respective environment variable to point to the icon directory. On MS Windows and Motif, this variable is `TK25_ICON`.

If you include icons in your application, keep the following in mind:

- Icons rendered on small monitors (like laptops) can be too small to read.
- Certain icons have special meanings on certain platforms.

5.2.1.6 Using buttons

In MS Windows, a button's moat (the emphatic border around a button to designate a default) is very small compared to that on Motif. Therefore, buttons appear to shrink when run on Motif. On Motif, you can avoid this by modifying the Motif resource file, `Tk2Motif` in `ORACLE_HOME/BIN`. (Oracle uses Motif resource files to control the visual appearance of UNIX-based applications.)

1. Locate the `Tk2Motif` file for your display type:
   - `.gs` (gray scale)
   - `.bw` (black and white)
   - `.rgb` (color)
2. Edit the `Tk2Motif` file and set the `Tk2Motif expandNonDefaultButtons` property to True.

In general, always provide enough space in your Windows buttons to accommodate the larger button size in Motif.

To maximize portability, make all buttons non-navigable. In Windows and Motif, clicking a button means the user actually navigates to the button. Because triggers are often dependent upon button navigation, this difference across platforms can create significant behavioral differences in your application.

Note: Making Windows and Motif buttons non-navigable is an excellent example of the kind of trade-off you might have to make if consistency across platforms is more important than adhering to standard platform behavior.

5.2.1.7 Creating menus

The placement and behavior of menus varies across platforms, as shown in Table 5–4:

Table 5–4 Platform restrictions: Menus

Platform	Menu Restrictions
Windows	Supports Multiple Document Interface (MDI) and Single Document Interface (SDI). In MDI, all windows belonging to an application are contained in a single window, and there is only one menu for the entire application. SDI is similar to Motif in that each window has its own menu.
Motif	Every window has a menu attached. The menu on a parent window may or may not be repeated on child windows.

If you are using a version of Windows that supports MDI and you want your applications to look the same across all platforms, specify in Motif that you do *not* want to repeat the parent window menu on child windows. Then you can design the parent window menu to look exactly like that on MS Windows.

Note: To prevent the screen from flashing when switching between form module windows, combine all the menu options into one single menu application and use the `SET_MENU_ITEM_PROPERTY` built-in to dynamically enable / disable the respective menu items accordingly.

5.2.1.8 Creating the console

Like menus, the placement and behavior of the console also varies across platforms, as shown in Table 5–5:

Table 5–5 Platform restrictions: Console

Platform	Console Restrictions
Windows	Appears at the bottom of the MDI window only.
Motif	Appears on the user-specified window.

To achieve consistency across platforms, place the console on the parent window in your Motif application to emulate the behavior of MDI Windows applications.

5.2.1.9 Miscellaneous

- When building a form for multiple platforms, right-align all prompts. Text often expands when ported to other platforms, and left-aligned prompts can cause fields to shift, creating a ragged margin.
- To provide complex functionality that is completely portable across platforms, employ one or more reusable components in your application. These reusable components are provided in the Demos and Add-ons, to help you build applications upon such powerful features as:
  - Navigator (Explorer) style interface
  - Wizard style interface that mimics the Wizards in Forms Developer and other Windows 95 products
  - Calendar window that automatically displays the calendar according to the NLS settings currently in effect
  - Image and icon files

- Standard menu in the Windows style

Refer to "Reusable Components" (under Forms Developer Demos) in the Form Builder online help for more information.

- Table 5–6 lists other miscellaneous issues related to porting GUIs:

Table 5–6 Platform restrictions: General

Platform	General Restrictions
Windows	A known positioning problem causes two lines forming a right angle on VGA screens to actually overlap on SVGA. Use bevels to avoid this problem.
Motif	(none)

5.2.2 Considering the operating system

No application is truly portable if it depends on functionality unique to a particular operating system. Here are some general rules to keep in mind:

- Avoid port-specific terminology when writing messages. For example, a message like "Press F1 for help" is not portable.
- Do not hardcode path names; path names vary across platforms. Instead, use environment variables to enable Form Builder to find your files during runtime.

  Suppose you need to read an image file called `OPEN.BMP` from your form. In a Windows-only application, you could simply code the path name in the call to `READ_IMAGE_FILE`:

  ```
  Read_Image_File('c:\orawin95\myapp\open.bmp', 'BMP', 'block1.image3');
  ```

  If you want the application to be portable, however, hardcoding won't work, since the name of the path is different on each platform. Instead, you can use an external variable to represent the path name.

  For example, in Windows95 or WindowsNT:

  1. Create a registry entry called `path_var` under the ORACLE key; in UNIX, create a shell variable also named `path_var`.
  2. Use the GETVAR procedure in the TOOL_ENV package to retrieve the value `path_var` using this platform-independent method:

     ```
     path_var varchar2(255);
     ...
     Tool_env.getvar('MYPATH', path_var);
     Read_Image_File(path_var||'open.bmp', 'BMP','block1.image3')'
     ```

The platform-specific path name, represented by the variable `path_var`, is appended to the name of the image file, `OPEN.BMP`. On Window95, `path_var` resolves to the path name `C:\ORAWIN95\MYAPP\`. On UNIX, `path_var` is something like `/oracle_home/myapp/`.

- Anything called through the HOST built-in procedure. Host commands execute port-specific operating system commands. To make your application easier to port:

  1. Create a separate procedure library (`.PLL`) for each platform.
  2. Place all operating system commands in the appropriate procedure library.
  3. Create a generic procedure library.
  4. Rewrite the script file for each platform, ensuring that each script has the same name.
  5. In your form module, make sure all calls refer to the generic procedure library.
  6. Before compiling on a given platform, copy that platform's `.PLL` to the generic procedure library.
  7. Compile.

  Section 5.2.3, "Strategies for developing cross-platform forms" explains how this handling of procedure libraries fits into the recommended development strategy for portable applications.

- Context-sensitive help is not portable. If your application uses native context-sensitive help, replace it with Forms Developer's portable help component, the Online Help Class. This component enables you provide context-sensitive help in your application similar to Windows 95 help.

  The component is built using Form Builder and PL/SQL native capabilities, so it is portable to all Forms-supported platforms. Because the help text you create is stored in the database, it is accessible to all users, and updates are immediately available to everyone.

  To use the Online Help Class in your application:

  1. Install the database objects.
  2. Create the help text for your application.
  3. Attach a PL/SQL library and add code in your key-help trigger to call the help as required.

Refer to the help topic About the Online Help Class" under Forms Developer Demos in the Form Builder online help for step-by-step instructions.

- Avoid including the platform-specific methods listed in Table 5–7.

Table 5–7 Platform-specific methods to avoid

Platform	Method
Windows	■ VBX controls ■ OLE containers ■ ActiveX (OCX) ■ DLLs (ORA_FFI)
Motif	Calls to loadable libraries (ORA_FFI)

These objects leave placeholders (empty spaces) on the platforms that do not support them. If you must include these objects in your application, see Section 5.2.3.4, "Hiding objects" for information on how to prevent the placeholders from appearing.

5.2.2.1 Including user exits

A *user exit* is a 3GL program you write yourself and then link into a form at compile time. User exits are always port-specific.

Before calling a 3GL program from your portable form, verify that the information and processes on which the program relies are available on all platforms. For example, a program that depends on information from the Windows registry can't access this information on other platforms, which means you may have to re-design the program or abandon it entirely.

Rather than accessing a 3GL program through the user interface exit, consider the use of the ORA_FFI built-in package (Oracle Foreign Function Interface). If you use the user exit interface to access your foreign functions, you must re-link the user exits or replace the DLL for each platform each time a 3GL program changes. Because ORA_FFI allows you to call foreign functions through a PL/SQL interface using PL/SQL language conventions, re-linking isn't required when you modify a program. For this reason, ORA_FFI is the preferred method for accessing 3GL programs from your forms.

5.2.3 Strategies for developing cross-platform forms

This section introduces some techniques you can use to develop portable forms:

- Section 5.2.3.1, "Creating a single source" describes an architecture for creating a single source that delivers maximum functionality on each of your deployment platforms.
- Section 5.2.3.2, "Subclassing visual attributes" discusses the importance of explicitly subclassing the visual attributes stored in your object libraries.
- Section 5.2.3.3, "Using the get_application_property built-in" discusses the use of this Form Builder built-in when developing portable applications.
- Section 5.2.3.4, "Hiding objects" provides sample code for removing the placeholders that appear when an object is not valid on a particular platform.

5.2.3.1 Creating a single source

While it may be tempting to consider creating a single source that aims at the lowest common denominator for all deployment platforms, this strategy severely limits the aesthetics you can provide in your application. A more effective strategy is to create a single source that delivers applications in each platform's native look-and-feel. The architecture depicted in illustrates how you might accomplish this:

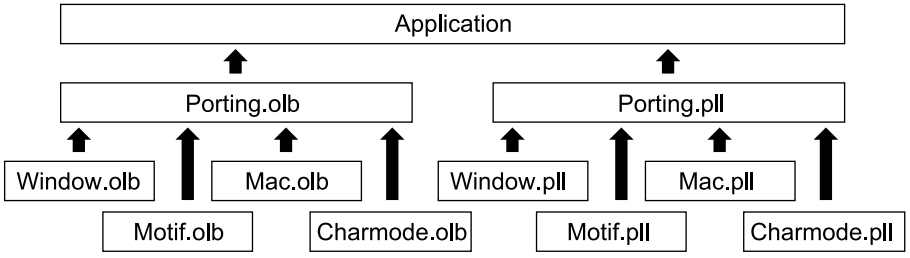


Figure 5–1 Port-specific implementation

To model your application on this architecture:

1. Create an object library for all standards and objects (`Porting.olb`). Refer to Section 2.2.2.1, "Using object libraries" in Chapter 2 for information on using object libraries.
2. Create separate object libraries for each deployment platform (`Window.olb`, `Motif.olb`, `Mac.olb`, `Charmode.olb`).
3. Create a common library for port-specific code (`Porting.pll`).

4. Create separate libraries for each platform's port-specific code (`Window.pll, Motif.pll, Mac.pll, Charmode.pll`).
5. In each platform's UI repository (`.OLB`) and library (`.PLL`), develop code to handle the application objects in the manner ideal for that particular platform. Use the same name for a given object in each UI repository and library.
6. Write your application, referring to the standards and objects in the repository and to the port-specific code in the library.
7. When you're ready to compile your application for a particular platform, copy that UI's repository and library to `Porting.olb` and `Porting.pll` and compile.

5.2.3.2 Subclassing visual attributes

Visual attributes are the font, color, and pattern properties you set for form and menu objects. By carefully defining the visual attributes of your form objects, you can ensure that users on each platform enjoy the native look-and-feel unique to that environment.

Many Form Builder objects, such as items and canvases, refer to visual attributes to define their appearance. Visual attributes must be defined in the same module as the object that refers to them.

Visual attributes are usually stored in an object library. It's a good idea to create subclasses of these visual attributes in each module. When you subclass an object from an object library in your form, any changes made to the library object are automatically applied to the form object. However, this does not apply to changes made to the library object's visual attributes. So, by subclassing, rather than coping the visual attributes, you ensure that your modules always reflect the latest definition of the visual attributes.

5.2.3.3 Using the get_application_property built-in

The `GET_APPLICATION_PROPERTY` built-in function returns information about your application, allowing you to react dynamically at runtime based on the settings of one or more of these variables:

- DISPLAY_HEIGHT and DISPLAY_WIDTH. Specifies how big the current display is. The unit depends on how you have set up the form coordinate system.

- OPERATING_SYSTEM. Specifies the name of the platform on which the application currently is running (MSWINDOWS, WIN32COMMON, SunOS, VMS, UNIX, or HP-UX).
- USER_INTERFACE: Specifies the name of the user interface technology on which the application is currently running (WEB, MOTIF, MSWINDOWS, MSWINDOWS32, PM, X, VARCHAR2MODE, BLOCKMODE, or UNKNOWN).

Depending on the value of a variable, you can dynamically hide objects that are not available on that deployment platform, reposition other objects to take up that space and, if necessary, alter the attributes of an object to suit the standards on that deployment platform. See Section 5.2.3.4, "Hiding objects" for more information.

5.2.3.4 Hiding objects

To prevent users from seeing placeholders on platforms that do not support OLE, VBX, and ActiveX objects, you can put these objects in a separate window invoked from the menu or a button and dynamically enable/disable the menu item or button. Or you can use this code fragment to hide/show the port-specific objects and reposition other objects to take their place:

```
WHEN-NEW-FORM-INSTANCE trigger:
declare
ui  varchar2(15) ;
begin
ui := get_application_property (user_interface);
if ui = 'CHARMODE' or ui = 'MOTIF' then
   set_item_property ('VBXOBJECT1', displayed, property_false);
   set_item_property ('OLEOBJECT1', displayed, property_false);
   set_item_property ('TEXTITEM1', position, 43, 4);
end if;
end;
```

Note: Item prompts are automatically hidden when you hide the associated item.

5.2.4 Designing forms for character-mode

If you are creating an application for both character-mode and bit-mapped environments, single-sourcing is probably not the best approach. Developing for the lowest common denominator, character-mode, deprives your GUI users of the ease

of use associated with bit-mapped controls. The "Bit-map Only" column in Table 5–8 lists the functions you'd have to avoid:

Table 5–8 Character-mode vs. bit-mapped environments

Character Mode	Bit-map Only
■ Boxes	■ Images
■ Horizontal lines	■ Color
■ Vertical lines	■ Drawings
■ ASCII text	■ Ellipses
■ Boldface text	■ Drill-down buttons (reports)
■ Underlines	■ Italicized text
	■ Bit-map patterns
	■ Diagonal lines
	■ Multimedia support

While there are methods for disabling these GUI functions in a character-mode environment, this task can be extremely time-consuming and frustrating for you. So, if you know from the beginning that you have to support these two widely disparate sets of users, it's best for everyone—you and your users—to simply create two entirely separate applications.

It's much easier to develop for character-mode if you make Form Builder **look** like character-mode as much as possible. Table 5–9 lists some recommended property settings:

Table 5–9 Property settings that resemble character-mode

Property	Recommendations/Notes
Boilerplate font	■ Windows: FixedSys, Regular, 9 pt ■ Motif: Font=Fixed, Size=12.0, Weight=Medium, Style=Plain
Coordinate information	■ Coordinate system: Real[1] ■ Real Unit: Point ■ Character Cell Width: 6 ■ Character Cell Height: 14

Table 5–9 Property settings that resemble character-mode

Property	Recommendations/Notes
View	■ Grid: on ■ Grid Snap: on ■ Show Canvas: off
View→Settings→Ruler	■ Units: Character cells ■ Character Cell Size Horizontal: 6 ■ Character Cell Size Vertical: 14 ■ Grid Spacing: 1 ■ Snap Points per Grid Spacing: 2

[1] Improves portability of the form from character-mode to bit-mapped environments. If the form will be deployed in character-mode only, use the Character coordinate system.

As you develop your application strictly for character-mode, keep the following in mind:

Table 5–10 Recommendations for character-mode applications

Topic	Recommendations/Notes
General	■ Remember that everything is in monospace font. ■ Create keyboard equivalents for each widget, even when the widget does not have the current focus. ■ Avoid scrolling, as it is very hard to use. ■ Hide OLE, VBX, and ActiveX objects if you do not want users to see their placeholders. ■ Be sure that widgets have sufficient space to display themselves entirely, as all UI widgets are rendered in their character-mode equivalents. ■ Because users cannot move an LOV with a mouse, use the `set_lov_property` built-in to dynamically position the LOV.

Table 5–10 Recommendations for character-mode applications

Topic	Recommendations/Notes
Navigation	■ Since the user does not have a mouse, users cannot navigate between windows or forms from within the application. Provide buttons or menu options for navigating between forms. ■ Since windows cannot be repositioned with a mouse, ensure that a displayed window does not obscure the context required for that window. When the user is done with a window, disable the window programmatically, or set the window's Remove On Exit property to true.
Layout	■ There are only 80x24 character cells on the screen. The first line is used for the menu; the last two at the bottom for the console display and the message and status lines. Plan your screens carefully to fully utilize the remaining space. ■ Fonts are monospaced and thus consume much more space on average than proportional fonts. Design your screens so that boilerplate and textual widgets can be rendered with one character per cell.
Coordinate system	■ Use the Character coordinate system and turn on the grid snap. This will ensure that your objects' sizes are in multiples of the character cell size.
Menus	■ Menus are displayed on the first line of the screen. ■ Common menu items like Cut, Copy, and Paste not available. ■ Define hot keys for commonly used menu items to reduce cumbersome navigation to the first line of the screen.
Bevels	■ Not available.
Buttons	■ Avoid use of button palettes; make all actions available from the menu instead. Because character-mode does not allow the user to retain context when navigating to a button, buttons do not work well in this mode.
Icons	■ Not available. Make sure that all iconic buttons in the GUI environment are also represented by menu options. ■ Iconic buttons display with just the buttons' labels. Make sure the labels are meaningful and that there is sufficient space to display them.

Table 5–10 Recommendations for character-mode applications

Topic	Recommendations/Notes
Color	■ Precede negative numbers with a minus sign since color is not available. ■ On monochrome displays, colors snap to black or white. Avoid using dark colors for both background and foreground, as both are snapped to black.

5.3 Designing Portable Reports

When preparing a report to run on multiple platforms, consider the following:

- **Fonts**. Not all font types, styles, and sizes are available on all target GUIs. You can handle this in one of two ways:
  - Use a font that you know exists on the target GUI or one that maps well to the default font of the target GUI.
  - Modify the font mapping file, `UIFONT.ALI`, to ensure that the fonts map correctly. See Section 5.2.1.4.1, "Defining font aliases" for more details on using the `UIFONT.ALI` file.

  Note: Because screen font and printer font metrics are not always the same, your printed report may not look the same as it did on the screen. In particular, text fields can expand on the printed page, causing adjacent fields to shift and possibly creating new and unpredictable page breaks. To avoid this, use expand-only fields and be sure each field is large enough to accommodate the largest font reasonably possible.
- **Colors.** If possible, use a color that you know exists on the target GUI; otherwise, use one that maps well to the default color of the target GUI. The following colors are typically available on many platforms: blue, magenta, red, cyan, green, yellow. See Section 5.2.1.3, "Using color" for some recommendations on including color in portable reports.
- **DPI**. The dots-per-inch (DPI) that your monitor uses may not be the same as the DPI used by the person who runs the report. The DPI only affects how alpha-numeric characters word-wrap on the screen. If you design a report that may be displayed in the Previewer view, try to use the same DPI as the people who will run it.
- **Buttons**. If you provide buttons in a report, users viewing the report through the Previewer can press the buttons to display a multimedia object (sound, video, image) or to perform an action through PL/SQL code, such as drilling

down to another report. See Section 5.2.1.6, "Using buttons" for some guidelines on creating portable buttons. Note that if you run a report containing buttons in character-mode, the buttons are simply ignored; they do not create a placeholder.

5.3.1 Designing a report for character-mode environments

Character-mode reports are often needed in environments where users need to send their report output to bulletin boards, spreadsheets, dump files, or to character-only printers. Character-mode output also provides a number of advantages:

- **Portability.** Because they are strictly ASCII or EBCDIC files, character-mode reports can be printed or exported anywhere.
- **Protecting printer investment.** Character-mode reports require no special formatting—unlike complicated postscript output—thus protecting your investment in older printers.
- **Printer code support.** Reports Developer provides support for printer escape codes, which enable users to exploit printer-specific features at runtime, such as special font sizes, highlighting, and more. Refer to the Report Builder online help for information on printer definition files and printer codes.
- **Performance.** Character-mode reports run much faster than an equivalent bit-mapped report. Bit-mapped reports typically require more formatting time and have larger (Postscript, PCL5) output files.

5.3.1.1 Design considerations

Reports built for bit-mapped environments cannot easily be adapted to character-mode. If you know you will need to run a report in a character-mode environment, it is best to build the report as a character-mode report. However, if you must convert a bit-mapped report to character mode, refer to the Report Builder online help, index entry: ASCII report, creating. You will also find step-by-step instructions there for building a character-mode report.

5.4 Designing Portable Displays

If you have standalone graphics—graphics that are not part of a container application such as a form or report—porting is fairly straightforward. Most graphics, however, are embedded within forms and reports, which can introduce problems when moving across platforms. When developing graphics for multiple environments, observe the following guidelines:

- To ensure that text scales uniformly—especially when the graphic is embedded in a form or report—use a scalable truetype font and set the Scalable Fonts flag to true for all text objects **except chart labels**. The Scalable Fonts flag is not available for text labels in a chart. As soon as a chart is updated, fonts are re-set to their original size. Thus, choose a font and size for chart labels that is legible at the greatest range of chart sizes. A good bet is small, sharp fonts that display well at 8-10 point sizes. Anything larger may cause your chart to become unreadable when embedded in a small chart area of a form or report.
- Timers and drag-and-drop code are supported only in standalone Graphics applications. If you include these functions in a form or report, they are ignored.
- Limit your use of colors to the core 16, which are available in the Designer (upper-left corner of the palette), as well as through their mnemonic names (red, green, blue, yellow, magenta, cyan, black, white, gray, darkgray, darkyellow, darkcyan, darkmagenta, darkblue, darkgreen, and darkred).
- Set colors through the layout editor, rather than through PL/SQL. Colors chosen in the layout editor are automatically adjusted to the nearest available color. Colors set through code can result in an error if the color is not available on your system at the current resolution.
- Isolate platform-dependent code with calls to the application property `og_get_ap_platform`, and to the built-in subprograms `og_append_directory` and `og_append_file`. Refer to the Graphics Builder online help for more information.

6

Taking Advantage of Open Architecture

This chapter offers guidelines to help you take advantage of the open and extensible development environment available in both Forms Developer and Reports Developer.

Section	Description
Section 6.1, "Working with OLE Objects and ActiveX Controls"	Describes support for component technologies and provides steps and guidelines for creating applications that include OLE objects and ActiveX controls.
Section 6.2, "Using Foreign Functions to Customize Your Applications"	Describes how to customize and extend your applications with 3GL foreign functions.
Section 6.3, "Using the Open API to Build and Modify Form Builder Applications"	Introduces the Open API and explains how to use the Open API to build and modify Form Builder applications.
Section 6.4, "Designing Applications to Run against ODBC Datasources"	Discusses ODBC support and provides detailed steps and guidelines that describe how to run applications against ODBC datasources.

6.1 Working with OLE Objects and ActiveX Controls

This section describes what OLE and ActiveX are, and how you can exploit this technology. This section includes these topics:

- Section 6.1.1, "What is OLE?"
  - Section 6.1.1.1, "When should I use OLE?"
  - Section 6.1.1.9, "Adding an OLE object to your application"
  - Section 6.1.1.10, "Manipulating OLE objects"
  - Section 6.1.1.11, "OLE examples"
- Section 6.1.2, "What are ActiveX controls?"
  - Section 6.1.2.1, "When should I use ActiveX controls?"
  - Section 6.1.2.2, "Manipulating ActiveX controls"
  - Section 6.1.2.7, "Adding an ActiveX control to your application"
  - Section 6.1.2.8, "ActiveX examples"

Note: Support for OLE and ActiveX is limited to the Windows platform.

6.1.1 What is OLE?

Object Linking and Embedding (OLE) is a Microsoft standard that allows you to integrate and reuse different software components within a single application.

Integrating an application with a Microsoft Excel document, for example, enables you to offer both Forms Developer (or Reports Developer) and Microsoft Excel features. Your users can format a Microsoft Excel document with any of the text processing features provided by Microsoft Excel, while using Forms Developer or Reports Developer features for displaying and manipulating data from the database.

By incorporating OLE objects within your application, you can seamlessly integrate a diverse group of specialized components to build full-fledged applications. You no longer have to build entire applications from the ground up. You can deliver applications in a shorter amount of time and at a lower cost.

6.1.1.1 When should I use OLE?

Use OLE when:

- You want to leverage an existing OLE-compliant application within your application.

  For example, you can enhance your application's capabilities with word processor documents, spreadsheet documents, knob controls, video clips, sound, and so on.

- You want to provide your application users with a familiar interface.

  On Microsoft Windows, most users are familiar with Microsoft Word and Microsoft Excel. Rather than creating word processing or spreadsheet functionality to your application, you could leverage and embed a Word or Excel document within your application.

- Your applications are primarily deployed on the Windows platform.

6.1.1.2 About OLE servers and containers

OLE uses the concept of client and server. The client is an application that requests and uses the services of another application. The server is the one that provides these services.

- **OLE Server Application**

  An OLE server application creates OLE objects that are embedded or linked in OLE containers. The server application is responsible for the creation, storage, and manipulation of OLE objects. For example, the server decides how to repaint the object when certain portions are exposed.

  Graphics Builder and Microsoft Word are examples of OLE servers.

- **OLE Container Application**

  Unlike OLE server applications, OLE container applications do not create documents for embedding and linking. Instead, OLE container applications provide a place to store and display objects that are created by OLE server applications.

  Form Builder and Report Builder are examples of OLE container applications.

6.1.1.3 About embedded and linked objects

You can link or embed OLE objects within your application.

- **Embedded Object.** An embedded object has both its presentation and native data stored *within* your application, or as a LONG RAW column in the database.

- **Linked Object.** A linked object only contains presentation information and a reference to its native data. The content of the linked object is not stored within your application or as a LONG RAW column in a database; it is stored in a separate, linked file.

There is no functional difference between linking and embedding. The OLE container treats the objects equally, by executing the same code, whether they are linked or embedded. The only difference is that embedding an OLE object increases the size of your application. This could eventually lead to performance considerations (particularly on a file server), because the larger the application, the longer it will take to open and load into memory.

6.1.1.4 About the registration database

Each client machine contains an OLE registration database. The registration database stores a set of classes that categorize OLE objects. The information in the registration database determines the object classes that are available for embedding and linking in OLE containers.

OLE server applications export a set of classes that become members of the registration database. Each computer has a single registration database. If the registration database does not already exist when an OLE server application is installed, one is created.

A single OLE server application can add many OLE classes to the registration database. The process of adding classes to the registration database is transparent and occurs during the installation of an OLE server application. For example, when you install Microsoft Excel, several classes are added to the registration database; some of the classes that are installed in the registration database include Excel Application, Excel Application 5, Excel Chart, Excel Sheet, ExcelMacrosheet, and ExcelWorkSheet.

6.1.1.5 About OLE activation styles

Activating an OLE object enables you to access features from the OLE server application. There are two ways to activate an OLE object: *in-place activation* or *external activation*.

- **In-place Activation.** In-place activation enables your users to manipulate the OLE object *within* your application without switching to a different window.

  During in-place activation, the activated object appears within a hatched border, and the toolbar, menu and other controls of the activated object temporarily replace standard menu options. The replacement menu options and toolbars provide access to features that are available from the OLE server

application. Standard menu options and toolbars re-appear when you deactivate in-place activation. To deactivate in-place activation, you click anywhere outside the hatched border.

Note: In-place activation is available for embedded objects, but it is not available for linked objects.

- **External Activation.** External activation enables your users to manipulate the OLE object in a *separate* window. When an OLE object is activated, the object's OLE server application is launched, and the OLE object appears in a separate OLE server application window. The separate window has the menu options and toolbars of the OLE server application. To deactivate external activation, you must explicitly exit the OLE server application.

  External activation is available for both embedded and linked objects.

  When the contents of a linked source file is modified with external activation, a linked object can be updated manually or automatically. Manual updates require an explicit instruction for an object to reflect changes from a linked source file. Automatic updates occur as soon as you modify a linked source file.

  Note: Both in-place activation and external activation are dependent on the OLE activation property settings of the OLE container. If the OLE server application is accessible, the activation property settings of the OLE container determine whether in-place activation or external activation occurs when an embedded OLE object is activated. Linked objects can only be activated with external activation; in-place activation does not apply to linked objects, even if the in-place activation property is set to Yes.

6.1.1.6 About OLE automation

Occasionally, you may want to interact with or manipulate the data within an OLE object. To do so, you use PL/SQL and OLE automation.

OLE automation enables the server application to expose a set of commands and functions that can be invoked from an OLE container application. By using these commands and functions, you can manipulate OLE objects from the OLE container environment.

In both Forms Developer and Reports Developer, you use PL/SQL to access any command or function that is exposed by an OLE server application. Built-ins provide a PL/SQL Application Programming Interface for creating, manipulating, and accessing OLE commands and functions.

Note: Many of the options available for manipulating an OLE object in an OLE container application are determined by the OLE server application. For instance, options from the OLE popup menu, also known as OLE verbs, are exposed by the OLE server application. The information contained in the registration database, such as object classes, is also dependent on the OLE server application.

6.1.1.7 OLE support

Both Forms Developer and Reports Developer provide OLE server and container support as well as support for OLE automation.

Component	Container	Server Application	OLE2 Automation
Form Builder	Yes	No	Yes
Graphics Builder	No	Yes	Yes
Procedure Builder	No	No	No
Project Builder	No	No	No
Query Builder	No	No	No
Report Builder	Yes	No	Yes
Schema Builder	No	No	No
Translation Builder	No	No	No

6.1.1.7.1 OLE container support

As OLE container applications, Form Builder and Report Builder support the following:

- Embedding and linking of OLE server objects into OLE containers.
- In-place activation of embedded contents in OLE containers (Form Builder only).

  In-place activation enables you to access menus and toolbars from OLE server applications to edit embedded OLE objects while you are in Form Builder.
- Programmatic access to OLE objects, properties, and methods through OLE automation support from PL/SQL.

  Using PL/SQL, you can invoke commands and functions that are exposed by OLE servers.
- Seamless storage of OLE objects in a database in LONG RAW columns.

You can save OLE objects to a database, as well as query OLE objects from a database. When linked objects are saved, only the image and the link information are retained in the database. The contents of a linked object remains in a linked source file. Saving an embedded object retains all the contents of an embedded object in the database.

6.1.1.7.2 OLE server support

Graphics Builder is an OLE server application. You can embed or link Graphics Builder displays within your Forms Developer or Reports Developer application.

Recommendation: If you want to add a Graphics Builder display to your application, don't embed or link it as an OLE object. Instead, use the Chart Wizard to add graphical displays to your applications.

6.1.1.7.3 OLE container properties

OLE container properties determine OLE display attributes, OLE container interaction with the server, container storage, and so on.

Note: In addition to container properties, you can also set OLE object properties. Each OLE object can expose several properties. You access OLE object properties by clicking the right mouse button to display the popup menu.

This section lists the OLE container properties supported by both Forms Developer and Reports Developer.

Component	Property	Description
Form Builder	■ OLE Activation Style	Specifies the event that will activate the OLE containing item, either double-click, focus-in, or manual.
	■ OLE Class	Determines what class of OLE objects can reside in an OLE container.
	■ OLE In-place Activation	Specifies if OLE in-place activation is used for editing embedded OLE objects.

Component	Property	Description
	■ OLE Inside-Out Support	Specifies if the OLE server of the embedded object enables inside-out object support during in-place activation. Inside-out activation enables for more than one embedded object to have an active editing window within an OLE container.
	■ OLE Popup Menu Items	Determines which OLE popup menu commands are displayed and enabled when the mouse cursor is on the OLE object and the right mouse button is pressed. The OLE popup menu commands manipulate OLE objects.
	■ OLE Resize Style	Determines how an OLE object is displayed in an OLE container.
	■ OLE Tenant Aspect	Determines how an OLE object appears in an OLE container, either content, icon, or thumbnail.
	■ OLE Tenant Types	Specifies the type of OLE objects that can be tenants of the OLE container, either embedded, linked, any, static, or none.
	■ Show OLE Popup Menu	Determines whether the right mouse button displays a popup menu of commands for interacting with the OLE object.
	■ Show OLE Tenant Type	Determines whether a border defining the OLE object type surrounds the OLE container.
Report Builder	Create New	Specifies that you want to embed your OLE object within your report application.
	Create from File	Specifies that you want to link your OLE object within your report application.
	Display as Icon	Specifies whether the OLE object should appear as an icon. By default, the OLE object appears as an empty container.

6.1.1.7.4 OLE/ActiveX built-ins

This section lists the OLE and ActiveX built-ins supported by different components.

Component	Built-in	Description
Form Builder	■ ACTIVATE_SERVER	Activates an OLE server associated with an OLE container and prepares the OLE server to receive OLE automation events from the OLE container.
	■ ADD_OLEARGS	Establishes the type and value of an argument that will be passed to the OLE object's method.
	■ CALL_OLE	Passes control to the identified OLE object's method.
	■ CALL_OLE_<return type>	Passes control to the identified OLE object's method. Receives a return value of the specified type. There are five versions of the function (denoted by the value in returntype), one for each of the argument types CHAR, NUM, OBJ, RAW, and VAR.
	■ CLOSE_SERVER	Deactivates the OLE server associated with an OLE container. Terminates the connection between an OLE server and the OLE container.
	■ CREATE_OLEOBJ	In its first form, creates an OLE object, and establishes the object's persistence. In its second form, alters the persistence of a previously-instantiated OLE object.
	■ CREATE_VAR	Creates an empty, unnamed variant. There are two versions of the function, one for scalars and the other for arrays.
	■ DESTROY_VARIANT	Destroys a variant that was created by the CREATE_VAR function.

Component	Built-in	Description
	■ EXEC_VERB	Causes the OLE server to execute the verb identified by the verb name or the verb index. An OLE verb specifies the action that you can perform on an OLE object.
	■ FIND_OLE_VERB	Returns an OLE verb index. An OLE verb specifies the action that you can perform on an OLE object, and each OLE verb has a corresponding OLE verb index.
	■ GET_INTERFACE_POINTER	Returns a handle to an OLE2 automation object.
	■ GET_OLEARG_<type>	Obtains the nth argument from the OLE argument stack. There are five versions of the function (denoted by the value in type), one for each of the argument types CHAR, NUM, OBJ, RAW, and VAR.
	■ GET_OLE_MEMBERID	Obtains the member ID of a named method or property of an OLE object.
	■ GET_VAR_BOUNDS	Obtains the bounds of an OLE variant's array.
	■ GET_VAR_DIMS	Determines if an OLE variant is an array, and if so, obtains the number of dimensions in that array.
	■ GET_VAR_TYPE	Obtains the type of an OLE variant.
	■ GET_VERB_COUNT	Returns the number of verbs that an OLE server recognizes. An OLE verb specifies the action that you can perform on an OLE object, and the number of verbs available depends on the OLE server.
	■ GET_VERB_NAME	Returns the name of the verb that is associated with the given verb index.

Component	Built-in	Description
	■ INITIALIZE_CONTAINER	Inserts an OLE object from a server-compatible file into an OLE container.
	■ INIT_OLE_ARGS	Establishes how many arguments are going to be defined and passed to the OLE object's method.
	■ LAST_OLE_ERROR	Returns the identifying number of the most recent OLE error condition.
	■ LAST_OLE_EXCEPTION	Returns the identifying number of the most recent OLE exception that occurred in the called object.
	■ OLEVAR_EMPTY	An OLE variant of type VT_ EMPTY.
	■ PTR_TO_VAR	First, creates an OLE variant of type VT_PTR that contains the supplied address. Then, passes that variant and type through the function VARPTR_TO_VAR.
	■ RELEASE_OBJ	Shuts down the connection to the OLE object.
	■ SERVER_ACTIVE	Indicates whether or not the server associated with a given container is running.
	■ SET_OLE	Changes the value of an OLE property. There are three versions of the procedure, one for each of the new-value types: NUMBER, VARCHAR, and OLEVAR.
	■ SET_VAR	Sets a newly-created OLE variant to its initial value. Or, resets an existing OLE variant to a new value. There are four versions of the procedure, one for each of the new value types CHAR, NUMBER, OLEVAR, and table.

Component	Built-in	Description
	■ TABLE_FROM_BLOCK	Populates a table from a block.
	■ TO_VARIANT	Creates an OLE variant and assigns it a value. There are four versions of the function.
	■ VARPTR_TO_VAR	Changes a variant pointer into a simple variant.
	■ VAR_TO_TABLE	Reads an OLE array variant and populates a PL/SQL table from it.
	■ VAR_TO_<type>	Reads an OLE variant and transforms its value into an equivalent PL/SQL type. There are six versions of the function (denoted by the value in type), one for each for of the types CHAR, NUM, OBJ, RAW, TABLE, and VARPTR.
	■ VAR_TO_VARPTR	Creates an OLE variant that points to an existing variant.
Developer OLE2 Package	■ ADD_ARG	Adds an argument to a given argument list.
	■ CREATE_ARGLIST	Creates an argument list to be passed to an OLE server.
	■ CREATE_OBJ	Returns a handle to a newly created OLE object. This is usually used for OLE objects that do not have a user interface, such as a spell-checker.
	■ DESTROY_ARGLIST	Destroys the specified argument list.
	■ GET_CHAR_PROPERTY	Returns a character property of the OLE object.
	■ GET_NUM_PROPERTY	Returns a number property of the OLE object.
	■ GET_OBJ_PROPERTY	Returns an object type property of the OLE object.

Component	Built-in	Description
	■ INVOKE	Executes the specified OLE server procedure.
	■ INVOKE_CHAR	Executes the specified OLE server function. This function returns a character value.
	■ INVOKE_NUM	Executes the specified OLE server function. This function returns a number value.
	■ INVOKE_OBJ	Executes the specified OLE server function. This function returns an object type value.
	■ LAST_EXCEPTION	Returns an OLE error.
	■ SET_PROPERTY	Sets the OLE property with the specified value.
	■ RELEASE_OBJ	Deallocates all resources for the specified OLE object.

6.1.1.8 OLE guidelines

When working with OLE objects, consider these guidelines:

Item	Recommendation
Embedding or Linking an OLE object	You should link an OLE object when: ■ Your users prefer to work with the OLE object within the OLE server environment (your users prefer external activation). You link your OLE object when your users are more comfortable editing a spreadsheet, for example, within Microsoft Excel, rather than within your application. ■ The OLE object is used in multiple applications. ■ The size of your application is a concern. You should embed an OLE object when: ■ Your users can work with OLE objects within your application; your users prefer in-place activation. ■ You prefer to maintain a single application, rather than maintaining an application with multiple OLE source files. ■ You are not concerned about the size of your application.

Item	Recommendation
OLE Activation Style	You should use external activation. Linked objects can only be activated with external activation.
Display Style	for optimum performance, set the Display Style property for your OLE object to Icon.
Creating OLE objects at design-time or runtime?	You should create your OLE objects at design-time. When you create an OLE container in a Form, Form Builder automatically initializes the OLE object. In contrast, if you insert an OLE object at runtime, you must initialize the OLE object manually. **Note:** If you manually insert an OLE object during Forms Runtime, the OLE object appears in the OLE container until the next record query. For any subsequent record queries, the OLE container appears in a state as defined in the Form Builder or populated with an OLE object from the database.
Portability	OLE objects are only supported on Microsoft Windows. If portability is an issue, you should not incorporate OLE objects within your application. Instead, consider developing the features within Forms Developer (or Reports Developer), or consider developing a 3GL foreign function.
Setting OLE properties within Report Builder	Report Builder OLE container properties are only available in the Create OLE Object dialog; Report Builder does not expose OLE container properties within the Property Palette. When working within Report Builder, set OLE properties within the Create OLE Object dialog.

6.1.1.9 Adding an OLE object to your application

For detailed steps about how to add an OLE object to your application, refer to the online help.

6.1.1.10 Manipulating OLE objects

OLE server applications expose a set of commands that allow you to manipulate an OLE object programmatically.

You can manipulate OLE objects by:

- Getting and setting OLE properties.
- Calling OLE methods to perform special commands.

Note: Before you can call an OLE method, you must first import the OLE object's methods and properties into Forms Developer or Reports Developer. Importing OLE methods and properties enables you to interact with the OLE object within the native environment.

You can access OLE methods from your application by using the STANDARD (Form Builder only) and OLE2 built-in packages.

6.1.1.11 OLE examples

This section provides several examples to help you get started with OLE.

6.1.1.11.1 Example 1: setting an OLE property using bind variable syntax

Within your form applications, you can use the :item('item_name').ocx.server_name.property bind variable syntax to assign or retrieve property values.

For example:

```
:item('OLEitem').OCX.SpreadSheet.CellForeGroundColor:=
:item('OLEitem').OCX.SpreadSheet.CellForeGroundColor + 1;
```

OLEitem is the name of the item, SpreadSheet is the name of the OLE control server, and CellForeGroundColor is the name of the property.

6.1.1.11.2 Example 2: setting an OLE property using property assessors

Within your form applications, you can also use property assessor functions and procedures to get and set property values.

For example:

```
Variant OleVar;
Variant := EXCEL_WORKSHEET.ole_Range(:CTRL.interface,
  To_variant('A1'));
```

EXCEL_WORKSHEET is the name of the program unit created from the OLE Importer. OLE_RANGE is the name of the property accessor.

6.1.1.11.3 Example 3: modifying cells in an Excel spreadsheet

This example gets and sets cell values in an Excel spreadsheet.

```
PACKAGE spreadsheet IS
procedure setcell(trow number, col number, val number);
function getcell(trow number, col number) return number;
END;
```

```
PACKAGE BODY spreadsheet IS
obj_hnd ole2.obj_type;/* store the object handle */
FUNCTION get_object_handle return ole2.obj_type IS
BEGIN
  /* If the server is not active, activate the server
     and get the object handle.
  */
  if not forms_ole.server_active ('spreadsheet') then
    forms_ole.activate_server('spreadsheet');
    obj_hnd := forms_ole.get_interface_pointer('spreadsheet');
  end if;
  return obj_hnd;
END;
 /*
Excel cells are accessed with the following syntax in Visual Basic:
ActiveSheet.Cells(row, column).Value
In PL/SQL, we need to first get the active sheet using the
forms_ole.get_interface_pointer built-in.  We can then use that to call the
Cells method with the row and column in an argument list to get a handle to
that specific cell.  Lastly, we access the value of that cell.
*/

PROCEDURE SETCELL (trow number, col number, val number) IS
  d ole2.obj_type;
  c ole2.obj_type;
  n number;
  lst ole2.list_type;
BEGIN
  /* Activate the server and get the object handle
     to the spreadsheet.
  */
  d := get_object_handle;
  /* Create an argument list and insert the specified
     row and column into the argument list.
  */
  lst := ole2.create_arglist;
  ole2.add_arg(lst,trow);
  ole2.add_arg(lst,col);
  /* Call the Cells method to get a handle to the
     specified cell.
  */
  c := ole2.invoke_obj(d,'Cells',lst);
  /* Set the value of that cell. */
  ole2.set_property(c,'Value',val);
```

```
  /* Destroy the argument list and the cell object
     handle.
  */
  ole2.destroy_arglist(lst);
  ole2.release_obj(c);
END;

 FUNCTION GETCELL(trow number, col number) return number IS
  c ole2.obj_type;
  d ole2.obj_type;
  n number;
  lst ole2.list_type;
BEGIN
  /* Activate the server and get the object handle
     to the spreadsheet.
  */
  d := get_object_handle;
  /* Create an argument list and insert the specified
     row and column into the argument list.
  */
  lst := ole2.create_arglist;
  ole2.add_arg(lst,trow);
  ole2.add_arg(lst,col);
  /* Call the Cells method to get the value in the
     specified cell.
  */
  c := ole2.invoke_obj (d,'Cells',lst);
  n := ole2.get_num_property (c, 'Value');
  /* Destroy the argument list. */
  ole2.destroy_arglist(lst);
  ole2.release_obj(c);
  return n;
END;
END;
```

To access a cell, use the following code:

```
spreadsheet.setcell(3, 5, 91.73);
:block1.item1 := spreadsheet.getcell(2, 4);
```

6.1.2 What are ActiveX controls?

ActiveX controls (originally known as OLE or OCX controls) are stand-alone software components that you embed within your application to provide light-weight user interface controls.

ActiveX controls differ from OLE objects in several ways:

- An ActiveX control is not a separate application, but a server that plugs into an ActiveX container—ActiveX controls are self-contained.
- Each ActiveX control exposes a set of properties, methods, and *events*. Properties define the ActiveX control's physical and logical attributes, methods define actions that the ActiveX control can perform, and events denote some change in status in the ActiveX control.
- ActiveX controls must be deployed and installed with your applications.

6.1.2.1 When should I use ActiveX controls?

ActiveX controls are typically used to enhance an application by providing some additional, self-contained functionality.

For example, you can enhance your application with a tabbed property sheet, a spin control, a calendar control, a help control, and so on.

A significant amount of effort is required to develop your own ActiveX controls or OLE servers. It is recommended that you use ActiveX controls and OLE servers developed and distributed by third party vendors.

6.1.2.2 Manipulating ActiveX controls

Each ActiveX control exposes a set of properties, methods, and events. Properties define the ActiveX control's physical and logical attributes, methods define actions that the ActiveX control can perform, and events denote some change in status in the ActiveX control.

You can manipulate an ActiveX control by:

- Setting and getting ActiveX control properties.
- Calling ActiveX control methods.

  Note: Before you can invoke an ActiveX control method, you must first import its methods and events into Forms Developer. Importing ActiveX methods and events enables you to interact with the ActiveX control within the native Forms Developer environment.
- Responding to ActiveX control events.

To manipulate an ActiveX control, you use the STANDARD and OLE2 (both within Forms Developer) built-in packages.

6.1.2.3 Responding to ActiveX events

You can respond to an ActiveX event by writing your own code within an ActiveX event package or within the On-Dispatch-Event trigger.

Each ActiveX event is associated with a PL/SQL procedure defined in the events' package. When the control fires an event, the code in the procedure is automatically executed.

Procedure names are determined by an internal number that represents the corresponding event. The restricted procedure produced by an event has an application programming interface similar to the following:

```
PROCEDURE /*Click*/ event4294966696(interface OleObj);
```

Note: ActiveX procedures run in restricted mode. When calling the event procedure within an On-Dispatch-Event trigger, you can explicitly define whether the procedure is run in restricted or unrestricted mode by using the `FORMS4W.DISPATCH_EVENT` call. When defining a restricted procedure, `OUT` parameters are not observed.

6.1.2.4 Deploying your ActiveX control

Deploying an application that contains an ActiveX control requires that you deploy the ActiveX control.

To deploy an ActiveX control, you must:

- Register the ActiveX control on the client-machine.

  If you install an ActiveX control by using the installation program supplied with the ActiveX control, registration occurs automatically.

  For manual registration, use `regActiveX32.exe` or `regsvr32.exe`; both are available with Microsoft development tools and from ActiveX control vendors.

- Copy ActiveX DLLs to the client-machine (for example, `C:\WINDOWS\SYSTEM`).

  Most ActiveX controls require a supporting DLL, such as the Microsoft Foundation Class runtime library (`MFC40.DLL`). The DLL must be in the `\WINDOWS\SYSTEM` directory or in the search path. If the DLL is out of date or missing, your ActiveX control will not register properly.

Note: ActiveX controls, whether distributed by third party ActiveX control vendors or bundled with application development tools, may require that you pay

additional fees or obtain additional licenses prior to distributing the ActiveX control.

6.1.2.5 ActiveX support

Support means the ability to create, manipulate, and communicate with ActiveX controls.

Component	Container
Form Builder	Yes
Graphics Builder	No
Procedure Builder	No
Project Builder	No
Query Builder	No
Report Builder	No
Schema Builder	No
Translation Builder	No

6.1.2.5.1 ActiveX properties

This section lists the ActiveX properties supported by Forms Developer.

Component	Property	Description
Form Builder	OLE Class	Determines what class of OLE objects can reside in an OLE container.
	Control Properties	Allows you to set ActiveX control properties. You can access the ActiveX properties dialog through the Property Palette or by clicking the ActiveX control, then clicking the right mouse button.
	About Control	Displays information about the ActiveX control
	Control Help	Displays control-specific help (if available).

6.1.2.5.2 ActiveX/OLE built-ins

Refer to Section 6.1.1.7.4 for a list of the ActiveX and OLE built-ins supported by different components.

6.1.2.6 ActiveX guidelines

This section provides guidelines for working with ActiveX controls.

Item	Recommendation
Creating your own ActiveX Control	A significant amount of effort is required to develop your own ActiveX controls or OLE servers. It is recommended that you use ActiveX controls and OLE servers developed and distributed by third party vendors.
Initializing an ActiveX Control	Use ActiveX controls in blocks with the Single Record property set to Yes, because single records are immediately initialized when Forms Runtime starts up. For multiple records, each record is not initialized until you navigate to the record. Without initialization, the ActiveX control item is empty, giving the impression that no ActiveX control is available.
Managing OLE Variant Types	■ Some OLE servers such as Microsoft Excel use variant types. Use the STANDARD built-in package to do the necessary conversion to and from variant types. ■ The lifetime and scope of a variant type is limited to a trigger (variant memory space is released when a trigger exits). To extend the lifetime and scope of a variant type, set the persistent parameter in To_Variant() to TRUE and assign the results to a global variable. **Note:** Global variants must be explicitly destroyed using `Destroy_Variant()`. Similarly, OLE objects created with Create_OleObj() are global in scope (the persistent parameter defaults to TRUE). You must explicitly call `Release_Obj()` to release global objects.

Item	Recommendation
Moving ActiveX Files	You should maintain your ActiveX files within the “install” directory; do not move your ActiveX files to a different directory. At installation, the directories in which the ActiveX control is installed are registered in the Windows Registration Database in Windows 95 and Windows NT, making the ActiveX Control visible to your development environment. When you move an ActiveX Control to a different directory, or rename the directory, you invalidate the information in the registry. If you find it necessary to move the ActiveX Control or rename its directory, use `regsrv32.exe` or `regActiveX32.exe` utilities provided with most Microsoft development products to re-register the ActiveX in its new location.
Portability Issues	We support ActiveX on the Windows platform only. ActiveX controls cannot be used on the Web or on UNIX. If portability is an issue, do not use an ActiveX control.

Item	Recommendation

Debugging ActiveX Calls	Given that object types cannot be checked at compile time, it is possible to call a function on an object which is not defined for the class of that object. Because the functions are bound by ID rather than by name, a different function may be called than expected, leading to unusual errors.
	
	One way to guarantee that you are calling the correct method is to change the generated function, replacing the hardcoded constant with a call to GET_OLE_MEMBERID. For example:
	
	`Procedure Ole_Add(interface OleObj, TimeBegin VARCHAR2, TimeEnd VARCHAR2, Text VARCHAR2, BackColor OleVar := OleVar_Null) IS`
	
	`BEGIN`
	`Init_OleArgs (4);`
	`Add_OleArg (TimeBegin);`
	`Add_OleArg (TimeEnd);`
	`Add_Olearg (Text);`
	`Add_OleArg (BackColor);`
	`Call_Ole (interface, 2);`
	`END;`
	
	Replace the `Call_ole()` with: `Call_Ole (interface, Get_Ole_MemberID(interface, 'Add'));`
	
	You can check for `FORM_SUCCESS` after the `GET_OLE_MEMBERID` call.

Restrictions	■ ActiveX event procedures are restricted. In general, `GO_ITEM` cannot be called within ActiveX procedure code, unless the same event applies to multiple items and a `GO_ITEM` is necessary. In this case, you can use the `GO_ITEM` built-in by doing the following: in the On-Dispatch-Trigger (block or form level), call `DISPATCH_EVENT(RESTRICTED_ALLOWED)`. **Note:** You do not have to explicitly call the event procedure because it will automatically be called following the On-Dispatch trigger code. ■ Initialization events for ActiveX controls do not fire in Forms Runtime. These initialization events are intentionally disabled. Instead, you can use When-New-Item-Instance or When-New-Record-Instance in place of the control's native initialization events.

6.1.2.7 Adding an ActiveX control to your application

For information about how to add an ActiveX control to your application, refer to the online help.

6.1.2.8 ActiveX examples

This section provides several examples to help you get started with ActiveX controls.

6.1.2.8.1 Example 1: setting ActiveX control properties

In Form Builder, you can use the `:item('item_name').ocx.server_name.property` bind variable syntax to assign or retrieve ActiveX property values.

For example:

```
:item('ActXitem').OCX.Spindial.spindialctrl.1.Needleposition:=
:item('ActXitem').OCX.Spindial.spindialctrl.1.Needleposition + 1;
```

`ActXitem` is the name of the item, `Spindial.spindialctrl.1` is the name of the ActiveX control server, and `Needleposition` is the name of the property.

The following code also works if your `system.cursor_item` is an ActiveX control:

```
:form.cursor_item.OCX.spindial.spindialctrl.1.Needlposition :=
:form.cursor_item.OCX.spindial.spindialctrl.1.Needlposition + 1;
```

6.1.2.8.2 Example 2: getting ActiveX control properties

In Form Builder, you can use the property accessor functions and procedures to get and set ActiveX properties.

For example:

```
tblname varchar2;
tblname := table_pkg.TableName(:item('Oblk.Oitm').interface);
```

`Table_pkg` is the name of the program unit created from the OLE Importer. `TableName` is the name of the property accessor. `Oblk` is the name of the block and `Oitm` is the name of the item.

6.1.2.8.3 Example 3: calling ActiveX control methods

This example gets a cell value from a Spread Table ActiveX control by using the `GetCellByColRow` method, which is provided in the SpreadTable package.

```
DECLARE
   Cur_Row number;
   Cur_Col number;
   The_OLE_Obj OleObj;
BEGIN
   Cur_Row:=SpreadTable.CurrentRow(:ITEM('BLK.ITM').interface);
   Cur_Col:=SpreadTable.CurrentCol(:ITEM('BLK.ITM').interface);
   The_OLE_Obj:=SpreadTable.GetCellByColRow(:ITEM('BLK.ITM').interface,
                                        Cur_Col, Cur_Row);
END;
```

6.2 Using Foreign Functions to Customize Your Applications

You can customize and supplement your applications with foreign functions.

This section addresses:

- Section 6.2.1, "What are foreign functions?"
- Section 6.2.2, "The foreign function interface"
- Section 6.2.3, "Foreign function guidelines"
- Section 6.2.4, "Creating a foreign function"
- Section 6.2.5, "Foreign function examples"

6.2.1 What are foreign functions?

Foreign functions are subprograms written in a 3GL programming language that allow you to customize your applications to meet the unique requirements of your users.

Foreign functions can interact with Oracle databases, and both Forms Developer and Reports Developer variables, items, columns, and parameters. You can also call any external function, such as Windows DLLs or APIs.

6.2.1.1 When should I use a foreign function?

Foreign functions are often used to perform the following tasks:

- Perform complex data manipulation.
- Pass data to Forms Developer or Reports Developer from operating system text files.
- Manipulate LONG RAW data.
- Pass entire PL/SQL blocks for processing by the server.
- Set font and color attributes for applications.
- Send mail directly from an application.
- Display Windows help as part of your application.
- Access the Microsoft Windows SDK.
- Leverage low-level system services, such as pipes.
- Control real time devices, such as a printer or a robot.

6.2.1.2 Foreign function types

You can develop three types of foreign functions:

6.2.1.2.1 Oracle Precompiler foreign functions An Oracle Precompiler foreign function is the most common foreign function. Using the Oracle Precompiler, you can create foreign functions that access Oracle databases as well as Forms Developer or Reports Developer variables, items, columns, and parameters.

An Oracle Precompiler foreign function incorporates the Oracle Precompiler interface. This interface enables you to write a subprogram in one of the following supported host languages with embedded SQL commands: Ada, C, COBOL, FORTRAN, Pascal, and PL/I.

An Oracle Precompiler foreign function source file includes host programming language statements and Oracle Precompiler statements with embedded SQL statements. Precompiling an Oracle Precompiler foreign function replaces the embedded SQL statements with equivalent host programming language statements. After precompiling, you have a source file that you can compile with a host language compiler.

6.2.1.2.2 Oracle Call Interface (OCI) foreign functions An OCI foreign function incorporates the Oracle Call Interface. This interface enables you to write a subprogram that contains calls to Oracle databases. A foreign function that incorporates only the OCI (and not the Oracle Precompiler interface) cannot access Forms Developer or Reports Developer variables, items, columns, and parameters.

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

6.2.1.2.3 Non-Oracle foreign functions A non-Oracle foreign function does not incorporate either the Oracle Precompiler interface or the OCI. For example, a non-Oracle foreign function might be written entirely in the C language. A non-Oracle foreign function cannot access Oracle databases, or Forms Developer or Reports Developer variables, items, columns, and parameters.

6.2.2 The foreign function interface

Both Forms Developer and Reports Developer use PL/SQL as their programming language. In order to call a foreign function, such as a C function in a Windows DLL, PL/SQL must have an interface to communicate with the foreign function.

You can communicate with your foreign function through two distinct interfaces, either the Oracle Foreign Function Interface (`ORA_FFI`) or the user exit interface.

6.2.2.1 The Oracle Foreign Function Interface (ORA_FFI)

ORA_FFI is a portable and generic mechanism for enabling Forms Developer or Reports Developer to call 3GL routines from PL/SQL subprograms.

Foreign functions that are invoked from a PL/SQL interface must be contained in a dynamic library. Examples of dynamic libraries include dynamic link libraries on Microsoft Windows and shared libraries on UNIX systems.

6.2.2.2 User exit interface to foreign functions

The user exit interface is a platform-specific mechanism for enabling Forms Developer or Reports Developer to call 3GL routines from PL/SQL subprograms.

The foreign functions that you invoke from a user exit interface must be contained in a dynamic link library (.DLL) or linked with an application executable.

6.2.2.3 Comparing ORA_FFI and user exits

This section describes the advantages and disadvantages of using ORA_FFI and user exits.

Foreign Function	Advantage	Disadvantage
User Exit	■ User exits are linked to an executable. This "tight binding" allows you to use and take advantage of the current database connection.	■ The most significant disadvantage to using user exits is the maintenance burden. You must relink your user exit whenever you modify your user exit or upgrade Forms Developer or Reports Developer. ■ User exits are not generic; they are platform-specific.

Foreign Function	Advantage	Disadvantage
ORA_FFI	■ ORA_FFI is a pure PL/SQL specification. The ORA_FFI specification exists within a library (.PLL file), not within a component of Forms or Reports. When you upgrade to a higher version of Forms or Reports or modify the foreign function, you don't have to modify or regenerate the PLL file. ■ ORA_FFI is generic. ■ Both Forms and Reports provide several ORA_FFI packages (D2KWUTIL.PLL) that allow you to access libraries that are already available (Windows API functions).	■ If you are using ORA_FFI and you are writing your own external code modules with Pro*C, you cannot use the current open database connection. You must open a second connection. ■ You cannot pass complex datatypes, such as structures or arrays. For example, you cannot use EXEC TOOLS GET or EXEC TOOLS PUT to interface with Forms Developer or Reports Developer.

6.2.3 Foreign function guidelines

This section provides guidelines for working with foreign functions.

Item	Recommendation
Which foreign function interface should I use?	Use the Oracle Foreign Function Interface (ORA_FFI). ORA_FFI is a portable, generic, and requires only minor or no maintenance
Can I perform screen I/O from a foreign function?	You should not perform host language screen I/O from a foreign function. This restriction exists because the runtime routines that a host language uses to perform screen I/O conflict with the routines that Forms Developer and Reports Developer use to perform screen I/O. However, you can perform host language file I/O from a foreign function.
Which host language should I use to write my user exit?	Your host language is a matter of preference. However, C is the recommended language. **Note:** Some C runtime functions are not available in .DLL files. For more information, refer to your compiler documentation.

Item	Recommendation

Which precompiler should I use to precompile my user exit?	You should use Pro*C version 2.2.4 and 8.0.4.
	When precompiling, be sure to specify the following MSVC compiler flags:
	`Large, Segment Setup: SS != DS, DSloads on function entry`
	`Assume 'extern' and Uninitialized Data 'far' is checked Yes`
	`In Windows Prolog/Epilogue, Generate prolog/Epilogue for None`

Do I have to recompile my user exit when I upgrade from a previous version of Forms Developer or Reports Developer?	Yes. User exits can create a maintenance burden especially if you maintain several different executables, each with a different set of user exits. When you modify a user exit or upgrade to a higher version of Forms or Reports, you must relink the user exit with the Forms or Reports executables.
Can I deploy a foreign function on the Web?	`ORA_FFI` and user exits do not function on the Web. On web deployments, foreign functions interface with the DLLs on the server-side, not on the browser-side.

For more information about foreign functions, refer to the following publications:

ORACLE Precompiler interface	Programmer's Guide to the ORACLE Precompilers
Supported host languages	The Oracle Installation Guide for your operating system
Operating system-specific requirements when working with foreign functions	Online help
OCI	Oracle Call Interface Programmer's Guide
Building DLLs	Online help and your compiler documentation
ORA_FFI	Online help
User Exits	Online help
PL/SQL	PL/SQL User's Guide or online help

6.2.4 Creating a foreign function

This section provides detailed steps that describe how to create a foreign function interface:

- Creating an ORA_FFI interface to a foreign function
- Creating a user exit interface to a foreign function

6.2.4.1 Creating an ORA_FFI interface to a foreign function

The following example creates a PL/SQL package called WinSample. The WinSample package includes interfaces to the foreign function GetPrivateProfileString in the dynamic library KRNL386.EXE.

Note: When you create an ORA_FFI interface to a foreign function, you perform two basic steps. First, you create and associate a subprogram with a foreign function (the dispatcher function). By associating a PL/SQL subprogram with a foreign function, you can invoke the foreign function each time you call the associated PL/SQL subprogram. Associating a foreign function with a PL/SQL subprogram is necessary because both Forms Developer and Reports Developer use PL/SQL constructs. Second, you create a PL/SQL function which passes the arguments to the dispatcher function. The dispatcher function invokes the foreign function.

1. Create a package specification.

   Your package spec must represent the library. It must also define the PL/SQL function that you want to invoke.

   For example:

```
PACKAGE WinSample IS
FUNCTION GetPrivateProfileString
(Section IN  VARCHAR2,
Entry IN  VARCHAR2,
DefaultStr IN  VARCHAR2,
ReturnBuf IN  OUT VARCHAR2,
BufLen IN  PLS_INTEGER,
Filename IN  VARCHAR2)
RETURN PLS_INTEGER;
END;
```

   In this example, you call the WinSample.GetPrivateProfileString PL/SQL function to invoke the GetPrivateProfileString foreign function in the dynamic library KRNL386.EXE.

Note: You should check the parameters for the C function GetPrivateProfileString, and specify the matching PL/SQL parameter types and the PL/SQL return types. The C datatype int is equivalent to the PL/SQL parameter IN PLS_INTEGER and the PL/SQL return type PLS_INTEGER. The C datatype char is equivalent to the PL/SQL parameter IN VARCHAR2.

2. Define the library and its function handles.

   For example:

```
PACKAGE BODY WinSample IS
lh_KRNL386 ORA_FFI.LIBHANDLETYPE;
fh_GetPrivateProfileString ORA_FFI.FUNCHANDLETYPE;
```

   In this step, you declare the handle types for the library and the function. Later you will load the library and register the function using ORA_FFI.LOAD_LIBRARY and ORA_FFI.REGISTER_FUNCTION. Each of these functions returns a handle (a pointer) to the specified library and the function. ORA_FFI.LIBHANDLETYPE and ORA_FFI.FUNCHANDLETYPE are the PL/SQL datatypes for these handles.

3. Create the dispatcher function. The dispatcher function invokes your foreign function.

   For example:

```
 FUNCTION i_GetPrivateProfileString
(funcHandle IN ORA_FFI.FUNCHANDLETYPE,
Section IN OUT VARCHAR2,
Entry IN OUT VARCHAR2,
DefaultStr IN OUT VARCHAR2,
ReturnBuf IN OUT VARCHAR2,
BufLen IN PLS_INTEGER,
Filename IN OUT VARCHAR2)
 RETURN PLS_INTEGER;
 PRAGMA INTERFACE(C,i_GetPrivateProfileString,11265);
```

   The first argument of the dispatcher function that calls a foreign function must have at least one parameter, and the first parameter must be a handle to the registered foreign function that the subprogram invokes.

   When you call the dispatcher function from the PL/SQL function, you pass the function handle as defined in step 2 (fh_GetPrivateProfileString).

When the dispatcher function gets called, the PRAGMA statement passes control to a memory location (11265 as specified in the above code) that communicates with the dynamic library.

4. Create the PL/SQL function that calls the dispatcher function. This PL/SQL function is the function that you defined in the package spec (Step 1).

For example:

```
FUNCTION GetPrivateProfileString
(Section IN  VARCHAR2,
Entry IN  VARCHAR2,
DefaultStr IN  VARCHAR2,
ReturnBuf IN  OUT VARCHAR2,
BufLen IN  PLS_INTEGER,
Filename IN  VARCHAR2)
RETURN PLS_INTEGER IS
Section_l VARCHAR2(512)    := Section;
Entry_l VARCHAR2(512)    := Entry;
DefaultStr_l VARCHAR2(512) := DefaultStr;
ReturnBuf_l VARCHAR2(512)  := RPAD(SUBSTR(NVL
(ReturnBuf,' '),1,512),512,CHR(0));
BufLen_l PLS_INTEGER    := BufLen;
Filename_l VARCHAR2(512)   := Filename;
rc PLS_INTEGER;
BEGIN
     rc  := i_GetPrivateProfileString
(fh_GetPrivateProfileString,
Section_l,
Entry_l,
DefaultStr_l,
ReturnBuf_l,
BufLen_l,
Filename_l);
ReturnBuf := ReturnBuf_l;
RETURN (rc);
END;
```

This is the PL/SQL function you call from your application. This function passes the arguments to the dispatcher function i_GetPrivateProfileString, then i_GetPrivateProfileString invokes the C function GetPrivateProfileString in KRNL386.EXE. Recall that the first argument of a dispatcher function must be a function handle. Here fh_GetPrivateProfileString is used to pass the function handle declared in Step 2.

5. Build the package body.

   The package body must perform four steps to initialize a foreign function:

   - Load the library
   - Register the functions that are in the library
   - Register the parameters (if any)
   - Register the return type (if any)

   For example:

```
BEGIN
/* Load the library .*/
 lh_KRNL386 := ORA_FFI.LOAD_LIBRARY
('location of the DLL here','KRNL386.EXE');

/* Register the foreign function. */
fh_GetPrivateProfileString := ORA_FFI.REGISTER_FUNCTION(lh_
KRNL386,'GetPrivateProfileString',ORA_FFI.PASCAL_STD);

/* Register the parameters. */
ORA_FFI.REGISTER_PARAMETER
(fh_GetPrivateProfileString,ORA_FFI.C_CHAR_PTR); ORA_FFI.REGISTER_
PARAMETER
(fh_GetPrivateProfileString,ORA_FFI.C_CHAR_PTR); ORA_FFI.REGISTER_
PARAMETER
(fh_GetPrivateProfileString,ORA_FFI.C_CHAR_PTR); ORA_FFI.REGISTER_
PARAMETER
(fh_GetPrivateProfileString,ORA_FFI.C_CHAR_PTR);
   ORA_FFI.REGISTER_PARAMETER
(fh_GetPrivateProfileString,ORA_FFI.C_INT); ORA_FFI.REGISTER_PARAMETER
(fh_GetPrivateProfileString,ORA_FFI.C_CHAR_PTR);

/* Register the return type. */
ORA_FFI.REGISTER_RETURN(fh_GetPrivateProfileString,ORA_FFI.C_INT);
END WinSample;
```

Recall that you declared two handles for the library and the function in Step 2. In this step, you assign values to the handles by using the ORA_FFI.LOAD_LIBRARY and ORA_FFI.REGISTER_FUNCTION functions.

ORA_FFI.LOAD_LIBRARY takes two arguments: the location and the name of the dynamic library. ORA_FFI.REGISTER_FUNCTION takes three arguments: the library handle for the library where the function resides, the function name, and the calling standard. The calling standard can be either C_STD (for the C calling standard) or PASCAL_STD (for the Pascal calling standard).

After you load the library and register the function, you must register the parameters and return types (if there are any).

ORA_FFI.REGISTER_PARAMETER and ORA_FFI.REGISTER_RETURN take two arguments each: the function handle and the argument type.

6. Within Forms Developer or Reports Developer, create a library file (.PLL) that includes your package, then attach it to your application.

7. Call the foreign function from your application.

   For example:

```
x := Winsample.GetPrivateProfileString
('Oracle', 'ORACLE_HOME', '<Not Set>', 'Value', 100,'oracle.ini');
```

6.2.4.2 Creating a user exit interface to a foreign function

User exits are not generic; they are platform-specific. Some details of implementing user exits are specific to each operating system. The following example describes how to create a user exit on Windows 95.

On Microsoft Windows, a foreign function that can be invoked from a user exit is contained in a dynamic link library (.DLL). A DLL is a library that loads into memory only when the contained code is invoked.

6.2.4.2.1 Example: creating a user exit on Windows 95

The following example creates a foreign function that adds an ID column to the EMP table.

This example uses several sample files, including:

- **UE_SAMP.MAK** is a project file that includes the IAPXTB control structure. Building this project generates UE_SAMP.DLL.
- **IFXTB60.DLL** is the default file containing foreign functions that can be invoked from a user exit interface. This file is a DLL that ships with Form

Builder, and does not initially contain user-defined foreign functions. This file is placed in the ORACLE_HOME\BIN directory during installation. When you create new foreign functions, replace the existing IFXTB60.DLL file with a new IFXTB60.DLL.

- **UE_XTB.C** is a template source file for creating an IAPXTB control structure. UE_XTB.C contains an example of an entry for the IAPXTB control structure. Modify this file and add your foreign function entries.
- **UE.H** is a sample header file that is used to define the IAPXTB control structure.
- **IFXTB60.DEF** contains definitions you need to build your own DLL. Use IFXTB60.DEF to export foreign functions. IFXTB60.DEF contains several export statements. You should not modify these export statements as they are used by Form Builder to access the user exit interface.
- **UEZ.OBJ** is an .OBJ file that you link to your own .OBJ files.

The user exit sample files are located in your ORACLE_HOME directory (for example, C:\ORAWIN95\FORMS60\USEREXIT).

1. Write a foreign function.

   For example, create a text file called UEXIT.PC, then add the following:

```
/* UEXIT.PC file */
/* This foreign function adds an ID column to the EMP table. */
#ifndef UE
#include "ue.h"
#endif
#ifndef _WINDLL
#define SQLCA_STORAGE_CLASS  extern
#endif
EXEC SQL INCLUDE sqlca.h;
void  AddColumn() {
EXEC SQL alter table EMP add ID varchar(9);
}
```

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

   For example, use Pro*C to precompile the UEXIT.PC file. When you precompile UEXIT.PC, Pro*C creates a C file called UEXIT.C.

   Note: When precompiling, be sure to specify the following MSVC compiler flags:

   ```
   Large, Segment Setup: SS != DS, DSloads on function entry
   ```

```
Assume 'extern' and Uninitialized Data 'far' is checked Yes

In Windows Prolog/Epilogue, Generate prolog/Epilogue for
None
```

3. Create your header files.

   Your header file must define your foreign function.

   For example, modify the sample header file, UE.H, by adding the following:

```
extern   void   AddColumn();
```

4. Create the IAPXTB control structure.

   For example, modify the sample file, UE_XTB.C, by adding an include statement for UE.H (# include "ue.h"), the name of the user exit (Add_ID_Column), the name of the foreign function (AddColumn), and the language type(XITCC).

```
#ifndef UE
#include "ue.h"
#endif /* UE */
#include "ue_samp.h"
/* Define the user exit table */
exitr iapxtb[] =  {  /* Holds exit routine pointers */
     "Add_ID_Column", AddColumn, XITCC,
     (char *) 0, 0, 0         /* zero entry marks the end */
}; /* end iapxtb */
```

5. Build your DLL. The steps for building a DLL vary depending on your particular compiler. For more information, refer to your compiler documentation.

   For example, using your compiler, create a project that contains: UE_SAMP.MAK, IFXTB60.DEF, UEZ.OBJ, UE_XTB.C, and UEXIT.C.

   Before building your DLL, you must link the following files:

```
LIBC.LIB
OLDNAMES
C:\ORAWIN95\FORMS60\USEREXIT\IFR60.LIB
C:\ORAWIN95\PRO20\USEREXIT\SQLLIB18.LIB
C:\ORAWIN95\PRO20\USEREXIT\SQXLIB18.LIB
```

After building the UE_SAMP.MAK project, the result is a DLL named UE_SAMP.DLL. Add the UE_SAMP.DLL entry to the list of DLLs defined by the FORMS60_USEREXITS parameter in the registry.

Alternatively, you can rename UE_SAMP.DLL to IFXTB60.DLL, backup the IFXTB60.DLL in the C:\ORAWIN95\BIN directory, and copy the new IFXTB60.DLL to the C:\ORAWIN95\BIN directory.

6. Invoke the foreign function from a user exit.

   For example, create a When-Button-Pressed Trigger that calls the foreign function from a user exit.

   The following statement demonstrates how to invoke the AddColumn foreign function by specifying the user exit name Add_ID_Column in the USER_EXIT built-in:

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

6.2.5 Foreign function examples

This section includes several examples that describe how to use foreign functions.

6.2.5.1 Using ORA_FFI to call Windows help

```
/* WinHelp ORA_FFI.                                                    */
/*                                                                     */
/*                                                                     */
/* Usage: WinHelp.WinHelp(helpfile VARCHAR2,                           */
/*                        command  VARCHAR2,                           */
/*                        data    {VARCHAR2/PLS_INTEGER See Below})    */
/*                                                                     */
/*        command can be one of the following:                         */
/*                                                                     */
/*            'HELP_INDEX'         Help Contents                       */
/*            'HELP_CONTENTS'            "                             */
/*            'HELP_CONTEXT'       Context Key  (See below)            */
/*            'HELP_KEY'           Key Search                          */
/*            'HELP_PARTIALKEY'    Partial Key Search                  */
/*            'HELP_QUIT'          Quit                                */
/*                                                                     */
/*        data contains a string for the key search or a numeric context */
/*        value if using topics.                                       */
/*                                                                     */
/* Winhelp.Winhelp('C:\ORAWIN95\TOOLS\DOC60\US\IF60.HLP',               */
```

```
/*                    'HELP_PARTIALKEY',                                    */
/*                    'ORA_FFI');                                           */
/*                                                                          */
/* The commented sections replace the line below if using HELP_CONTEXT keys */

PACKAGE WinHelp IS
  FUNCTION WinHelp(helpfile IN VARCHAR2,
                   command  IN VARCHAR2,
                   data     IN VARCHAR2)
                   RETURN PLS_INTEGER;
END;

PACKAGE BODY WinHelp IS
  lh_USER ora_ffi.libHandleType;
  fh_WinHelp ora_ffi.funcHandleType;

  FUNCTION i_WinHelp(funcHandle IN ora_ffi.funcHandleType,
                     hwnd       IN PLS_INTEGER,
                     helpfile   IN OUT VARCHAR2,
                     command    IN PLS_INTEGER,
                     data       IN OUT VARCHAR2)
                     RETURN PLS_INTEGER;

  PRAGMA INTERFACE(C,i_WinHelp,11265);

  FUNCTION WinHelp(helpfile IN VARCHAR2,
                   command  IN VARCHAR2,
                   data     IN VARCHAR2)
                   RETURN PLS_INTEGER
  IS
    hwnd_l     PLS_INTEGER;
    helpfile_l VARCHAR2(512) := helpfile;
    command_l  PLS_INTEGER;
    data_l     VARCHAR2(512) := data;
    rc         PLS_INTEGER;

    FUNCTION Help_Convert(command IN VARCHAR2)
                          RETURN PLS_INTEGER
    IS
    BEGIN
      /* The windows.h definitions for command */

      /* HELP_CONTEXT      0x0001 */
      /* HELP_QUIT         0x0002 */
      /* HELP_INDEX        0x0003 */
```

```
      /* HELP_CONTENTS     0x0003 */
      /* HELP_HELPONHELP   0x0004 */
      /* HELP_SETINDEX     0x0005 */
      /* HELP_SETCONTENTS  0x0005 */
      /* HELP_CONTEXTPOPUP 0x0008 */
      /* HELP_FORCEFILE    0x0009 */
      /* HELP_KEY          0x0101 */
      /* HELP_COMMAND      0x0102 */
      /* HELP_PARTIALKEY   0x0105 */
      /* HELP_MULTIKEY     0x0201 */
      /* HELP_SETWINPOS    0x0203 */

      if command = 'HELP_CONTEXT'    then return(1);   end if;
      if command = 'HELP_KEY'        then return(257); end if;
      if command = 'HELP_PARTIALKEY' then return(261); end if;
      if command = 'HELP_QUIT'       then return(2);   end if;
      /* If nothing else go to the contents page */
      return(3);
    END;

  BEGIN
    hwnd_l :=
    TO_PLS_INTEGER(Get_Item_Property(name_in('SYSTEM.CURSOR_ITEM'),WINDOW_
HANDLE));

    command_l := Help_Convert(command);

    rc  := i_WinHelp(fh_WinHelp,
                     hwnd_l,
                     helpfile_l,
                     command_l,
                     data_l);

    RETURN (rc);
  END ;

BEGIN
  BEGIN
    lh_USER := ora_ffi.find_library('USER.EXE');
    EXCEPTION WHEN ora_ffi.FFI_ERROR THEN
    lh_USER := ora_ffi.load_library(NULL,'USER.EXE');
  END ;

  fh_WinHelp :=
ora_ffi.register_function(lh_USER,'WinHelp',ora_ffi.PASCAL_STD);
```

```
  ora_ffi.register_parameter(fh_WinHelp,ORA_FFI.C_INT);        /* HWND   */
  ora_ffi.register_parameter(fh_WinHelp,ORA_FFI.C_CHAR_PTR);   /* LPCSTR */
  ora_ffi.register_parameter(fh_WinHelp,ORA_FFI.C_INT);        /* UINT   */
  ora_ffi.register_parameter(fh_WinHelp,ORA_FFI.C_CHAR_PTR);   /* DWORD  */

  ora_ffi.register_return(fh_WinHelp,ORA_FFI.C_INT);           /* BOOL   */

END WinHelp;
```

6.2.5.2 Using ORA_FFI to open the File Open dialog on Windows

```
PACKAGE OraDlg IS
FUNCTION OraMultiFileDlg
(Title IN  VARCHAR2,
Filter IN  VARCHAR2,
Dir IN  VARCHAR2,
FileString IN OUT VARCHAR2)
RETURN PLS_INTEGER;
FUNCTION OraSingleFileDlg
(Title IN  VARCHAR2,
Filter IN  VARCHAR2,
Dir IN  VARCHAR2,
FileString IN OUT VARCHAR2)
RETURN PLS_INTEGER;
END OraDlg;
PACKAGE BODY OraDlg IS
     lh_ORADLG ora_ffi.libHandleType;
fh_OraMultiFileDlg ora_ffi.funcHandleType;
fh_OraSingleFileDlg ora_ffi.funcHandleType;
FUNCTION i_OraMultiFileDlg
(funcHandle IN ora_ffi.funcHandleType,
Title IN OUT VARCHAR2,
Filter IN OUT VARCHAR2,
Dir IN OUT VARCHAR2,
FileString IN OUT VARCHAR2)
RETURN PLS_INTEGER;
PRAGMA INTERFACE(C,i_OraMultiFileDlg,11265);
FUNCTION OraMultiFileDlg
(Title IN  VARCHAR2,
Filter IN  VARCHAR2,
Dir IN  VARCHAR2,
FileString IN OUT VARCHAR2)
RETURN PLS_INTEGER IS
```

```
Title_l VARCHAR2(128)  :=  RPAD(SUBSTR(NVL(Title,'Open'),1,128),128,CHR(0));
Filter_l VARCHAR2(128)  :=  RPAD(SUBSTR(NVL
(Filter,'All Files (*.*)|*.*|'),1,128),128,CHR(0));
Dir_l VARCHAR2(256)  :=  RPAD(SUBSTR(NVL(Dir,' '),1,256),256,CHR(0));
FileString_l VARCHAR2(2000)  :=  RPAD(SUBSTR(NVL(FileString,'
'),1,2000),2000,CHR(0));
rc PLS_INTEGER;
BEGIN
     rc  := i_OraMultiFileDlg(fh_OraMultiFileDlg,
Title_l,
Filter_l,
Dir_l,
FileString_l);
     FileString := FileString_l;
RETURN (rc);
END ;
FUNCTION i_OraSingleFileDlg
(funcHandle IN ora_ffi.funcHandleType,
Title IN OUT VARCHAR2,
Filter IN OUT VARCHAR2,
Dir IN OUT VARCHAR2,
FileString IN OUT VARCHAR2)
RETURN PLS_INTEGER;
PRAGMA INTERFACE(C,i_OraSingleFileDlg,11265);
FUNCTION OraSingleFileDlg
(Title IN  VARCHAR2,
Filter IN  VARCHAR2,
Dir IN  VARCHAR2,
FileString IN OUT VARCHAR2)
RETURN PLS_INTEGER IS
Title_l VARCHAR2(128)  :=  RPAD(SUBSTR(NVL(Title,'Open'),1,128),128,CHR(0));
Filter_l VARCHAR2(128)  :=  RPAD(SUBSTR(NVL
(Filter,'All Files (*.*)|*.*|'),1,128),128,CHR(0));
Dir_l VARCHAR2(256)  :=  RPAD(SUBSTR(NVL(Dir,' '),1,256),256,CHR(0));
FileString_l VARCHAR2(2000)  :=  RPAD(SUBSTR(NVL(FileString,'
'),1,2000),2000,CHR(0));
rc PLS_INTEGER;
BEGIN
     rc  := i_OraSingleFileDlg(fh_OraSingleFileDlg,
Title_l,
Filter_l,
Dir_l,
FileString_l);
     FileString := FileString_l;
RETURN (rc);
```

```
END ;
BEGIN
     BEGIN
     lh_ORADLG := ora_ffi.find_library('ORADLG.DLL');
     EXCEPTION WHEN ora_ffi.FFI_ERROR THEN
     lh_ORADLG := ora_ffi.load_library(NULL,'ORADLG.DLL');
     END ;
     fh_OraMultiFileDlg := ora_ffi.register_function
(lh_ORADLG,'OraMultiFileDlg',ora_ffi.PASCAL_STD);
     ora_ffi.register_parameter(fh_OraMultiFileDlg,ORA_FFI.C_CHAR_PTR);
     ora_ffi.register_parameter(fh_OraMultiFileDlg,ORA_FFI.C_CHAR_PTR);
     ora_ffi.register_parameter(fh_OraMultiFileDlg,ORA_FFI.C_CHAR_PTR);
     ora_ffi.register_parameter(fh_OraMultiFileDlg,ORA_FFI.C_CHAR_PTR);
     ora_ffi.register_return(fh_OraMultiFileDlg,ORA_FFI.C_LONG);
     fh_OraSingleFileDlg := ora_ffi.register_function
(lh_ORADLG,'OraSingleFileDlg',ora_ffi.PASCAL_STD);
     ora_ffi.register_parameter(fh_OraSingleFileDlg,ORA_FFI.C_CHAR_PTR);
     ora_ffi.register_parameter(fh_OraSingleFileDlg,ORA_FFI.C_CHAR_PTR);
     ora_ffi.register_parameter(fh_OraSingleFileDlg,ORA_FFI.C_CHAR_PTR);
     ora_ffi.register_parameter(fh_OraSingleFileDlg,ORA_FFI.C_CHAR_PTR);
     ora_ffi.register_return(fh_OraSingleFileDlg,ORA_FFI.C_LONG);
END OraDlg;
```

6.2.5.3 Using ORA_FFI to call Unix(SUN) executables with a STDIN/STDOUT type interface

```
/* Copyright (c) 1997 by Oracle Corporation */
/*
   NAME
     ora_pipe_io_spec.sql - Specification for access to Unix Pipe mechanism
   DESCRIPTION
     Demonstration of how to use the ORA_FFI Package to provide access to the
     Unix Pipe C functions.
   PUBLIC FUNCTION(S)
     popen    - Open the Pipe command
     get_line - Get a line of Text from a Pipe
     put_line - Put a line of Text into a Pipe
     pclose   - Close the Pipe
     is_open  - Determine whether the Pipe descriptor is open.
   NOTES

     In Order to use these routines you could write  the following
     PL/SQL Code:

        -- Example of Calls to ora_pipe_io functions
```

```
        DECLARE
            stream ora_pipe_io.PIPE;
            buffer VARCHAR2(240);
        BEGIN
            stream := ora_pipe_io.popen('ls -l', ora_pipe_io.READ_MODE);

            loop
                exit when not ora_pipe_io.get_line(stream, buffer, 240);
                :directory.file := buffer;
                down;
            end loop;

            ora_pipe_io.pclose(stream);
        END;

   MODIFIED   (MM/DD/YY)
    smclark    08/05/94 -  Creation
*/

PACKAGE ora_pipe_io is

    /*
    ** Arguments to popen.
    */
    READ_MODE constant VARCHAR2(1) := 'r';
    WRITE_MODE constant VARCHAR2(1) := 'w';

    /* ------------- TYPE PIPE ----------- */
    /*
    ** Public Type PIPE - Handle to a Un*x pipe
    **
    ** Do not modify the private members of this type
    */
    TYPE PIPE is RECORD
        (file_handle ORA_FFI.POINTERTYPE,
         is_open boolean,
         read_write_mode VARCHAR2(1));


    /* ------------ FUNCTION POPEN ----------- */
    /*
    ** Function POPEN -- Open a Un*x pipe command
    **
    ** Given a Unix command to execute and a Pipe read/write mode in which
    ** to execute the instruction this Function will execute the Command
```

```
** and return a handle, of type PIPE, to the resulting Input/Output
** stream.
**
** The command to be executed is limited to 1024 characters.
*/
FUNCTION popen(command in VARCHAR2,
               ctype in VARCHAR2)
RETURN PIPE;


/* ------------ PROCEDURE PCLOSE ----------- */
/*
** Procedure PCLOSE -- Close a pipe
**
** Close a previously opened pipe.
**
** Raises a VALUE_ERROR exception if incorrect arguments are passed.
*/
PROCEDURE pclose(stream in out PIPE);


/* ------------ FUNCTION GET_LINE ----------- */
/*
** Function GET_LINE
** -- Get a line of text into a buffer from the read mode pipe.
**
** Get a line of text from a previously opened pipe.
**
** Raises a VALUE_ERROR exception if incorrect arguments are passed.
** For example
** if you pass a pipe which has never been opened (using popen)
*/
FUNCTION get_line(stream in out PIPE,
                  s in out VARCHAR2,
                  n in PLS_INTEGER)
RETURN BOOLEAN;


/* ------------ PROCEDURE PUT_LINE ----------- */
/*
** Procedure PUT_LINE -- Put a line of text into a a write mode pipe.
**
** Put a line of text into a previously opened pipe.
**
** Raises a VALUE_ERROR exception if incorrect arguments are passed.
```

```
    ** For example
    ** if you pass a pipe which has never been opened (using popen)
    **
    ** The Internal buffer for the string to write is limited to 2048 bytes
    */
    PROCEDURE put_line(stream in out PIPE,
                       s in VARCHAR2);


    /* ------------ FUNCTION IS_OPEN ----------- */
    /*
    ** Function IS_OPEN -- Determines whether a pipe is open.
    **
    ** Returns TRUE if the pipe is open, FALSE if the pipe is closed.
    */
    FUNCTION is_open(stream in PIPE)
    RETURN BOOLEAN;
END;

/*    ora_pipe_io_body.sql - Body of Package for access to Unix Pipe mechanism
   DESCRIPTION
     Demonstration of how to use the ORA_FFI Package to provide access to the
     Unix Pipe C functions.
   PUBLIC FUNCTION(S)
     popen    - Open the Pipe command
     get_line - Get a line of Text from a Pipe
     put_line - Put a line of Text into a Pipe
     pclose   - Close the Pipe
     is_open  - Determine whether the Pipe descriptor is open.
   PRIVATE FUNCTION(S)
     icd_popen, icd_fgets, icd_fputs, icd_pclose
   NOTES
   MODIFIED   (MM/DD/YY)
    smclark    11/02/94 -  Modified for production release changes to ORA_FFI.
    smclark    08/05/94 -  Creation
*/

 PACKAGE BODY ora_pipe_io is
    lh_libc   ora_ffi.libHandleType;
    fh_popen  ora_ffi.funcHandleType;
    fh_pclose ora_ffi.funcHandleType;
    fh_fgets  ora_ffi.funcHandleType;
    fh_fputs  ora_ffi.funcHandleType;

    /* ------------ FUNCTION ICD_POPEN ----------- */
```

```
/*
** Function ICD_POPEN -- Interface routine to C function popen
**
** This function acts as the interface to the popen function in
** libc.
*/
FUNCTION icd_popen(funcHandle in ora_ffi.funcHandleType,
                   command in out VARCHAR2,
                   ctype in out VARCHAR2)
return ORA_FFI.POINTERTYPE;

pragma interface(c, icd_popen, 11265);


/* ------------ PROCEDURE ICD_PCLOSE ----------- */
/*
** Function ICD_PCLOSE -- Interface routine to C function pclose
**
** This function acts as the interface to the pclose function in
** libc.
*/
PROCEDURE icd_pclose(funcHandle in ora_ffi.funcHandleType,
                     stream in out ORA_FFI.POINTERTYPE);

pragma interface(c, icd_pclose, 11265);


/* ------------ FUNCTION ICD_FGETS ----------- */
/*
** Function ICD_FGETS -- Interface routine to C function fgets
**
** This function acts as the interface to the fgets function in
** libc.
*/
FUNCTION icd_fgets(funcHandle in ora_ffi.funcHandleType,
                   s in out VARCHAR2, n in PLS_INTEGER,
                   stream in out ORA_FFI.POINTERTYPE)
RETURN ORA_FFI.POINTERTYPE;

pragma interface(c, icd_fgets, 11265);


/* ------------ FUNCTION ICD_FPUTS ----------- */
/*
** Function ICD_FPUTS -- Interface routine to C function fputs
```

```
**
** This function acts as the interface to the fputs function in
** libc.
*/
PROCEDURE icd_fputs(funcHandle in ora_ffi.funcHandleType,
                    s in out VARCHAR2,
                    stream in out ORA_FFI.POINTERTYPE);

pragma interface(c, icd_fputs, 11265);


/* ------------ FUNCTION POPEN ----------- */
/*
** Function POPEN -- Open a Un*x pipe command
*/
FUNCTION popen(command in VARCHAR2,
               ctype in VARCHAR2)
RETURN PIPE is

    /*
    ** Take a copy of the arguments because we need to pass them
    ** IN OUT to icd_popen, but we really don't want people to have
    ** to call our routines in the same way.
    */
    cmd varchar2(1024) := command;
    cmode varchar2(1) := ctype;

    stream PIPE;
BEGIN
    if (cmode not in (READ_MODE, WRITE_MODE))
       or (cmode is NULL)
       or (cmd is NULL)
    then
        raise VALUE_ERROR;
    end if;

    stream.file_handle := icd_popen(fh_popen, cmd, cmode);
    stream.is_open := TRUE;
    stream.read_write_mode := ctype;
    return(stream);
END popen;


/* ------------ PROCEDURE PCLOSE ----------- */
/*
```

```
** Procedure PCLOSE -- Close a pipe
*/
PROCEDURE pclose(stream in out PIPE) is
BEGIN
    icd_pclose(fh_pclose, stream.file_handle);
    stream.is_open := FALSE;
END pclose;


/* ------------ FUNCTION GET_LINE ----------- */
/*
** Function GET_LINE -- Get a line of text into a buffer
** from the read mode pipe.
*/
FUNCTION get_line(stream in out PIPE,
                  s in out VARCHAR2, n in PLS_INTEGER)
RETURN BOOLEAN is
    buffer ORA_FFI.POINTERTYPE;
BEGIN
    if (n <= 0)
       or (stream.is_open = FALSE)
       or (stream.is_open is NULL)
       or (stream.read_write_mode <> READ_MODE)
    then
        raise VALUE_ERROR;
    end if;

    /*
    ** Initialise the Buffer area to reserve the correct amount of space.
    */

    s := rpad(' ', n);

    buffer := icd_fgets(fh_fgets, s, n, stream.file_handle);

    /*
    ** Determine whether a NULL pointer was returned.
    */
    return (ora_ffi.is_null_ptr(buffer) = FALSE);
END get_line;


/* ------------ PROCEDURE PUT_LINE ----------- */
/*
** Procedure PUT_LINE -- Put a line of text into a a write mode pipe.
```

```
    */
    PROCEDURE put_line(stream in out PIPE,
                       s in VARCHAR2) is
        buffer varchar2(2048) := s;
    BEGIN
        if (stream.is_open = FALSE)
           or (stream.is_open is NULL)
           or (stream.read_write_mode <> WRITE_MODE)
        then
            raise VALUE_ERROR;
        end if;

        icd_fputs(fh_fputs, buffer, stream.file_handle);
        buffer := chr(10);
        icd_fputs(fh_fputs, buffer, stream.file_handle);
    END put_line;


    /* ------------ FUNCTION IS_OPEN ----------- */
    /*
    ** Function IS_OPEN -- Determines whether a pipe is open.
    */
    FUNCTION is_open(stream in PIPE)
    RETURN BOOLEAN is
    BEGIN
        return(stream.is_open);
    END is_open;

BEGIN
    /*
    ** Declare a library handle as libc.  (Internal so NULL,NULL)
    */
    lh_libc := ora_ffi.load_library(NULL, NULL);
    if ora_ffi.is_null_ptr(lh_libc) then
        raise VALUE_ERROR;
    end if;

    /*
    ** Register the popen function, it's return type and arguments.
    */
    fh_popen := ora_ffi.register_function(lh_libc, 'popen');
    if ora_ffi.is_null_ptr(fh_popen) then
        raise VALUE_ERROR;
    end if;
    ora_ffi.register_return(fh_popen, ORA_FFI.C_DVOID_PTR);
```

```
    ora_ffi.register_parameter(fh_popen, ORA_FFI.C_CHAR_PTR);
    ora_ffi.register_parameter(fh_popen, ORA_FFI.C_CHAR_PTR);

    /*
    ** Register the pclose function, it's return type and arguments.
    */
    fh_pclose := ora_ffi.register_function(lh_libc, 'pclose');
    if ora_ffi.is_null_ptr(fh_pclose) then
        raise VALUE_ERROR;
    end if;
    ora_ffi.register_return(fh_pclose, ORA_FFI.C_VOID);
    ora_ffi.register_parameter(fh_pclose, ORA_FFI.C_DVOID_PTR);

    /*
    ** Register the fgets function, it's return type and arguments.
    */
    fh_fgets := ora_ffi.register_function(lh_libc, 'fgets');
    if ora_ffi.is_null_ptr(fh_fgets) then
        raise VALUE_ERROR;
    end if;
    ora_ffi.register_return(fh_fgets, ORA_FFI.C_DVOID_PTR);
    ora_ffi.register_parameter(fh_fgets, ORA_FFI.C_CHAR_PTR);
    ora_ffi.register_parameter(fh_fgets, ORA_FFI.C_INT);
    ora_ffi.register_parameter(fh_fgets, ORA_FFI.C_DVOID_PTR);

    /*
    ** Register the fputs function, it's return type and arguments.
    */
    fh_fputs := ora_ffi.register_function(lh_libc, 'fputs');
    if ora_ffi.is_null_ptr(fh_fputs) then
        raise VALUE_ERROR;
    end if;
    ora_ffi.register_return(fh_fputs, ORA_FFI.C_VOID);
    ora_ffi.register_parameter(fh_fputs, ORA_FFI.C_CHAR_PTR);
    ora_ffi.register_parameter(fh_fputs, ORA_FFI.C_DVOID_PTR);

END ora_pipe_io;
```

6.3 Using the Open API to Build and Modify Form Builder Applications

This section describes the non-interactive, programmatic method for building and modifying Form Builder applications. It includes these topics:

- Section 6.3.1, "What is the Open API?"
- Section 6.3.2, "Guidelines for using the Open API"
- Section 6.3.3, "Using the Open API"
- Section 6.3.4, "Open API examples"

6.3.1 What is the Open API?

The Open API is an advanced Form Builder feature for C/C++ developers that want the power and flexibility to create or modify form modules in a non-interactive environment.

Note: Before using the Open API, you should have a thorough understanding of Form Builder objects and their properties and relations.

6.3.1.1 When should I use the Open API?

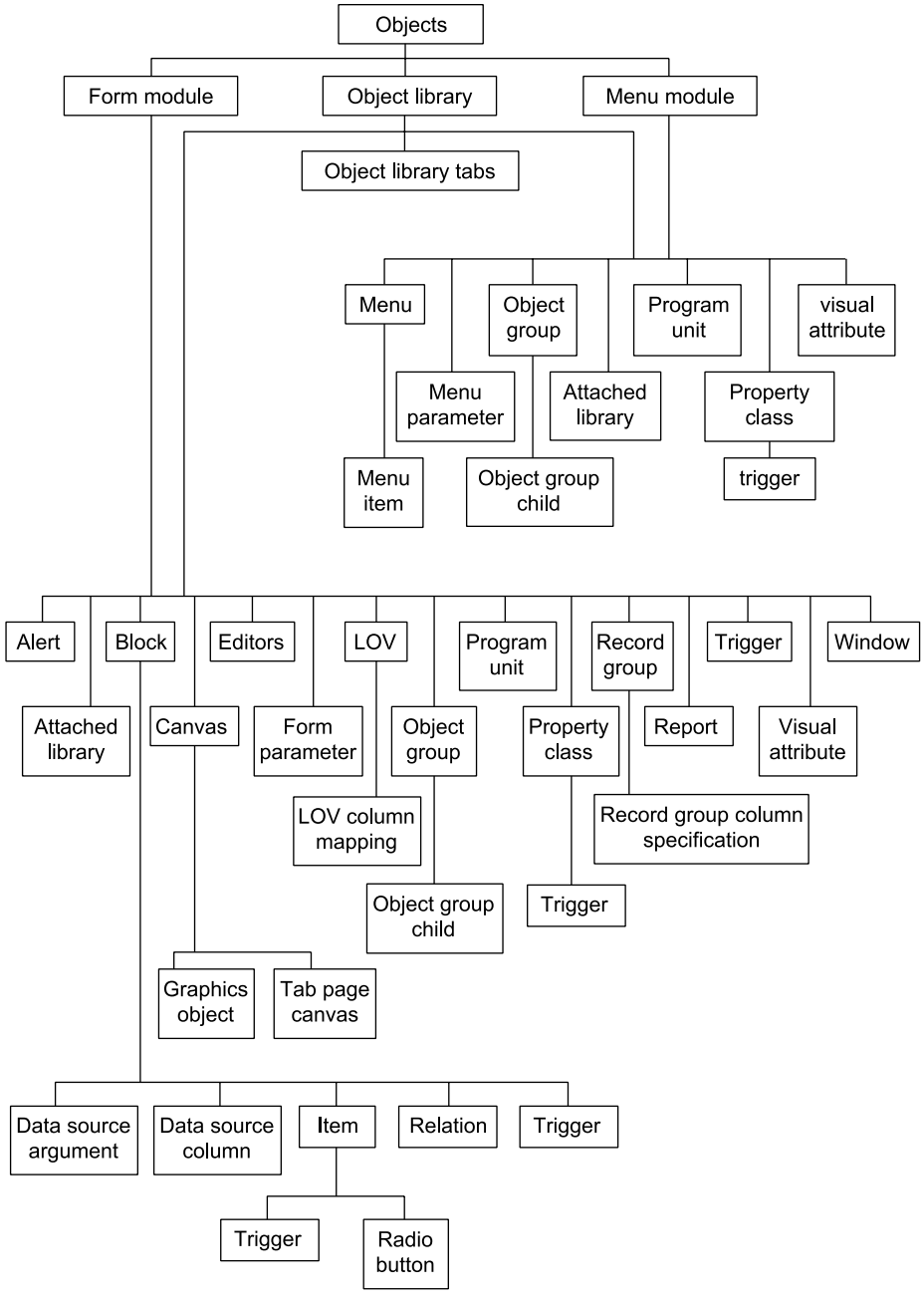
Use the Open API when you want to quickly propagate development changes to a large number of form modules. You might, for example, use the Open API to update your applications to the current corporate standards for look and feel. This could involve updating hundreds for form modules.

You can also use the Open API to:

- Compile a set of forms
- Collect dependency information
- Write your own documentation

6.3.1.2 Open API header files

The Open API consists of one C header file for each Form Builder object. There are 34 Form Builder objects (see the figure). These objects correspond to the Form Builder objects that you are familiar with at design-time. Each header file contains several functions and macros that you use to create and manipulate Form Builder objects.


Objects
Form module
Object library
Menu module
Object library tabs
Menu
Object group
Program unit
visual attribute
Menu parameter
Attached library
Property class
Menu item
Object group child
trigger
Alert
Block
Editors
LOV
Program unit
Record group
Trigger
Window
Attached library
Canvas
Form parameter
Object group
Property class
Report
Visual attribute
LOV column mapping
Record group column specification
Object group child
Trigger
Graphics object
Tab page canvas
Data source argument
Data source column
Item
Relation
Trigger
Trigger
Radio button

6.3.1.3 Open API properties

Within the Open API, you manipulate Form Builder objects by setting object properties.

Open API properties have their own unique names, such as D2FP_FONT_NAM. These properties correspond to the Form Builder properties that you are familiar with at design-time.

A property can be one of the following: Boolean, Text, Number, Object, or Blob.

The table below lists some common item properties with their corresponding Open API equivalents.

Open API Property	Form Builder (design-time) Property
D2FP_ACCESS_KEY	Access Key
D2FP_BEVEL_STY	Bevel
D2FP_CNV_NAM	Canvas
D2FP_ENABLED	Enabled
D2FP_FONT_NAM	Font Name
D2FP_HEIGHT	Width / Height
D2FP_X_POS	X Position
D2FP_Y_POS	Y Position

6.3.1.4 Open API functions and macros

You use Open API functions and macros to create, destroy, duplicate, subclass, get, and set object properties.

For example, to determine an item's font size, use the D2FITMG_FONT_SIZ macro:

```
d2fitmg_font_siz(ctx, obj, val);
```

This macro returns the value of the Font Size property of the item object as type number.

To set a text item property, use the D2FITMST_SETTEXTPROP function:

```
d2fitmst_SetTextProp(d2fctx *pd2fctx, d2fitm *pd2fitm, ub2 pnum,text *prp );
```

This function sets the value of the specified item text property. You specify a pointer to the context in `pd2fctx`, the item in `pd2fitm`, the property number in `pnum`, and a handle to the text value in `prp`.

6.3.2 Guidelines for using the Open API

When working with the Open API, consider these guidelines:

Item	Recommendation
File Backups	The Open API is non-interactive; validation and error checking are not supported. Before using the Open API, you should backup your form modules (`.FMB`s).
Creating a relation object	When creating a relation object, you must: ■ Create the object. ■ Set relation object properties. ■ Call the `d2frelup_Update` function to instantiate the object.

6.3.3 Using the Open API

This section provides detailed steps that describe how to create and modify Form Builder modules using the Open API.

6.3.3.1 Creating and modifying modules using the Open API

To create or modify a Form Builder module:

1. Include the appropriate C header files in your C source code.
2. Make calls to the desired APIs in your C source code.
   - Initialize the context structure.
   - Make `load` function calls to open an existing form module, menu module, or object library.
   - Make the necessary Open Forms API function calls to perform the desired operations, including connecting to an existing database, if required.
   - Generate an `.FMX` or `.MMX` compiled form using the appropriate `CompileFile()` function.

- Make the required function calls to save the associated module (for example, `d2ffmdsv_Save()` for a form module, `d2fmmdsv_Save()` for a menu module, or `d2folbsv_Save()` for an object library).
- Finally, call the context destroy function, `d2fctxde_Destroy()`, to destroy the Open Forms API context. Note that this function call must be your final one.

3. Link your source files against the Open API library (`ifd2f60.lib`).
4. Compile the files to create an executable (`.EXE` file).
5. Run the executable to create or modify your Form modules (`.FMB`).

6.3.4 Open API examples

This section includes several examples that describe how to use the Open API.

6.3.4.1 Modifying modules using the Open API

```
/*
This example determines if the Form Builder object is a subclassed object and
returns the file path of the parent to NULL if the object is subclassed. This
sample only processes the following object types:  form level triggers, alerts,
blocks, items, item level triggers, radio buttons, and block level triggers.
Use a similar method to process other object types.
*/
#include <stdio.h>
#include <string.h>
#include <windows.h>
#include <d2ferr.h>
#include <d2fctx.h>
#include <d2ffmd.h>
#include <d2fblk.h>
#include <d2fitm.h>
#include <d2falt.h>
#include <d2ftrg.h>
#include <d2frdb.h>
#define BUFSIZE 128
int WINAPI WinMain(HANDLE hInstance,
                   HANDLE hPrevInstance,
                   LPSTR lpszCommandLine,
                   int cmdShow)
{
    d2fctx*   pd2fctx;
    d2ffmd*   pd2ffmd;
```

```
d2fblk*   pd2fblk;
d2fitm*   pd2fitm;
d2fctxa   d2fctx_attr;
d2fstatus status;
d2falt*   pd2falt;
d2ftrg*   pd2ftrg;
d2frdb*   pd2frdb;
int counter;
char buf[BUFSIZE];
char* form_name=(char*)0;
/* Get the form name from the command line */
strncpy(buf, lpszCommandLine, BUFSIZE);
form_name = strtok(buf, ".");
/* Initialize the attribute mask */
d2fctx_attr.mask_d2fctxa = 0;
/* for MS Windows-only attributes */
d2fctx_attr.d2fihnd_d2fctxa = hInstance;
d2fctx_attr.d2fphnd_d2fctxa = hPrevInstance;
d2fctx_attr.d2fcmsh_d2fctxa = cmdShow;
/* Create the API context */
status = d2fctxcr_Create(&pd2fctx, &d2fctx_attr);
/* Load the form */
status = d2ffmdld_Load(pd2fctx, &pd2ffmd, form_name, FALSE) ;
if (status == D2FS_D2FS_SUCCESS)
{
  /*** Process Form Level Trigger Objects ***/
  for(status = d2ffmdg_trigger(pd2fctx,pd2ffmd,&pd2ftrg);
      pd2ftrg != NULL;
      status = d2ftrgg_next(pd2fctx,pd2ftrg,&pd2ftrg))
  {
    if (d2ftrgis_IsSubclassed(pd2fctx,pd2ftrg) == D2FS_YES)
       d2ftrgs_par_flpath(pd2fctx,pd2ftrg,NULL);
  }
  /*** Process Alert Objects ***/
  for(status = d2ffmdg_alert(pd2fctx,pd2ffmd,&pd2falt);
      pd2falt != NULL;
      status = d2faltg_next(pd2fctx,pd2falt,&pd2falt))
  {
    if (d2faltis_IsSubclassed(pd2fctx,pd2falt) == D2FS_YES)
       d2falts_par_flpath(pd2fctx,pd2falt,NULL);
  }
  /*** Process Block Objects ***/
  for(status = d2ffmdg_block(pd2fctx,pd2ffmd,&pd2fblk);
      pd2fblk != NULL;
      status = d2fblkg_next(pd2fctx,pd2fblk,&pd2fblk))
```

```
{
  if (d2fblkis_IsSubclassed(pd2fctx,pd2fblk) == D2FS_YES)
     d2fblks_par_flpath(pd2fctx,pd2fblk,NULL);
}
/* Process Item Objects */
for(status = d2fblkg_item(pd2fctx,pd2fblk,&pd2fitm);
    pd2fitm != NULL;
    status = d2fitmg_next(pd2fctx,pd2fitm,&pd2fitm))
{
  if (d2fitmis_IsSubclassed(pd2fctx,pd2fitm) == D2FS_YES)
     d2fitms_par_flpath(pd2fctx,pd2fitm,NULL);
  /* Process Item Level Trigger Objects */
   for(status = d2fitmg_trigger(pd2fctx,pd2fitm,&pd2ftrg);
       pd2ftrg != NULL;
       status = d2ftrgg_next(pd2fctx,pd2ftrg,&pd2ftrg))
   {
       if (d2ftrgis_IsSubclassed(pd2fctx,pd2ftrg)==D2FS_YES)
       {
         d2ftrgs_par_flpath(pd2fctx,pd2ftrg,NULL);
         printf("item trigger is Subclassed\n");
       }
       else if  (d2ftrgis_IsSubclassed(pd2fctx,
                                       pd2ftrg)==D2FS_NO)
       printf("item trigger is NOT Subclassed\n");
   }
/* Process Radio Button Objects *
  for(status = d2fitmg_rad_but(pd2fctx,pd2fitm,&pd2frdb);
      pd2frdb != NULL;
      status = d2frdbs_next(pd2fctx,pd2frdb,&pd2frdb))
  {
      if (d2frdbis_IsSubclassed(pd2fctx,pd2frdb)==D2FS_YES
      {
        d2frdbs_par_flpath(pd2fctx,pd2frdb,NULL);
        printf("radio button is Subclassed\n");
      }
      else if (d2frdbis_IsSubclassed(pd2fctx,
                                     pd2frdb)==D2FS_NO)
      printf("radio button is NOT Subclassed\n");
 }
}
/* Process Block Level Trigger Objects */
for(status = d2fblkg_trigger(pd2fctx,pd2fblk,&pd2ftrg);
    pd2ftrg != NULL;
    status = d2ftrgg_next(pd2fctx,pd2ftrg,&pd2ftrg))
{
```

```
          if (d2ftrgis_IsSubclassed(pd2fctx,pd2ftrg) == D2FS_YES)
          {
            d2ftrgs_par_flpath(pd2fctx,pd2ftrg,NULL);
            printf("block trigger is Subclassed\n");
          }
          else if (d2ftrgis_IsSubclassed(pd2fctx,
                                         pd2ftrg)==D2FS_NO)
          printf("block trigger is NOT Subclassed\n");
       }

      /* Save out the form */
      d2ffmdsv_Save(pd2fctx, pd2ffmd, (text *)0, FALSE) ;
      /* Generate the forms executable (fmx) */
      d2ffmdcf_CompileFile(pd2fctx, pd2ffmd ) ;
      /* Destroy the API Context */
      d2fctxde_Destroy(pd2fctx) ;
     }
}
```

6.3.4.2 Creating modules using the Open API

```
/*
This example creates a master-detail form based on the dept and emp database
tables owned by the user scott.  The master contains the following fields:
empno, ename, job, sal, and deptno.  The detail contains the following fields
deptno, dname, and loc.  The join condition is deptno.
*/
#include<stdio.h>
#include<string.h>
#include<windows.h>
#include<d2fctx.h>
#include<d2ffmd.h>
#include<d2ffpr.h>
#include<d2fob.h>
#include<d2fcnv.h>
#include<d2ftrg.h>
#include<d2blk.h>
#include<d2fitm.h>
#include<d2fwin.h>
#include<d2frel.h>
#define D2FS_SUCCESS 0
#define FAIL 1
#define BUFSIZE 128
#define WBP_TXT "null;\n"
int WINAPI WinMain(HANDLE hInstance,
```

```
                    HANDLE hPrevInstance,
                    LPSTR lpszCommandLine,
                    int cmdShow)
{
d2fctx  *pd2fctx;
d2ffmd  *pd2ffmd;
d2fcnv  *pd2fcnv;
d2fwin  *pd2fwin;
d2fblk  *pempblk;
d2fblk  *pdeptblk;
d2frel  *pd2frel;
d2fitm  *pEempnoitm;
d2fitm  *pEenameitm;
d2fitm  *pEjobitm;
d2fitm  *pEsalitm;
d2fitm  *pEdeptnoitm;
d2fitm  *pDdeptnoitm;
d2fitm  *pDdnameitm;
d2fitm  *pDlocitm;
text  *name = (text *)0;
text  *form_name = (text *)0;
d2fctxa  d2fctx_attr;
d2fstatus retval;
char buf[BUFSIZE];
/* Get form name */
strncpy(buf, "empdept", BUFSIZE);
form_name = (text*)strtok(buf, ".");
/* Initialize the attribute mask */
d2fctx_attr.mask_d2fctxa = 0;
/* for MS Windows-only attributes */
d2fctx_attr.d2fihnd_d2fctxa = hInstance;
d2fctx_attr.d2fphnd_d2fctxa = hPrevInstance;
d2fctx_attr.d2fcmsh_d2fctxa = cmdShow;
/* Create the API context */
status = d2fctxcr_Create(&pd2fctx, &d2fctx_attr);
/* Create the context */
d2fctxcn_Connect(pd2fctx, (text*)"scott/tiger@test");
/* Create the form */
d2ffmdcr_Create(pd2fctx, &pd2ffmd, form_name);
/* Create a window */
d2fwincr_Create(pd2fctx,pd2ffmd,&pd2fwin,(text*)"MYWIN");
/*** Create Canvas and set canvas-related properties ***/
/* Create a canvas */
d2fcnvcr_Create(pd2fctx, pd2ffmd, &pd2fcnv, (text*)"MYCANVAS");
/* Set viewport width */
```

```
d2fcnvs_vprt_wid(pd2fctx, pd2fcnv, 512);
/* Set viewport height */
d2fcnvs_vprt_hgt(pd2fctx, pd2fcnv, 403);
/* Set window */
dwfcnvs_wnd_obj(pd2fctx, pd2fcnv, pd2fwin);
/* Set viewport X-position */
d2fcnvs_vprt_x_pos(pd2fctx, pd2fcnv, 0);
/* Set viewport Y-position */
d2fcnvs_vprt_y_pos(pd2fctx, pd2fcnv, 0);
/* Set width */
d2fcnvs_width(pd2fctx, pd2fcnv, 538)
/* Set height */
d2fcnvs_height(pd2fctx, pd2fcnv, 403)
/*** Create Emp block and set block-related properties ***/
/* Create block */
d2fblkcr_Create(pd2fctx, pd2ffmd, &pempblk, (text*)"EMP");
/* Set to database block */
d2fblks_db_blk(pd2fctx, pempblk, TRUE);
/* Set query data source to Table */
d2fblks_qry_dat_src_typ(pd2fctx, pempblk, D2FC_QRDA_TABLE);
/* Set query data source name to EMP table */
d2fblks_qry_dat_src_nam(pd2fctx, pempblk, "EMP");
/* Set DML data source type to Table */
d2fblks_dml_dat_typ(Pd2fctx, pempblk, D2FC_DMDA_TABLE);
/* Set DML data source name to EMP table */
d2fblks_dml_dat_nam(pd2fctx, pempblk, (text*)"EMP");
/*** Create Dept block and set block-related properties ***/
/* Create block */
d2fblkcr_Create(pd2fctx, pd2ffmd, &pdeptblk, (text*)"DEPT");
/* Set to database block */
d2fblks_db_blk(pd2fctx, pdeptblk, TRUE);
/* Set query data source to Table */
d2fblks_qry_dat_src_typ(pd2fctx, pdeptblk, D2FC_QRDA_TABLE);
/* Set query data source name to EMP table */
d2fblks_qry_dat_src_nam(pd2fctx, pdeptblk, "DEPT");
/* Set DML data source type to Table */
d2fblks_dml_dat_typ(Pd2fctx, pdeptblk, D2FC_DMDA_TABLE);
/* Set DML data source name to EMP table */
d2fblks_dml_dat_nam(pd2fctx, pdeptblk, (text*)"DEPT");
/*** Create empno item and item-related properties ***/
/* Create item */
d2fitmcr_Create(pd2fctx, pempblk, &pEempnoitm, (text*)"EMPNO");
/* Set item type */
d2fitms_itm_type(pd2fctx, pEempnoitm, D2FC_ITTY_TI);
/* Set Enable property */
```

```
d2fitms_enabled(pd2fctx, pEempnoitm, TRUE);
/* Set item (keyboard) navigable property */
d2fitms_kbrd_navigable(pd2fctx, pEempnoitm, TRUE);
/* Set item Data Type property */
d2fitms_dat_typ(pd2fctx, pEempnoitm, D2FC_DATY_NUMBER);
/* Set item Max Length property */
d2fitms_max_len(pd2fctx, pEempnoitm, 6);
/* Set item Required property */
d2fitms_required(pd2fctx, pEempnoitm, TRUE);
/* Set Distance Between Records property */
d2fitms_dist_btwn_recs(pd2fctx, pEempnoitm, 0);
/* Set Database block(Database Item) property */
d2fitms_db_itm(pd2fctx, pEempnoitm, TRUE);
/* Set Query Allowed */
d2fitms_qry_allowed(pd2fctx, pEempnoitm, TRUE);
/* Set Query Length */
d2fitms_qry_len(pd2fctx, pEempnoitm, 6);
/* Set Update Allowed */
d2fitms_updt_allowed(pd2fctx, pEempnoitm, TRUE);
/* Set Item Displayed (Visible) */
d2fitms_visible(pd2fctx, pEempnoitm, TRUE);
/* Set Item Canvas property */
d2fitms_cnv_obj(pd2fctx, pEempnoitm, pd2fcnv);
/* Set Item X-position */
d2fitms_x_pos(pd2fctx, pEempnoitm, 32);
/* Set Item Y-position */
d2fitms_y_pos(pd2fctx, pEempnoitm, 50);
/* Set Item Width */
d2fitms_width(pd2fctx, pEempnoitm, 51);
/* Set Item Height */
d2fitms_height(pd2fctx, pEempnoitm, 17);
/* Set Item Bevel */
d2fitms_bevel(pd2fctx, pEempnoitm, D2FC_BEST_LOWERED);
/* Set item Hint */
d2fitms_hint(pd2fctx, PEempnoitm, (text*)"Enter value for :EMPNO");
/*** Create Ename item and item-related properties ***/
/* Create item */
d2fitmcr_Create(pd2fctx, pempblk, &pEenameitm, (text*)"ENAME");
/* Set item type */
d2fitms_itm_type(pd2fctx, pEenameitm, D2FC_ITTY_TI);
/* Set Enable property */
d2fitms_enabled(pd2fctx, pEenameitm, TRUE);
/* Set item (keyboard) navigable property */
d2fitms_kbrd_navigable(pd2fctx, pEenameitm, TRUE);
/* Set item Data Type property */
```

```
d2fitms_dat_typ(pd2fctx, pEenameitm, D2FC_DATY_CHAR);
/* Set item Max Length property */
d2fitms_max_len(pd2fctx, pEenameitm, 10);
/* Set Distance Between Records property */
d2fitms_dist_btwn_recs(pd2fctx, pEenameitm, 0);
/* Set Database block(Database Item) property */
d2fitms_db_itm(pd2fctx, pEenameitm, TRUE);
/* Set Query Allowed */
d2fitms_qry_allowed(pd2fctx, pEenameitm, TRUE);
/* Set Query Length */
d2fitms_qry_len(pd2fctx, pEenameitm, 10);
/* Set Update Allowed */
d2fitms_updt_allowed(pd2fctx, pEenameitm, TRUE);
/* Set Item Displayed (Visible) */
d2fitms_visible(pd2fctx, pEenameitm, TRUE);
/* Set Item Canvas property */
d2fitms_cnv_obj(pd2fctx, pEenameitm, pd2fcnv);
/* Set Item X-position */
d2fitms_x_pos(pd2fctx, pEenameitm, 83);
/* Set Item Y-position */
d2fitms_y_pos(pd2fctx, pEenameitm, 50);
/* Set Item Width */
d2fitms_width(pd2fctx, pEenameitm, 77);
/* Set Item Height */
d2fitms_height(pd2fctx, pEenameitm, 17);
/* Set Item Bevel */
d2fitms_bevel(pd2fctx, pEenameitm, D2FC_BEST_LOWERED);
/* Set item Hint */
d2fitms_hint(pd2fctx, PEenameitm, (text*)"Enter value for :ENAME");
/*** Create JOB item and item-related properties ***/
/* Create item */
d2fitmcr_Create(pd2fctx, pempblk, &pEjobitm, (text*)"JOB");
/* Set item type */
d2fitms_itm_type(pd2fctx, pEjobitm, D2FC_ITTY_TI);
/* Set Enable property */
d2fitms_enabled(pd2fctx, pEjobitm, TRUE);
/* Set item (keyboard) navigable property */
d2fitms_kbrd_navigable(pd2fctx, pEjobitm, TRUE);
/* Set item Data Type property */
d2fitms_dat_typ(pd2fctx, pEjobitm, D2FC_DATY_CHAR);
/* Set item Max Length property */
d2fitms_max_len(pd2fctx, pEjobitm, 9);
/* Set Distance Between Records property */
d2fitms_dist_btwn_recs(pd2fctx, pEjobitm, 0);
/* Set Database block(Database Item) property */
```

```
d2fitms_db_itm(pd2fctx, pEjobitm, TRUE);
/* Set Query Allowed */
d2fitms_qry_allowed(pd2fctx, pEjobitm, TRUE);
/* Set Query Length */
d2fitms_qry_len(pd2fctx, pEjobitm, 9);
/* Set Update Allowed */
d2fitms_updt_allowed(pd2fctx, pEjobitm, TRUE);
/* Set Item Displayed (Visible) */
d2fitms_visible(pd2fctx, pEjobitm, TRUE);
/* Set Item Canvas property */
d2fitms_cnv_obj(pd2fctx, pEjobitm, pd2fcnv);
/* Set Item X-position */
d2fitms_x_pos(pd2fctx, pEjobitm, 160);
/* Set Item Y-position */
d2fitms_y_pos(pd2fctx, pEjobitm, 50);
/* Set Item Width */
d2fitms_width(pd2fctx, pEjobitm, 70);
/* Set Item Height */
d2fitms_height(pd2fctx, pEjobitm, 17);
/* Set Item Bevel */
d2fitms_bevel(pd2fctx, pEjobitm, D2FC_BEST_LOWERED);
/* Set item Hint */
d2fitms_hint(pd2fctx, PEjobitm, (text*)"Enter value for :JOB");
/*** Create SALARY item and item-related properties ***/
/* Create item */
d2fitmcr_Create(pd2fctx, pempblk, &pEsalitm, (text*)"SAL");
/* Set item type */
d2fitms_itm_type(pd2fctx, pEsalitm, D2FC_ITTY_TI);
/* Set Enable property */
d2fitms_enabled(pd2fctx, pEsalitm, TRUE);
/* Set item (keyboard) navigable property */
d2fitms_kbrd_navigable(pd2fctx, pEsalitm, TRUE);
/* Set item Data Type property */
d2fitms_dat_typ(pd2fctx, pEsalitm, D2FC_DATY_NUMBER);
/* Set item Max Length property */
d2fitms_max_len(pd2fctx, pEsalitm, 9);
/* Set Distance Between Records property */
d2fitms_dist_btwn_recs(pd2fctx, pEsalitm, 0);
/* Set Database block(Database Item) property */
d2fitms_db_itm(pd2fctx, pEsalitm, TRUE);
/* Set Query Allowed */
d2fitms_qry_allowed(pd2fctx, pEsalitm, TRUE);
/* Set Query Length */
d2fitms_qry_len(pd2fctx, pEsalitm, 9);
/* Set Update Allowed */
```

```
d2fitms_updt_allowed(pd2fctx, pEsalitm, TRUE);
/* Set Item Displayed (Visible) */
d2fitms_visible(pd2fctx, pEsalitm, TRUE);
/* Set Item Canvas property */
d2fitms_cnv_obj(pd2fctx, pEsalitm, pd2fcnv);
/* Set Item X-position */
d2fitms_x_pos(pd2fctx, pEsalitm, 352);
/* Set Item Y-position */
d2fitms_y_pos(pd2fctx, pEsalitm, 50);
/* Set Item Width */
d2fitms_width(pd2fctx, pEsalitm, 70);
/* Set Item Height */
d2fitms_height(pd2fctx, pEsalitm, 17);
/* Set Item Bevel */
d2fitms_bevel(pd2fctx, pEsalitm, D2FC_BEST_LOWERED);
/* Set item Hint */
d2fitms_hint(pd2fctx, PEsalitm, (text*)"Enter value for :SAL");
/*** Create DEPTNO item and item-related properties ***/
/* Create item */
d2fitmcr_Create(pd2fctx, pempblk, &pEdeptnoitm, (text*)"DEPTNO");
/* Set item type */
d2fitms_itm_type(pd2fctx, pEdeptnoitm, D2FC_ITTY_TI);
/* Set Enable property */
d2fitms_enabled(pd2fctx, pEdeptnoitm, TRUE);
/* Set item (keyboard) navigable property */
d2fitms_kbrd_navigable(pd2fctx, pEdeptnoitm, TRUE);
/* Set item Data Type property */
d2fitms_dat_typ(pd2fctx, pEdeptnoitm, D2FC_DATY_NUMBER);
/* Set item Max Length property */
d2fitms_max_len(pd2fctx, pEdeptnoitm, 4);
/*Set item Required property */
d2fitms_required(pd2fctx, pEdeptnoitm, TRUE);
/* Set Distance Between Records property */
d2fitms_dist_btwn_recs(pd2fctx, pEdeptnoitm, 0);
/* Set Database block(Database Item) property */
d2fitms_db_itm(pd2fctx, pEdeptnoitm, TRUE);
/* Set Query Allowed */
d2fitms_qry_allowed(pd2fctx, pEdeptnoitm, TRUE);
/* Set Query Length */
d2fitms_qry_len(pd2fctx, pEdeptnoitm, 4);
/* Set Update Allowed */
d2fitms_updt_allowed(pd2fctx, pEdeptnoitm, TRUE);
/* Set Item Displayed (Visible) */
d2fitms_visible(pd2fctx, pEdeptnoitm, TRUE);
/* Set Item Canvas property */
```

```
d2fitms_cnv_obj(pd2fctx, pEdeptnoitm, pd2fcnv);
/* Set Item X-position */
d2fitms_x_pos(pd2fctx, pEdeptnoitm, 493);
/* Set Item Y-position */
d2fitms_y_pos(pd2fctx, pEdeptnoitm, 50);
/* Set Item Width */
d2fitms_width(pd2fctx, pEdeptnoitm, 30);
/* Set Item Height */
d2fitms_height(pd2fctx, pEdeptnoitm, 17);
/* Set Item Bevel */
d2fitms_bevel(pd2fctx, pEdeptnoitm, D2FC_BEST_LOWERED);
/* Set item Hint */
d2fitms_hint(pd2fctx, PEdeptnoitm, (text*)"Enter value for :DEPTNO");
/*** Create DEPTNO item and item-related properties ***/
/* Create item */
d2fitmcr_Create(pd2fctx, pdeptblk, &pDdeptnoitm, (text*)"DEPTNO");
/* Set item type */
d2fitms_itm_type(pd2fctx, pDdeptnoitm, D2FC_ITTY_TI);
/* Set Enable property */
d2fitms_enabled(pd2fctx, pDdeptnoitm, TRUE);
/* Set item (keyboard) navigable property */
d2fitms_kbrd_navigable(pd2fctx, pDdeptnoitm, TRUE);
/* Set item Data Type property */
d2fitms_dat_typ(pd2fctx, pDdeptnoitm, D2FC_DATY_NUMBER);
/* Set item Max Length property */
d2fitms_max_len(pd2fctx, pDdeptnoitm, 4);
/*Set item Required property */
d2fitms_required(pd2fctx, pDdeptnoitm, TRUE);
/* Set Distance Between Records property */
d2fitms_dist_btwn_recs(pd2fctx, pDdeptnoitm, 0);
/* Set Database block(Database Item) property */
d2fitms_db_itm(pd2fctx, pDdeptnoitm, TRUE);
/* Set Query Allowed */
d2fitms_qry_allowed(pd2fctx, pDdeptnoitm, TRUE);
/* Set Query Length */
d2fitms_qry_len(pd2fctx, pDdeptnoitm, 4);
/* Set Update Allowed */
d2fitms_updt_allowed(pd2fctx, pDdeptnoitm, TRUE);
/* Set Item Displayed (Visible) */
d2fitms_visible(pd2fctx, pDdeptnoitm, TRUE);
/* Set Item Canvas property */
d2fitms_cnv_obj(pd2fctx, pDdeptnoitm, pd2fcnv);
/* Set Item X-position */
d2fitms_x_pos(pd2fctx, pDdeptnoitm, 32);
/* Set Item Y-position */
```

```
d2fitms_y_pos(pd2fctx, pDdeptnoitm, 151);
/* Set Item Width */
d2fitms_width(pd2fctx, pDdeptnoitm, 38);
/* Set Item Height */
d2fitms_height(pd2fctx, pDdeptnoitm, 17);
/* Set Item Bevel */
d2fitms_bevel(pd2fctx, pDdeptnoitm, D2FC_BEST_LOWERED);
/* Set item Hint */
d2fitms_hint(pd2fctx, PDdeptnoitm, (text*)"Enter value for :DEPTNO");
/*** Create DNAME item and item-related properties ***/
/* Create item */
d2fitmcr_Create(pd2fctx, pdeptblk, &pDdnameitm, (text*)"DNAME");
/* Set item type */
d2fitms_itm_type(pd2fctx, pDdnameitm, D2FC_ITTY_TI);
/* Set Enable property */
d2fitms_enabled(pd2fctx, pDdnameitm, TRUE);
/* Set item (keyboard) navigable property */
d2fitms_kbrd_navigable(pd2fctx, pDdnameitm, TRUE);
/* Set item Data Type property */
d2fitms_dat_typ(pd2fctx, pDdnameitm, D2FC_DATY_CHAR);
/* Set item Max Length property */
d2fitms_max_len(pd2fctx, pDdnameitm, 14);
/* Set Distance Between Records property */
d2fitms_dist_btwn_recs(pd2fctx, pDdnameitm, 0);
/* Set Database block(Database Item) property */
d2fitms_db_itm(pd2fctx, pDdnameitm, TRUE);
/* Set Query Allowed */
d2fitms_qry_allowed(pd2fctx, pDdnameitm, TRUE);
/* Set Query Length */
d2fitms_qry_len(pd2fctx, pDdnameitm, 14);
/* Set Update Allowed */
d2fitms_updt_allowed(pd2fctx, pDdnameitm, TRUE);
/* Set Item Displayed (Visible) */
d2fitms_visible(pd2fctx, pDdnameitm, TRUE);
/* Set Item Canvas property */
d2fitms_cnv_obj(pd2fctx, pDdnameitm, pd2fcnv);
/* Set Item X-position */
d2fitms_x_pos(pd2fctx, pDdnameitm, 70);
/* Set Item Y-position */
d2fitms_y_pos(pd2fctx, pDdnameitm, 151);
/* Set Item Width */
d2fitms_width(pd2fctx, pDdnameitm, 102);
/* Set Item Height */
d2fitms_height(pd2fctx, pDdnameitm, 17);
/* Set Item Bevel */
```

```
d2fitms_bevel(pd2fctx, pDdnameitm, D2FC_BEST_LOWERED);
/* Set item Hint */
d2fitms_hint(pd2fctx, PDdnameitm, (text*)"Enter value for :DNAME");
/*** Create LOC item and item-related properties ***/
/* Create item */
d2fitmcr_Create(pd2fctx, pdeptblk, &pDlocitm, (text*)"LOC");
/* Set item type */
d2fitms_itm_type(pd2fctx, pDlocitm, D2FC_ITTY_TI);
/* Set Enable property */
d2fitms_enabled(pd2fctx, pDlocitm, TRUE);
/* Set item (keyboard) navigable property */
d2fitms_kbrd_navigable(pd2fctx, pDlocitm, TRUE);
/* Set item Data Type property */
d2fitms_dat_typ(pd2fctx, pDlocitm, D2FC_DATY_CHAR);
/* Set item Max Length property */
d2fitms_max_len(pd2fctx, pDlocitm, 13);
/* Set Distance Between Records property */
d2fitms_dist_btwn_recs(pd2fctx, pDlocitm, 0);
/* Set Database block(Database Item) property */
d2fitms_db_itm(pd2fctx, pDlocitm, TRUE);
/* Set Query Allowed */
d2fitms_qry_allowed(pd2fctx, pDlocitm, TRUE);
/* Set Query Length */
d2fitms_qry_len(pd2fctx, pDlocitm, 13);
/* Set Update Allowed */
d2fitms_updt_allowed(pd2fctx, pDlocitm, TRUE);
/* Set Item Displayed (Visible) */
d2fitms_visible(pd2fctx, pDlocitm, TRUE);
/* Set Item Canvas property */
d2fitms_cnv_obj(pd2fctx, pDlocitm, pd2fcnv);
/* Set Item X-position */
d2fitms_x_pos(pd2fctx, pDlocitm, 173);
/* Set Item Y-position */
d2fitms_y_pos(pd2fctx, pDlocitm, 151);
/* Set Item Width */
d2fitms_width(pd2fctx, pDlocitm, 96);
/* Set Item Height */
d2fitms_height(pd2fctx, pDlocitm, 17);
/* Set Item Bevel */
d2fitms_bevel(pd2fctx, pDlocitm, D2FC_BEST_LOWERED);
/* Set item Hint */
d2fitms_hint(pd2fctx, PDlocitm, (text*)"Enter value for :LOC");
/*** Create Relations and relations-related properties ***/
/* Create Relation */
d2frelcr_Create(pd2fctx, (d2fob *)pdeptblk, &pd2frel, (text*)"DEPT_EMP");
```

```
/* Set Relation Detail block */
d2frels_detail_blk(pd2fctx, pd2frel, (text *)"EMP");
/* Set Master Deletes property */
d2frels_del_rec([pd2fctx, pd2frel, D2FC_DERE_NON_ISOLATED);
/* Set Deferred property */
d2frels_deferred(pd2ctx, pd2frel, FALSE);
/* Set Auto Query property */
d2frels_auto_qry(pd2ctx, pd2frel, FALSE);
/* Set Prevent Masterless property */
d2frels_prvnt_mstrless_ops(pd2ctx, pd2frel, FALSE);
/* Set Join Condition property */
d2frels_join_cond(pd2ctx, pd2frel, (text*)"DEPTNO");
/* Instantiate Relation: creates master-detail triggers */
d2frelup_Update(pd2fctx, pd2frel);
/* Save Form */
d2ffmdsv_Save(pd2fctx, pd2ffmd, (text*)0, FALSE, TRUE);
/* Compile Form */
d2ffmdcf_CompileFile(pd2fctx, pd2ffmd);
/* Destroy Context */
d2fctxde_Destroy(pd2fctx);
}
```

6.4 Designing Applications to Run against ODBC Datasources

The data within your enterprise often resides within several heterogeneous datasources. Some portion of your data, for example, might be stored within an Oracle database, while another portion is stored within an Informix database. Building a single application that can access each datasource can be a difficult task.

However, by taking advantage of Forms Developer's and Reports Developer's open datasource support, you can build generic applications that run transparently against any ODBC-compliant datasource.

This section describes open datasource support. It includes these topics:

- Section 6.4.1, "What is the Oracle Open Client Adapter (OCA)?"
- Section 6.4.2, "Open datasource guidelines"
- Section 6.4.3, "Configuring your application to run against an ODBC datasource"

6.4.1 What is the Oracle Open Client Adapter (OCA)?

When you connect to an ODBC datasource, you use the Oracle Open Client Adapter (OCA). OCA is an ODBC level 2-compliant utility that allows Forms Developer or Reports Developer on Microsoft Windows 95, Windows NT, and Windows 3.1 to access ODBC -compliant datasources through ODBC drivers.

OCA is included with both Forms Developer and Reports Developer. You use the Oracle Installer to install OCA.

6.4.1.1 When should I use OCA?

You should use OCA whenever your application must access non-Oracle datasources. Your Form and Report applications can automatically access any ODBC-compliant datasources. Refer to the online help for specific information about connecting to ODBC datasources.

6.4.1.2 OCA architecture

The Oracle Open Client Adapter consists of the following:

Component	Description
Forms Developer or Reports Developer Application	Performs processing and calls ODBC functions to submit SQL statements and retrieve results.
Oracle Open Client Adapter	Translates Oracle database calls to ODBC calls.
Driver Manager	Loads ODBC drivers for your application.
ODBC Drivers	Process ODBC function calls, submits SQL requests to a specific datasource, and returns results to your application.
Datasource	Consists of the data that user wants to access and its associated operating system, DBMS, and network platform (if any) used to access the DBMS.

6.4.1.3 Establishing an ODBC connection

To connect to an ODBC datasource, type the following connect string in the Connect dialog box:

```
[user[/password]]@ODBC:datasource[:dbname]
```

For example, to connect to Sybase System 10, type:

```
scott/tiger@ODBC:sybase_ds
```

6.4.1.4 ODBC drivers

When you connect to an ODBC datasource, you use an ODBC driver to communicate with the datasource. Both Forms Developer and Reports Developer include prebundled ODBC drivers for each supported datasource. These drivers are ODBC level 1-compliant and, to some extent, provide some level 2 functionality to achieve greater performance.

6.4.1.5 OPENDB.PLL

`OPENDB.PLL` is a PL/SQL library of functions that is included with OCA. You use `OPENDB.PLL` within applications to:

- Automatically adjust form and data block properties at runtime to suit the datasource.

- Open auxiliary connections to other datasources in addition to the application's main connection.
- Execute arbitrary SQL statements and Stored Procedure calls on any connection.
- Retrieve results sets from non-Oracle stored procedures.
- Obtain the DBMS and ODBC driver names and versions for a given connection.

For more information about OPENDB.PLL, refer to OCA_INFO.PDF in the ORACLE_HOME\TOOLS\DOC20 directory.

6.4.2 Open datasource guidelines

When working with multiple datasources, consider these guidelines:

Topic	Recommendation
Optimizing your application to run against multiple datasources	You do not have to optimize your application to run against multiple datasources unless you want to target a specific datasource to take advantage of features particular to that system.
Writing PL/SQL for use with ODBC datasources	SQL statements embedded in PL/SQL program units must conform to both Oracle SQL and the SQL dialect of the datasource that you connect against. Any statements that fail against Oracle will cause PL/SQL compilation failures. Similarly, any statements that use unsupported syntax will fail at execution. The SYSDATE and USER functions are the only exceptions to this restriction. These functions are Oracle-specific; and, OCA translates these functions to the corresponding ODBC functions, allowing these functions to work against all datasources. If you want to issue SQL statements that are datasource-specific, but conflict with Oracle syntax, use the EXEC_SQL package.

Topic	Recommendation
Referencing tables from more than one datasource	Many datasources allow you to access tables that are located in other datasources if you specify the database, owner, and table (for example, database.owner.tablename). PL/SQL does not recognize the three-part table syntax, and your client-side program unit or trigger will not compile. To work around this restriction, enclose the three-part name in double quotes, after calling the appropriate `OPENDB` function that removes double quotes.
Restrictions	■ When working with a non-Oracle7 datasource, you must store your application modules (forms, reports, and graphics) in the file system. Non-Oracle7 datasources cannot be used as a repository for storing application modules. ■ Trigger information for columns cannot be accessed from the Object Navigator (Database Objects node). ■ You can view stored procedure text only for datasources that emulate the Oracle ALL_SOURCE table (e.g., Microsoft SQL Server). You cannot edit database stored procedure text. ■ You cannot drag and drop PL/SQL program units from the client to a non-Oracle7 datasource. ■ Neither Forms Developer nor Reports Developer can use primary and foreign key constraint information of OCA datasources for default selection of master-detail relationships. These relationships must be identified directly where required. ■ Optimizer hints (/*hint*/ style comments) are ignored by any datasource that you connect to through OCA.

Topic	Recommendation
Troubleshooting	To view the SQL statements issued by OCA and the messages generated by the ODBC driver or the database: **1.** Verify that the following entry is set in the ORACLE.INI file on Windows 3.1 or in the registry on Windows NT and Windows 95: UB=ORACLE_HOME\OCA20 **2.** If you are unable to resolve the error, call Oracle Customer Support. **3.** Add the following entries to the ORACLE.INI file on Windows 3.1 or to the registry under SOFTWARE\ORACLE on Windows NT and Windows 95: OCA_DEBUG_SQL=TRUE OCA_DEBUG_ERROR=TRUE **4.** Run your application against the ODBC datasource to view SQL statements or error messages in the debug window. Click OK to close the debug window and continue processing.
Debugging Tips	You can display debug information by setting the OCA_DEBUG_SQL and OCA_DEBUG_ERROR environment variables to TRUE. Using these environment variables will help you identify SQL failures when using Forms Developer or Reports Developer against OCA datasources. When you set OCA_DEBUG to TRUE, any SQL statement that is sent to the ODBC driver is displayed before it is transmitted. When you set OCA_DEBUG_ERROR to TRUE, any errors that are returned by the ODBC driver are displayed in a dialog before being passed back to Forms or Reports.

6.4.3 Configuring your application to run against an ODBC datasource

To configure your application to run against an ODBC-compliant datasource, refer to the "Accessing non-Oracle datasources" topic in the online help.

Glossary

action

In Project Builder, a command string supplied either by Project Builder or by the user that applies to files of a given type or types. An action is not restricted to a single file type; for example, if the action "Compile" is defined for both forms and C source files, selecting the menu item **Compile Project** will compile all .FMB and .C files using the appropriate tools. See also: *pre-defined action*, *user-defined action*.

applet

A Java term for small programs that can be dynamically imported into Web pages or applications as needed.

bidirectional support

Support for languages whose natural writing direction is right-to-left, for example Middle Eastern and North African languages.

block

The representation of an entity on a form.

built-in macro

In Project Builder, a macro shipped with Project Builder. See also: *macro*.

canvas

The surface on which interface items and prompts are drawn. Canvasses are displayed in a window.

CGI — Common Gateway Interface

The industry-standard technique for running applications on a Web server. Whereas standard HTML documents retrieved from a Web server are static (the exact same text is retrieved every time) CGI enables a program running on the Web server to communicate with another computer to generate "dynamic" HTML documents in response to user-entered information.

character set

Encoding scheme in which each character is represented by a different binary value. For example, ISO8859-1 is an extended Latin character set that supports more than 40 Western European languages.

deliver

In Project Builder, to prepare and provide a completed application for distribution and deployment.

dependency view

In Project Builder, a view that shows the files in the Project Navigator in the order in which they depend on each other, with project nodes at the highest point in the hierarchy, followed by target nodes, which are followed by buildable components of those targets. For example, an executable form depends on and will be followed by an *.fmb* file, which may depend on and be followed by a library used for a USER-EXIT procedure, and so on. See also: *project view, target*.

dialog box

A window used to enter information needed to complete a specific action. The user must interact with this window before proceeding.

encryption

The practice of scrambling (encrypting) data in such a way that only an intended recipient can unscramble (decrypt) and read the data.

entity

A thing of significance to the user. ‘Assignments’ and ‘Sales Order Lines’ are examples of entities. A single entity may comprise several blocks, such as ‘Sales Rep’, ‘Quotas’, and ‘Territories’.

export

In Project Builder, the process of writing out a file containing project, type, action, and/or macro definitions in a portable format for distribution to others who may work on heterogeneous platforms. See also: *export file*, *import*.

export file

In Project Builder, the shareable, portable file created by exporting a project. The default extension of an export file is `.UPX`. See also: *export*, *import*.

field

An interface element that displays information to the user and/or accepts input from the user. Text items, check boxes, and poplists are examples of fields. Also known as 'widget' or 'item'.

firewall

A computer that regulates access to computers on a local area network from outside, and regulates access to outside computers from within the local area network.

format mask

A setting that defines the appearance of the value of a field. For example, a format mask is used to specify the display of currency amounts and dates.

Global Registry

A Project Builder registry that stores information common to an entire Forms Developer or Reports Developer installation. This information is restricted to type definitions and their associated actions and pre-defined or user-defined properties. The use of the Global Registry is optional; its functions can be performed by individual user registries. See also: *registry*, *user registry*.

group

In Project Builder, collections of related items available via submenus off the Launcher. Groups enable users to set up the Launcher much like the Windows 95 Start menu, with arbitrary "groups" that pop up to reveal other items and/or groups.

GUI — Graphical User Interface

The use of pictures rather than just words to represent the input and output of a program. Programs with GUIs run under a windowing system (such as X Windows, Microsoft Windows, Apple Macintosh, and so on). GUI programs display icons, buttons, and so on, in windows on the screen; users control the GUI programs mainly by moving a pointer on the screen (typically controlled by a mouse).

HTML — Hypertext Markup Language

A tag-based ASCII language used to specify the content and hypertext links to other documents on WWW servers on the Internet. End users with Web browsers view HTML documents and follow links to display other documents.

HTTP — Hypertext Transfer Protocol

The protocol used to carry WWW traffic between a WWW browser computer and the WWW server being accessed.

hyperlink

A reference (link) from some point in one hypertext document to (some point in) another document or another place in the same document. A Web browser usually displays a hyperlink in some distinguishing way (in a different color, font or style). When users activate hyperlinks (by clicking on them with a mouse) the browser displays the target of the link.

hypertext

A collection of documents containing cross-references which, with the aid of a Web browser, allow readers to move easily from one document to another.

implied item

In Project Builder, a project item, usually the result of automatic generation, which Project Builder recognizes and for which it automatically creates an entry in the Project Navigator. For example, Project Builder can recognize `.OBJ` files, generated as an immediate step in the compilation of C source files, and create entries for them in the Project Navigator. Although resetting the properties of an implied item is of limited use (the next compilation will destroy changes) such items can be useful, as they can be examined via actions such as Edit, View, and Print. See also: *action*, *item*.

import

In Project Builder, to read in a file containing project information. This is the recommended method for sharing projects. See also: *export*, *export file*.

inherit

In Project Builder, to obtain information for an action, type, macro, or property definition from an ancestor node in the dependency tree. If related attributes exist in an ancestor node, they may be inherited. Thus, filesystem items like forms and documents may inherit action definitions from subprojects, projects, a user registry, or the Global Registry; projects may inherit type definitions from a user registry or the Global Registry; and so on.

input item

In Project Builder, the file used to build a target. For example, an .FMB is the input item for an .FMX. Also called *source*.

Internet

A worldwide TCP/IP-based network of computers.

Intranet

An internal TCP/IP network, access to which is restricted (via a firewall) to individuals inside the company or organization. An intranet provides similar services within an organization to those provided by the Internet, but is not necessarily connected to the Internet. A common example of an intranet is when a company sets up one or more Web servers on an internal network for distribution of information or applications within the company.

IP (Internet Protocol) Address

A four-part number with no more than three digits in each part that uniquely identifies a computer on the Internet.

item

In Project Builder, an object in the file system associated with a project, such as a form or report, and pointed to or represented by a node in the Project Navigator.

JAR — Java ARchive

A file used for aggregating many files (Java class files, images, and so on) into one file.

Java

A computer language that supports programming for the Internet in the form of platform-independent "applets."

language environment variable

Environment variable which specifies the language, territory, and character set for a user's environment. The language environment variable can be any one of the following: NLS_LANG, DEVELOPER_NLS_LANG, or USER_NLS_LANG.

Launcher

In Project Builder, the secondary toolbar docked (by default) to the left of the Project Navigator. It provides simple organizational and application launching abilities.

macro

In Project Builder, a type-specific variable which may be used to customize and extend actions. A macro may be either a constant or a simple expression (which may, in turn, contain other constants and / or expressions). The use of macros offers great flexibility in issuing command options, and in allowing the user to modify sets of commands by changing one property definition.

master-detail

A relation between two entities that indicates a hierarchy of information. For example, a Sales Order consists of a Header entity and a Line entity; the Header is the master of the Line, and the Line is the detail of the Header.

modal

A state where the user must supply specific information before continuing operation of the application.

multilingual application

An application which can be deployed in more than one language and displays data according to local conventions.

ORACLE_HOME

An environment variable that indicates the root of the Oracle7 Server code tree.

PDF — Portable Document Format

A file format (native for Adobe Acrobat) for representing documents in a manner that is independent of the original application software, hardware, and operating system used to create the documents. A PDF file can describe documents containing any combination of text, graphics, and images in a device-independent and resolution independent format.

PL/SQL

Oracle's proprietary extension to the SQL language. Adds procedural and other constructs to SQL that make it suitable for writing applications.

port

A number that TCP uses to route transmitted data to and from a particular program.

pre-defined action

An action shipped with Project Builder and automatically available to the user via a menu item and/or a toolbar button. Pre-defined actions include Build, Deliver, and several source control options. When a pre-defined action is defined for a supported file type, the action is invoked for any selected item of that file type when the user calls that action from Project Builder. See also: *action*, *user-defined action*.

project

The basic data structure created by Project Builder. A project is a collection of pointers to files in the user's file system. It also contains information about behavior that the user may wish to apply to a given project, such as the specific editor to invoke to edit all files of types .CPP, .H, and .TXT. Project files can be exported and shared across platforms. See also: *export*, *project definition file*, *project item*.

project definition file

In Project Builder, a file that stores project data, which consists of project items and their properties. Each file has one project item by default, which can be thought of as the "root" or master project for that file. The user can create as many subproject items as necessary in this file; subprojects are items beneath the master project which allow the user to collect subgroups of items and change their properties at the parent (subproject) level. The default extension for a project file is .UPD. See also: *project*, *project item*.

project item

In Project Builder, an item that stores project-specific information, such as a connection string and an implicit list of contained items. Types are not defined here, but actions and user-defined macros for all types visible to a project may be defined and/or overridden here. See also: *item*, *project*, *project definition file*.

Project Navigator

In Project Builder, the window containing a hierarchical list of project items for the current session. The list appears in outline form, and enables the user to complete several tasks, such as creating, editing, renaming, and deleting objects. Although only one schema is visible at any time, the user can choose from two different schema by which to organize the objects. See also: *dependency view, project view.*

project view

In Project Builder, the project view shows objects in the Project Navigator organized by their type, and by their project/subproject relationships. The projects are organized alphabetically by project file, then alphabetically by category. See also: *dependency view, Project Navigator.*

Project Wizard

In Project Builder, a dialog that assists the user in accomplishing the steps necessary to create a new project or subproject.

prompt

A label that uniquely identifies an item. 'Salesperson' and 'Item Description' are examples of prompts.

region

A set of related items within an entity. For example, the Purchase Order Header entity might contain a 'Currency Information' region, which consists of the Rate, Type, and Date fields.

registry

In Project Builder, a global and/or user-specific configuration file used to store project definitions and environment information. See also: *Global Registry, user registry.*

RDBMS — Relational Database Management System

A database that allows the definition of data structures, storage and retrieval operations, and integrity constraints. In such a database, data and relations between them are organized in tables.

snap point

The point of a widget that corresponds to the (X,Y) position that locates it.

socket

The combination of an IP address and a port number.

target

In Project Builder, any item in the middle of the dependency tree; for example, an executable is a (compile) target for a library, while a library is a target for a group of objects and an object is a target for a source file. See also: *input item*.

toolbar

A series of iconic buttons that perform generic actions, such as List and Save.

TCP — Transmission Control Protocol

The underlying communication protocol for exchanging HTTP requests between clients and Web servers.

type

In Project Builder, a description of a file type, such as a form, a document, etc., containing such information as type name and description. Types are the foundation for defining actions and macros.

URL: Uniform Resource Locator

The "address" used to specify a WWW server and home page. For example:

```
http://www.acme.com/
```

indicates that the host's address is `www.acme.com`.

An URL most often is a filename (possibly with a long path to it and usually with an extension of `.HTML`, or `.HTM` (for PC-DOS filenames).

user-defined action

In Project Builder, a custom action defined by a Project Builder user. Such actions may apply to a single file type, or all file types. See also: *action*, *pre-defined action*.

user-defined macro

In Project Builder, a custom macro defined by a Project Builder user. Such macros may be used to modify both pre-defined and user-defined actions. See also: *action*, *built-in macro*, *macro*, *pre-defined action*.

user registry

In Project Builder, a project file storing configuration information on a per-user basis. This enables users to customize their individual development environments. The user registry inherits type information from the Global Registry, and may define new types as well as override aspects of types defined in the Global Registry. It also stores environment and preference information, such as the user's preferred connection string, UI settings, and general project information. See also: *Global Registry, registry.*

virtual directory

A synonym that the virtual file system maps to a file stored in the file system maintained by the host machine's operating system.

virtual file system

A mapping that associates the pathnames used in URL to the file system maintained by the host machine's operating system.

Web browser

A program that end users utilize to read HTML documents and programs stored on a computer (serviced by a Web server).

Web cartridge

A program executed on a Web server via the WRB.

Web server

A server process (HTTP daemon) running at a Web site which sends out Web pages in response to HTTP requests from remote Web browsers

window

A screen in a graphical user interface (GUI) environment. A window is a frame enclosing a surface on which elements are painted.

WRB — Oracle Web Request Broker

Provides a powerful distributed runtime environment for developing and deploying applications for the Web. The WRB runtime platform enables application developers to write applications that are independent of, and work with a number of, Web servers.

WWW — World Wide Web

The network of servers on the Internet, each of which has one or more home pages, which provide information and hypertext links to other documents on that and (usually) other servers.

Index

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

R

S

T

U

V

W