ProjectIndexing;¬Project Indexing

When you create or open a project, after some seconds you may notice triangular ªbranchº buttons appearing after source code files in the browser. Project Builder has indexed these files.

During indexing Project Builder stores all symbols of the project (classes, methods, globals, etc.) in virtual memory. This allows Project Builder to access project-wide information quickly. Indexing is indispensable to such features as name completion and Project Find. (More on these features later.)

Usually indexing happens automatically when you create or open a project. You can turn off this option if you wish. Choose Preferences from the Info menu and then choose the Indexing display. Turn off the ªIndex when project is openedº switch.

You can also index a project at any time by choosing Index Source Code from the Project menu. If you want to do without indexing (maybe you have memory constraints), choose Purge Indices from the Project menu.

TableRule.eps ¬

AWindowinOpenStep;¬A Window in OpenStep

A window in OpenStep looks very similar to windows in other user environments such as Windows or Macintosh. It is a rectangular area on the screen in which an application displays controls, fields, text, and graphics. Windows can be moved around the screen and stacked on top of each other like pieces of paper. A typical OpenStep window has a title bar, a content area, and several control objects.

_smallWindow.eps ¬

Many user-interface objects other than the standard window depicted above are windows.

Menus, pop-up lists, and pull-down lists are primarily windows, as are all varieties of panels: attention panels, inspectors, and tool palettes, to name a few. In fact, anything drawn on the screen must appear in a window.

NSWindow and the Window Server

Two interacting systems create and manage OpenStep windows. On the one hand, a window is created by the Window Server. The Window Server is a process integrating the NeXT Window System and Display Postscript. The Window Server draws, resizes, hides, and moves windows using Postscript primitives. The Window Server also detects user events (such as mouse clicks) and forwards them to applications.

The window that the Window Server creates is paired with an object supplied by the Application Kit: an instance of the NSWindow class. Each physical window in an object-oriented program is managed by an instance of NSWindow (or subclass).

When you create an NSWindow object, the Window Server creates the physical window that the NSWindow object will manage. The Window Server references the window by its window number, the NSWindow by its own identifier.

Application, Window, View

In a running OpenStep application, NSWindow objects occupy a middle position between an instance of NSApplication and the views of the application. (A view is an object that can draw itself and detect user events.) The NSApplication object keeps a list of its windows and tracks the current status of each. Each window, on the other hand, manages a hierarchy of views in addition to its PostScript window.

WindowViews.eps ¬

At the ªtopº of this hierarchy is the *content view*, which fits just within the window's *content*

rectangle. The content view encloses all other view (its *subviews*), which come below it in the hierarchy. The NSWindow distributes events to views in the hierarchy and regulates coordinate transformations among them.

Another rectangle, the *frame rectangle* defines the outer boundary of the window and includes the title bar and the window's controls. The lower-left corner of the frame rectangle defines the window's location relative to the screen's coordinate system and establishes the base coordinate system for the views of the window. Views draw themselves in coordinate systems transformed from (and relative to) this base coordinate system.

Key and Main Windows

Windows have numerous characteristics. They can be on-screen or off-screen. On-screen windows are ªlayeredº on the screen in tiers managed by the Window Server. On-screen windows also can carry a status: *key* or *main*.

Key windows respond to key presses for an application and are the primary recipient of action messages from menus and panels. Usually a window is made key when the user clicks it. Key windows have black (or dark blue) title bars. Each application can have only one key window.

An application has one main window, which can often have key status as well. The main window is the principal focus of user actions for an application. Often user actions in a modal key window (typically a panel such as the Font panel or an inspector) has a direct effect on the main window. In this case, the title bar of the main window (when it is not key) is a dark gray.

567565_TableRule.eps ¬

AligningonaGrid;¬Aligning on a Grid

You can align objects on a window by imposing a grid on the window. When you move objects in this grid, they ªsnapº to the nearest grid intersection like nails to a magnet. You set the edges of alignment and the spacing of the grid (in pixels) in the Alignment panel. Choose Format arrow.eps ¬ Align 885211_arrow.eps ¬ Alignment to display this panel.

IB_AlignmentPanel.tiff ¬

Be sure the grid is turned on before you move objects (Format 994803_arrow.eps ¬ Align 116064_arrow.eps ¬ Turn Grid On).

You can move selected user-interface objects in Interface Builder by pressing an arrow key. When the grid is turned on the unit of movement is whatever the grid is set to (in pixels). When the grid is turned off, the unit of movement is one pixel.

809224_TableRule.eps ¬

AOpenStepApplicationÑWhatYouGetªForFreeº;¬An OpenStep Application Ð What You Get ªFor Freeº

The simplest OpenStep application, even one without a line of code added to it, includes a wealth of features that you get ªfor freeº: You do not have to program these features yourself. You can see this when you test an interface in Interface Builder.

To enter test mode, choose Test Interface from the Document menu. Interface Builder simulates how your application (in this case, Currency Converter) would run, minus the behavior added by custom classes. Go ahead and try things out: move your windows, type in fields, click buttons.

Application and Window Behavior

In test mode Currency Converter behaves almost like any other application on the screen. Click elsewhere on the screen, and Currency Converter is deactivated, becoming totally or partially obscured by the windows of other applications.

CC_Test.tiff ¬

Reactivate Currency Converter by clicking on its window or by double-clicking its icon (the

default terminal icon) in the workspace. Move the window around by its title bar.

Here's some other tests you can make:

- Click the Edit submenu in Currency Converter's main menu. It expands and contracts as in any application.
- Click the miniaturize button or choose the Hide command. Double-click the document icon to get the window back.
- Click the close box and the Currency Converter window disappears. (Choose Quit from the main menu and re-enter test mode to get the window back.)

If we had configured Currency Converter's window in Interface Builder to retain the resize bar, we could also resize it now. We could also have set the auto-resizing attributes of the window and its views so that the window's objects would resize proportionally to the resized window or would retain their initial size (see Project Builder or Interface Builder Help for details on auto-resizing).

Controls and Text

The buttons and text fields of Currency Converter come with many built-in behaviors. Click the Convert button. Notice how the button is highlighted momentarily.

CC_TestBtn.tiff ¬

If you had buttons of a different style, such as radio buttons, they would also respond in characteristic ways to mouse clicks.

Now click in one of the text fields. See how the cursor blinks in place. Type some text and select it. Use the commands in the Edit menu to copy it and paste it in the other text field.

Do you recall the **nextKeyView** connections we made between Currency Converter's text fields? While a cursor is in a text field, press the Tab key and watch the cursor jump from field to field.

CC_TestTab.tiff ¬

When You Add Menu Commands

An application you design in Interface Builder can acquire extra functionality with the simple addition of a menu command or submenu. You've already seen what you get with the Services and Windows menu, both included by default. You can add other commands and submenus to the main menu for ªfreeº functionality without compilation. For example:

- The Font submenu adds behavior for applying fonts to text in NSText objects, such as the one in the scroll view object in the DataViews palette. Your application gets the Font panel and a font-manager object ªfor free.º
- The Text submenu allows you to align text anywhere there is editable text, and to display a ruler in the NSText object for tabbing, indentation, and alignment.

Many objects that display text or images can print their contents as PostScript data. Later you'll learn how to add the Print menu command and have it invoke this capability.

821009_TableRule.eps ¬

AOpenStepApplicationÑThePossibilitiesi;¬An OpenStep Application Ð The Possibilities

A OpenStep application can do an impressive range of things without a formidable programming effort on your part.

Document Management

Many applications create and manage semi-autonomous objects called *documents*. Documents contain discrete sets of information and support the entry and maintenance of that information. A word-processing document is a typical example. The application coordinates with the user and communicates with its documents to create, open, save, close and otherwise manage them.

The final tutorial is this book describes how to create an application based on a multi-document architecture.

File and Account Management

An application can use the Open panel of the Application Kit to help the user locate files in the file system and open them. It can also make the Save panel available for saving information in files. OpenStep also provides classes for managing files in the file system (creating, comparing, copying, moving, and so forth) and for managing system-account information and user defaults.

Communicating With Other Applications

OpenStep gives an application several ways of communicating information to and from other applications:

- **Pasteboard**: The pasteboard is a global facility for sharing information among applications. Applications can use the pasteboard to hold data that the user has cut or copied and may paste into another application.
- **Services**: Any application can avail itself of the services provided by another application, based on the type of the selected data (such as text). An application can also provide services to other applications such as encryption, language translation, or record fetching.
- **Drag-and-drop**: If your application implements the proper protocol, users can drag objects to and from the interfaces of other applications.

Editing Support

You can get several panels (and associated functionality) when you add a submenu to your application's main menu in Interface Builder. These ªadd-onsº includes the Font panel (and font management), the Color panel (and color management), and, although it's not a panel, the text ruler and the tabbing and indentation capabilities it provides.

Formatter classes enable your application to format numbers, dates, and other types of field values. Support for validating the contents of fields is also available.

Printing and Faxing

With just a simple Interface Builder procedure, OpenStep automates simple printing and faxing of views that contain text or graphics. When a user clicks the control, an appropriate panel helps to configure the print or fax process. The output is WYSIWYG.

Several Application Kit classes give you greater control over the printing of documents and forms, including features such as pagination and page orientation.

Help

You can create a help system for your application using Interface Builder, Project Builder, and an RTF text editor (such as Edit). The Application Kit includes an class for context-sensitive help. If the user clicks an object on the application's interface while pressing a Help key, a small window is displayed containing concise information on the object.

Custom Drawing and Animation

OpenStep lets you create your own custom views that draw their own content and respond to user actions. To assist you in this, OpenStep provides image-compositing and event-handling API as well as PostScript operators, operator functions, and client library functions.

Plug and Play

You can design some applications so that users can incorporate new modules later on. For example, a drawing program could have a tools palette: pencil, brush, eraser, and so on. You could create a new tool and have users install it. When the application is next started, this tool appears in the palette.

425984_TableRule.eps ¬

WhyanObjectisLikeaJellyDonut;¬Why an Object is Like a Jelly Donut

This book depicts objects as filled and segmented ªdonuts.º Why this unlikely shape?

ObjectLabels.eps ¬

This symbol illustrates *data encapsulation*, the essential characteristic of objects. An object consists of both data and procedures for manipulating that data. Other objects or external code cannot access that data directly, but must send messages to the object requesting its data.

An object's procedures (called *methods*) respond to the message and may return data to the requesting object. As the symbol suggests, an object's methods do the encapsulating, in effect mediating access to the object's data. An object's methods are also its interface, articulating the ways in which the object communicates with the world outside it.

The donut symbol also helps to convey the *modularity* of objects. Because an object encapsulates a defined set of data and logic, you can easily assign it to particular duties within a program. Conceptually, it is like a functional unitÐfor instance, ªCustomer RecordºÐthat you can move around on a design board; you can then plot communication paths to and from other objects based on their interfaces.

See the appendix ªObject Oriented Programmingº for a fuller description of data encapsulation, messages, methods, and other properties of objects.

275372_TableRule.eps ¬

TheModel-View-ControllerParadigm;¬The Model-View-Controller Paradigm

A common and useful paradigm for object-oriented applications, particularly business applications, is Model-View-Controller (MVC).Derived from Smalltalk-80, MVC proposes three types of objects in an application, separated by abstract boundaries and communicating with each other across those boundaries.

5Obj_Illo..eps ¬

Model Objects

This type of object represents special knowledge and expertise. Model objects hold a company's data and define the logic that manipulates that data. For example, a Customer object, common in business applications, is a Model object. It holds data describing the salient facts of a customer and has access to algorithms that access and calculate new data from those facts. A more specialized Model class might be one in a meteorological system called Front; objects of this class would contain the data and intelligence to represent weather fronts. Model objects are not displayable. They often are reusable, distributed, persistent, and portable to a variety of platforms.

View Objects

A View object in the paradigam represents something visible on the user interface (a window, for example, or a button). A View object is ªignorantº of the data it displays. The Application Kit usually provides all the View objects you need: windows, text fields, scroll views, buttons, browsers, and so on. But you might want to create your own View objects to show or represent your data in a novel way (for example, a graph view). You can also group View objects within a window in novel ways specific to an application. View objects, especially those in kits, tend to be

very reusable and so provide consistency between applications.

Controller Object

Acting as a mediator between Model objects and View objects in an application is a Controller object. There is usually one per application or window. A Controller object communicates data back and forth between the Model objects and the View objects. It also performs all the application-specific chores, such as loading nib files and acting as window and application delegate. Since what a Controller does is very specific to an application, it is generally not reusable even though it often comprises much of an application's code. (This last statement does not mean, however, that Controller objects *cannot* be reused; with a good design, they can.)

Because of the Controller's central, mediating role, Model objects need not know about the state and events of the user interface, and View objects need not know about the programmatic interfaces of the Model objects. You can make your View and Model objects available to others from a palette in Interface Builder.

Hybrid Models

MVC, strictly observed, is not advisable in all circumstances. Sometimes its best to combine roles. For instance, in a graphics-intensive application, such as an arcade game, you might have several View objects that merge the roles of View and Model. In some applications, especially simple ones, you can combine the roles of Controller and Model; these objects join the special data structures and logic of Model objects with the Controller's hooks to the interface.

A Note on Terminology

The Application Kit and Enterprise Objects Framework reserve special meanings for ªvew objectº and ªmodel .º A view object in the Application Kit denotes a user-interface object that inherits from NSView. In the Enterprise Objects Framework, a model establishes and maintains a

correspondence between an enterprise object class and data stored in a relational database. This book uses ªmodel objectº only within the context of the Model-View-Controller paradigm.

831762_TableRule.eps ¬

ClassVersusObject;¬Class Versus Object

To newcomers to the subject, explanations of object-oriented programming might seem to use the terms ªobjectº and ªclassº interchangeably. Are an object and a class the same thing? And if not, how are they different? How are they related?

An object and a class are both programmatic units. They are closely related, but serve quite different purposes in a program.

First, classes provide a taxonomy of objects, a useful way of categorizing them. Just as you can say a particular tree is a pine tree, you can identify a particular object by its class. You can thereby know its purpose and what messages you can send it. In other words, a class describes the type of an object.

Second, you use classes to generate *instances* Ðor objects. Classes define the data structures and behavior of their instances, and at run time create and initialize these instances.
In a sense, a class is like a factory, stamping out instances of itself when requested.

What especially differentiates a class from its instance is data. A instance has its own unique set of data but its class, strictly speaking, does not. The class defines the structure of the data its instances will have, but only instances can hold data.

A class, on the other hand, implements the behavior of **all** of its instances in a running program. The donut symbol used to represent objects is a bit misleading here, because it suggests that each object contains its own copy of code. This is fortunately not the case; instead of being duplicated, this code is shared among all current instances in the program.

Implicit in the notion of a taxonomy is *inheritance*, a key property of classes. Classes exist in a

hierarchical relationship to one another, with a subclass inheriting behavior and data structures from its superclass, which in turn inherits from its superclass.

See the appendix, ªObject-Oriented Programming,º for more on these and other aspects of classes.

787674_TableRule.eps ¬

PathsforObjectCommunication:Outlets,Targets,andActions;¬Paths for Object Communication: Outlets, Targets, and Actions

Outlets

An outlet is an instance variable that identifies an object.

OandA1.eps ¬

You can communicate with other objects in an application by sending messages to outlets.

An outlet can reference any object in an application: user-interface objects such as text fields and buttons, windows and panels, instances of custom classes, and even the application object itself.

OandA3.eps ¬

Outlets are declared as:

```
id €anObject;
```

You can use **id** as the type for any object; objects with **id** as their type are dynamically typed, meaning that the class of the object is determined at run time. You can statically type an object as a pointer to a class name; you can declare these objects as instance variables, but they are not outlets. What distinguishes outlets is their relationship to Interface Builder.

Interface Builder can ªrecognizeº outlets in code by their declarations, and it can initialize outlets. You usually set an outlet's value in Interface Builder by drawing connection lines between objects. There are ways other than outlets to reference objects in an application, but outlets and Interface Builder's facility for initializing them are a great convenience.

When You Make a Connection in Interface Builder

As with any instance variable, outlets must be initialized at run time to some reasonable valueÐin this case, an object's identifier (**id** value). Because of Interface Builder, an application can initialize outlets when it loads a nib file.

When you make a connection in Interface Builder, a special connector object holds information on the source and destination objects of the connection. (The source object is the object with the outlet.) This connector object is then stored in the nib file. When a nib file is loaded, the application uses the connector object to set the source object's outlet to the identifier of the destination object.

It might help to understand connections by imagining an electrical outlet (as used in the Classes display of the nib file window) embedded in the destination object. Also picture an electrical cord extending from the outlet in the source object Before the connection is made the cord is unplugged and the value of **destination** is undefined; after the connection is made (the cord is plugged in), the id value of the destination object is assigned to the destination outlet.

OandA2.eps ¬

Target/Action in Interface BuilderÐWhat's Going On

As you'll soon find out, you can view (and complete) target/action connections in Interface Builder's Connections inspector. This inspector is easy to use, but the relation of target and action in it might not be apparent. First, **target** is an outlet of a cell object that identifies the recipient of an action message. Well (you say) what's a cell object and what does it have to do

with a button?Ðthat's what I'm making the connection from.

One or more cell objects are always associated with a control object (that is, an object inheriting from NSControl, such as a button). Control objects ªdriveº the invocation of action methods, but they get the target and action from a cell. NSActionCell defines the target and action outlets, and most kinds of cells in the Application Kit inherit these outlets.

OandA4.eps ¬

For example, when a user clicks the Convert button of Currency Converter, the button gets the required information from its cell and sends the message **convert:** to the target outlet, which is an instance of your custom class ConverterController.

In the Actions column of the Connections inspector are all action methods defined by the class of the target object and known by Interface Builder. Interface Builder identifies action methods because their declarations follow the syntax:

```
- (void)doThis:(id)sender;
```

It looks in particular for the argument **sender**.

Which Direction to Connect?

Usually the outlets and actions that you connect belong to a custom subclass of NSObject. For these occasions, you need only follow a couple simple rules to know which way to draw a connection line in Interface Builder:

- To make an action connection, draw a line *to* the custom instance *from* a control object in the user interface, such as a button or a text field.
- To make an outlet connection, draw a line *from* the custom instance *to* another object in the application.

OandA5.eps ¬

Another way to clarify connections is to consider who needs to find whom. With outlets, the custom object needs to find some other object, so the connection is from the custom object to the other object. With actions, the control object needs to find the custom object, so the connection is from the control object.

These are only rules of thumb for the common case, and do not apply in all circumstances. For instance, many OpenStep objects have a **delegate** outlet; to connect these, you draw a connection line from the OpenStep object to your custom object.

766074_TableRule.eps ¬

ObjectiveCQuickReference;¬Objective-C Quick Reference

The Objective C language is a superset of ANSI C with special syntax and run-time extensions that make object-oriented programming possible. Objective C syntax is uncomplicated, but powerful in its simplicity. You can mix standard C and even C++ code with Objective C code.

The following summarizes some of the more basic aspects of the language. See *Object-Oriented Programming and the Objective C Language* for complete details. Also, see ªObject-Oriented Programmingº in the appendix for explanations of terms that are italicized.

Declarations

- Dynamically type objects by declaring them as **id**:

  ```
  id myObject;
  ```

  Since the class of dynamically typed objects is resolved at run time, you can refer to them in your code without knowing beforehand what class they belong to. Type outlets in this way as well as objects that are likely to be involved in *polymorphism* and *dynamic binding*.

- Statically type objects as a pointer to a class:

```
NSString *mystring;
```

  You statically type objects to obtain better compile-time type checking and to make code easier to understand.

- Declarations of *instance methods* begin with a minus sign (-) and, for *class methods*, with a plus sign (+):

```
- (NSString *)countryName;
+ (NSDate *)calendarDate;
```

- Put the type of value returned by a method in parentheses between the minus sign (or plus sign) and the beginning of the method name. (See above example.) Methods returning no explicit type are assumed to return **id**.

- Method argument types are in parentheses and go between the argument's *keyword* and the argument itself:

```
- initWithName:(NSString *)name
€€€€andType:(int)type;
```

  Be sure to terminate all declarations with a semicolon.

- By default, the scope of an instance variable is protected, making that variable directly accessible only to objects of the class that declares it or of a subclass of that class. To make an instance variable private (accessible only within the declaring class), insert the **@private** directive before the declaration.

Messages and Method Implementations

- Methods are procedures implemented by a class for its objects (or, in the case of class methods, to provide functionality not tied to a particular instance). Methods can be public or private; public methods are declared in the class's header file (see above). Messages are invocations of an object's method that identify the method by name.
- Message expressions consist of a variable identifying the receiving object followed by the name of the method you want to invoke; enclose the expression in brackets.

```
[anObject doSomethingWithArg:this];
```

€€€€€€€ receiver€€€€€€method to invoke €€€€€€€€€€€€

As in standard C, terminate statements with a semicolon.

- Messages often get values returned from the invoked method; you must have a variable of the proper type to receive this value on the left side of an assignment.

```
int result = [anObj calcTotal];
```

- You can nest message expressions inside other message expressions. This example gets the window of a form object and makes it the receiving object of another message.

```
[[form window] makeKeyAndOrderFront:self];
```

- A method is structured like a function: After the full declaration of the method comes the body of the implementing code enclosed by braces.
- Use **nil** to specify a null object; this is analogous to a null pointer. Note that some OpenStep methods do not accept **nil** objects as arguments.
- A method can usefully refer to two implicit identifiers: **self** and **super**. Both identify the object receiving a message, but they affect differently how the method implementation is located: **self** starts the search in the receiver's class whereas **super** starts the search in the receiver's superclass. Thus

```
[super init];
```

causes the **init** method of the superclass to be invoked.

- In method you can directly access the instance variables of your class's instances. However, *accessor methods* are recommended instead of direct access, except in cases where performance is of paramount importance. Chapter 4, ªTravel Advisor Tutorial,º describes accessor methods in greater detail.

725445_TableRule.eps ¬

WhatHappensWhenYouBuildanApplication;¬What Happens When You Build an Application

By clicking the Build button in Project Builder, you run the build tool. By default, the build tool is **gnumake**, but it can be any build utility that you specify as a project default in Project Builder. The build tool coordinates the compilation and linking process that results in an executable file. It also performs other tasks needed to build an application.

WhenYouBuild.eps ¬

The build tool manages and updates files based on the dependencies and other information specified in the project's makefiles. Every application project has three makefiles: **Makefile**, **Makefile.preamble**, and **Makefile.postamble**. **Makefile** is maintained by Project BuilderÐdon't edit it directlyÐbut you can modify the other two to customize your build.

The build tool invokes the compiler tool **cc**, passing it the source code files of the project. Compilation of these files (Objective-C, C++, and standard C) produces machine-readable object files for the architecture (or architectures) specified for the build. It puts these files in an architecture-specific subdirectory of **dynamic_obj**.

In the linking phase of the build, the build tool executes the link editor **ld** (via **cc**), passing it the

libraries and frameworks to link against the object files. Frameworks and libraries contain precompiled code that can be used by any application. Linking integrates the code in libraries, frameworks, and object files to produce the application executable file.If there are multiple architecture-specific object files, linking also combines these into a single ªfatº exectuable.

The build tool also copies nib files, sound, images, and other resources from the project to the appropriate localized or non-localized locations in the application wrapper.

An application wrapper is a file package with an extension of ª.appº. A file package is a directory that the Workspace Manager presents to users as a simple file; in other words, it hides the contents of the directory. The ª.appº extension tells the Workspace Manager that the application wrapper contains an executable that can be run (ªlaunchedº) by double-clicking.

203286_TableRule.eps ¬

WhereToGoForHelp;¬Where To Go For Help

Context-Sensitive Application Help

Project Builder and Interface Builder provide context-sensitive help on the details of their use. To activate context-sensitive help, Help-click a control, field, menu command, or other areas of the application. A small window appears that briefly describes the selected object..

The Help key varies by computer architecture. Consult OpenStep user documentation for the Help key on your machine.

IB_ContextHelp.eps ¬

Digital Librarian

Digital Librarian is an application that quickly searches for a word (or other lexical unit) in an on-line manual (or other target) and lists the documents that contain the word. You click a listed item and the document is displayed at the point where the word occurs. The contents of documents are indexed, making searching very fast.

OpenStep includes **NextDeveloper.bshlf**, a Digital Librarian bookshelf for developers in **/NextLibrary/Bookshelves**. This bookshelf contains most of the targets you are likely to want, and includes (as the topmost target) instructions on creating your own bookshelf and customizing it to your needs. When you choose Help from Project Builder or Interface Builder, a Digital Librarian bookshelf is opened that contains the on-line version of OpenStep Development: Tools and Techniques.

You can find Digital Librarian as **Librarian.app** in **/NextApps**.

DLexample.tiff ¬

Project Builder

Project Builder gives you several ways to get the information you need when developing an application.

Project Find: The Project Find panel allows you to search for definitions of, and references to, classes, methods, functions, constants, and other symbols in your project. SInce it is based on project indexing, searching is quick and thorough and leads directly to the relevant code. See Project Builder Help (*OpenStep Development: Tools and Techniques*) for a complete description of Project Find.

Reference Documenation Lookup: If the results of a search using Project Find includes OpenStep symbols, you can easily get related reference documentation that describes that symbol.

Frameworks: Under Frameworks in the project browser, you can browse the header files related to OpenStep frameworks within Project Builder. The Application Kit and Foundation frameworks always are included by default. See chapter 5, ªWhere to Go From Here,º for a fuller description.

_PB_FrameworkDoc.eps ¬

NeXT's Techncial Documentation

Most OpenStep programming documentation is located on-line in the above directory. The document files are in RTF format, so you can open them in Project Builder, Edit, or in most word processors. NeXT includes the following manuals under the **/NextDev** directory:

Reference

- *OpenStep General Reference* (specifications of classes, protocols, functions, types, and constants). This documentation is divided among, and located in, the frameworks NeXT provides, except for information that is common to all frameworks.
- *OpenStep Development Tools Reference* (compiler options, command-line utilities, and so on)
- *NeXT Assembler Manual*

Tasks and Concepts

- *Discovering OPENSTEP: A Developer Tutorial* (this manual)
- *OpenStep Development: Tools and Techniques* (a task-oriented approach to using the development tools that also functions as Help for Project Builder and Interface Builder)
- *Object-Oriented Programming and the Objective C Language*
- *Topics in OpenStep Programming* (concepts and programming procedures)
- *OpenStep Conversion Guide* (step-by-step instructions for converting 3.*x* NEXTSTEP applications to run on OpenStep 4.0.

The **/NextDev** directory also includes release notes. It also contains documentation on the following products, if they're installed: Enterprise Objects Framework, Distributed Objects (DO), Portable Distributed Objects (PDO).

See chapter 5, ªWhere to Go From Here,º for more information on NeXT's technical publications.